Grenoble, France
July 21-23, 2010

Association for Computing Machinery

Advancing Computing as a Science & Profession

PASCO'2010

Proceedings of the 2010 International Workshop on

Parallel Symbolic Computation

Sponsored by:

INRIA, CNRS & Grenoble University

In cooperation with:

SIGSAM

Edited by:

Marc Moreno Maza & Jean-Louis Roch

Proceedings of PASCO 2010: Parallel Symbolic Computation 2010
Editors: Marc Moreno Maza and Jean-Louis Roch
©PASCO 2010
ISBN 978-1-4503-0067-4/10/0007

Foreword to the PASCO 2010 Conference

The International Workshop on Parallel and Symbolic Computation (PASCO) is a series of workshops dedicated to the promotion and advancement of parallel algorithms and software in all areas of symbolic mathematical computation. The pervasive ubiquity of parallel architectures and memory hierarchy has led to the emergence of a new quest for parallel mathematical algorithms and software capable of exploiting the various levels of parallelism: from hardware acceleration technologies (multicore and multi-processor system on chip, GPGPU, FPGA) to cluster and global computing platforms. To push up the limits of symbolic and algebraic computations, beyond the optimization of the application itself, the effective use of a large number of resources (memory and specialized computing units) is expected to enhance the performance multi-criteria objectives: time, energy consumption, resource usage, reliability. In this context, the design and the implementation of mathematical algorithms with provable and adaptive performances is a major challenge.

Earlier meetings in the PASCO series include PASCO'94 (Linz, Austria), PASCO'97 (Maui, U.S.A.), PASCO'07 (London, Canada). PASCO 2010 is affiliated with the 2010 International Symposium on Symbolic and Algebraic Computation (ISSAC) in Munich, Germany. Immediately prior to the ISSAC 2010 meeting, PASCO is held in Grenoble, France.

The workshop PASCO 2010 is a three-day event including invited presentations and tutorials, contributed research papers and a programming contest. The call for papers solicited contributions from areas including:

- Design and analysis of parallel algorithms for computer algebra
- Practical parallel implementation of symbolic or symbolic-numeric algorithms
- High-performance software tools and libraries for computer algebra
- Applications of high-performance computer algebra
- Distributed data-structures for computer algebra
- Hardware acceleration technologies (multi-cores, GPUs, FPGAs) applied to computer algebra
- Cache complexity and cache-oblivious algorithms for computer algebra
- Compile-time and run-time techniques for automating optimization and platform adaptation of computer algebra algorithms

In response, 27 submissions (full papers and extended abstracts) were received, The program committee collected 88 referee reports. After careful consideration, 21 submissions were accepted for presentation and inclusion in the proceedings. In addition, we are grateful that the majority of the invited speakers contributed full papers as well.

We are grateful to all who contributed to the success of our meeting:

- the invited speakers and their co-authors:

Claude-Pierre Jeannerod (France)	Hervé Knochel (France)
Christophe Mouilleron (France)	Christophe Monat (France)
Jean-Michel Muller (France)	Erich L. Kaltofen (USA)
Guillaume Revy (France)	Stephen Lewin-Berlin (USA)
Christian Bertin (France)	Jeremy R. Johnson (USA)
Jingyan Jourdan-Lu (France)	Daniel Kunkle (USA)

- the authors of full papers and extended abstracts;

- the members of the program committee:

Daniel Augot (France)	Anton Leykin (USA)
Jean-Claude Bajard (France)	Gennadi Malaschonok (Russia)
Olivier Beaumont (France)	Michael Monagan (Canada)
Bruce Char (USA)	Winfried Neun (Germany)
Gene Cooperman (USA)	Clément Pernet (France)
Gabriel Dos Reis (USA)	Nicolas Pinto (USA)
Jean-Christophe Dubacq (France)	Manuel Prieto-Matias (Spain)
Jean-Guillaume Dumas (France)	Markus Pueschel (USA)
Jean-Charles Faugère (France)	Nathalie Revol (France)
Matteo Frigo (USA)	David Saunders (USA)
Thierry Gautier (France)	Éric Schost (Canada)
Pascal Giorgi (France)	Wolfgang Schreiner (Austria)
Stef Graillat (France)	Arne Storjohann (Canada)
Jeremy Johnson (USA)	Sivan Toledo (Israel)
Erich Kaltofen (USA)	Gilles Villard (France)
Herbert Kuchen (Germany)	Yuzhen Xie (Canada)
Philippe Langlois (France)	Kazuhiro Yokoyama (Japan)

- the local organizers, all from Grenoble University or the INRIA Grenoble:

Daniel Cordeiro	Clément Pernet
Jean-Guillaume Dumas	Christian Séguy
Thierry Gautier	Ahlem Zammit-Boubaker
Daniele Herzog	

- the anonymous reviewers;

- the supporting organizations:

ACM SIGSAM	University Joseph Fourier
INRIA	Grenoble Institute of Technology
CNRS	LIG and LJK
Grenoble University	ENSIMAG.

Marc Moreno Maza — *London*
Jean-Louis Roch — *Grenoble*
July 4, 2010

Contents

Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors

Claude-Pierre Jeannerod,[*] Christophe Mouilleron,[†] Jean-Michel Muller,[‡] Guillaume Revy[§]
Laboratoire LIP (CNRS, ENS de Lyon, INRIA, UCBL), Université de Lyon, France
Email: firstname.lastname@ens-lyon.fr

Christian Bertin, Jingyan Jourdan-Lu,[¶] Hervé Knochel, Christophe Monat,
STMicroelectronics
Compilation Expertise Center, Grenoble, France
Email: firstname.lastname@st.com

ABSTRACT

Recently, some high-performance IEEE 754 single precision floating-point software has been designed, which aims at best exploiting some features (integer arithmetic, parallelism) of the STMicroelectronics ST200 Very Long Instruction Word (VLIW) processor. We review here the techniques and software tools used or developed for this design and its implementation, and how they allowed very high instruction-level parallelism (ILP) exposure. Those key points include a hierarchical description of function evaluation algorithms, the exploitation of the standard encoding of floating-point data, the automatic generation of fast and accurate polynomial evaluation schemes, and some compiler optimizations.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*standards*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic, parallel algorithms*; G.4 [**Mathematical Software**]: *algorithm design and analysis, parallel and vector implementations*; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers*

[*]INRIA.

[†]ENS de Lyon.

[‡]CNRS.

[§]With LIP and ENS de Lyon when this work has been done, but now a member of ParLab within EECS Department at the University of California at Berkeley. E-mail: grevy@eecs.berkeley.edu

[¶]Also a member of LIP. Email: lujingyan@gmail.com

General Terms

Algorithms, Design, Performance, Reliability

Keywords

binary floating-point arithmetic, correct rounding, IEEE 754, polynomial evaluation, instruction-level parallelism, C software implementation, code generation, VLIW processor

1. INTRODUCTION

Although binary floating-point arithmetic has been standardized since over two decades and is widely used in media-intensive applications, not all embedded media processors have floating-point hardware. An example is the ST231, a 4-way VLIW processor from the STMicroelectronics ST200 family whose native arithmetic consists exclusively of 32-bit integer arithmetic. Consequently, for such so-called *integer* processors floating-point arithmetic must be implemented entirely in software.

Various software implementations of the IEEE 754 standard [1] for binary floating-point arithmetic already exist, such as the SoftFloat package [16]. SoftFloat is written in portable C and could thus be compiled for the ST231 using the ST200 C compiler. While being entirely satisfactory in terms of correctness, such a reference library may fail to exploit some key features of the given target, thus offering standard floating-point support at a possibly prohibitive cost. For example, for the five basic arithmetic operations (addition, subtraction, multiplication, division, and square root) in single precision and with rounding 'to nearest even', we get the following latencies (measured in number of clock cycles):

$+$	$-$	$\times$	$/$	$\sqrt{}$
48 cycles	49 cycles	31 cycles	177 cycles	95 cycles

This motivated the study of how to exploit the various features of the ST231, and resulted in the design and implementation of improved C software for floating-point functionalities gathered in a library called FLIP [14].[1] Compared

[1]The earliest report on this study is [4] and the latest release, FLIP 1.0, is available from `http://flip.gforge.inria.fr/` and delivered under the CeCILL-v2 license.

to SoftFloat, this alternative implementation of IEEE 754 arithmetic allows to obtain speed-ups from 1.5x to 5.2x, the new latencies being as follows (see [38, p. 4]):

$+$	$-$	$\times$	$/$	$\sqrt{\ }$
26 cycles	26 cycles	21 cycles	34 cycles	23 cycles

This study has shown further that similar speed-ups hold not only for rounding to nearest but in fact for all rounding modes and, perhaps more interestingly, that supporting so-called *subnormal numbers* (the tiny floating-point numbers that make gradual underflow possible) has an extra-cost of only a few cycles. Also, square root is now almost as fast as multiplication, and even slightly faster than addition. Finally, the code size (number of assembly instructions) has been reduced by a factor ranging between 1.2 and 4.2 depending on the operator.

To achieve such performances required the combination of various techniques and tools. The main techniques used are a careful exploitation of some nice features of the IEEE standard, a novel algorithmic approach to increase ILP exposure in function evaluation, as well as some compiler optimizations. The software tools used to assist the design and its validation are Gappa, Sollya, and CGPE. The goal of this paper is to review such techniques and tools through a few examples, showing how they can help expose more ILP while keeping code size small and allowing for the validation of the resulting codes. Although the optimizations we present here have been done for the ST231, most of them could still be useful when implementing IEEE floating-point arithmetic on other VLIW integer processors.

The paper is organized as follows. Section 2 provides some reminders about the IEEE 754 floating-point standard (Section 2.1), the architecture of the ST231 processor (Section 2.2), and the associated compiler (Section 2.3). In Section 3 we provide a high-level description of typical floating-point arithmetic implementations, which already allows to expose a great deal of ILP and to identify the most critical subtasks. We then detail three optimization examples: Section 4 presents two ways of taking advantage of the encoding of floating-point data prescribed by the IEEE 754 standard; Section 5 shows how some automatically-generated parallel polynomial evaluation schemes allow for even higher ILP exposure and can be used to accelerate the most critical part of the implementation; Section 6 details an example of optimization done at the compiler level. We conclude in Section 7 with some remarks on the validation of the optimized codes thus produced.

2. BACKGROUND

2.1 IEEE 754 floating-point arithmetic

Floating-point arithmetic is defined rigorously by the IEEE 754 standard [1, 2], whose initial motivation was to make it possible to "write portable software and prove that it worked" [25]. We simply recall some important features of this standard and refer to [6, 17, 34, 35] for in-depth descriptions.

The *data* specified by this standard consist of finite numbers, signed infinities, and quiet or signaling Not-a-Numbers (NaNs). This standard also defines several *formats* characterized by the values of three integers: the radix β, the precision p, and the maximal exponent $e_{\max}$. Although β can be either 2 or 10, we shall restrict here to binary formats, for which $\beta = 2$ and where $e_{\max}$ has the form

$$e_{\max} = 2^{w-1} - 1$$

for some positive integer w; in our implementation examples we shall restrict further to the *binary32* format (formerly called *single precision*), for which $p = 24$ and $e_{\max} = 127 = 2^7 - 1$.

For a given format, finite numbers have the form

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \qquad (1)$$

where

- $s_x \in \{0, 1\}$;
- $e_x \in \{e_{\min}, \dots, e_{\max}\}$ with $e_{\min} = 1 - e_{\max}$;
- $m_x = M_x \cdot 2^{1-p}$ with M_x an integer such that $0 \le M_x < 2^p$ if $e_x = e_{\min}$, and $2^{p-1} \le M_x < 2^p$ if $e_x > e_{\min}$.

The number x in (1) is called *subnormal* if $0 < |x| < 2^{e_{\min}}$, and *normal* if $|x| \ge 2^{e_{\min}}$. In particular, one may check that x is subnormal if and only if $e_x = e_{\min}$ and $1 \le M_x < 2^{p-1}$. Subnormals are a key feature of the IEEE 754 standard, as they allow for gradual rather than abrupt underflow [26]. Another important feature of the above definition of x is that it gives two signed zeros, $+0$ and -0.

Every floating-point datum x can be represented by its standard *encoding* into a k-bit integer

$$X = \sum_{i=0}^{k-1} X_i 2^i, \qquad (2)$$

where $k = p + w$. When x is a finite number, the bits X_i of X must be interpreted as follows:

$$[X_{k-1} \cdots X_0] = [\text{sign} \,|\, \underbrace{\text{biased exponent}}_{w \text{ bits}} \,|\, \underbrace{\text{trailing significand}}_{p-1 \text{ bits}}],$$

where the sign bit equals s_x, the biased exponent equals $e_x + e_{\max}$ for x normal, and 0 for x subnormal, and where the trailing significand contains the fraction bits of m_x. When x is infinite or NaN, special values of X are used, having in particular all the bits of the biased exponent field set to one. In our case, the input and output of the implemented operators correspond to this encoding for $k = 32$. We will see in Section 4 how to take advantage of the nice properties of this encoding to optimize implementations.

Correct rounding is of course another key feature of the IEEE 754 standard: it means that the result to return is the one which would have been produced by first using infinite precision and unbounded exponent range, and then rounding to the target format according to a given rounding direction. The 2008 revision of the standard [2] defines five rounding directions, the default being 'to nearest even.' Correct rounding has been mandatory for the five basic operations since 1985 [1]. Since 2008, this requirement has been extended to several operations and most notably the fused multiply-add (FMA); it is also now recommended that elementary functions (exp, cos, ...) be implemented with correct rounding.

Finally, five *exception flags* must be set after each operation, thus offering a diagnostic of specific behaviors: division by zero, overflow, underflow, inexactness, and non validity of an operation. Although such flags can be very useful when debugging applications, they have not been implemented yet in our context.

2.2 ST200 VLIW architecture

The ST200 is a 4-way VLIW family of cores dedicated to high performance media processing. It was originally designed by Fisher at HPLabs and further developed by STMicroelectronics for use as an IP block in their Systems On Chip (SOCs), such as the 710x family of High Definition TV (HDTV) decoders, where several instances of that core are used for audio and video processing. (We refer to [13] for a comprehensive study of VLIW architectures.)

The flagship processor of this family is the ST231, which has specific architectural features and specific instructions that turn out to be key to support floating-point emulation.

The execution model of the ST200 is the one of a classical VLIW machine where several instructions are grouped in a bundle and execute all at once, with a significant departure from that principle: when a register read-after-write (RAW) dependency is not fulfilled due to the latency of one instruction, the core waits until the result is available for reading. The most important benefit of that feature is that it achieves a very simple form of code compression, avoiding the usage of 'nop' operations that could otherwise be necessary to wait until a result is available.

The ST200 has 64 32-bit general-purpose integer registers and 8 1-bit predicate registers. It has no status flag or similar mechanism: all conditions are computed explicitly either in integer or boolean registers. Boolean registers are then used as predicates for branching instructions, or for specific instructions like addition with carry, or the 'select' instruction that chooses a value among two according to a boolean predicate value. This large set of registers implies that in practice, for the 32-bit code that we consider here, the register pressure is low, and every value can be computed and kept in a register.

The ST200 arithmetic-logic units (ALUs) are replicated for every execution lane available, except for some instructions requiring a relatively large silicon surface to be implemented: for instance four adders are available, meaning that a bundle can execute up to four simultaneous additions, but only two multipliers are implemented, restricting the number of simultaneous multiplications. These ALUs are fully pipelined, and all arithmetic operations have a one cycle latency except the multipliers which have a three cycle latency.

Most ST200 arithmetic instructions can have a 9-bit signed immediate argument, and a specific extension mechanism enables any instruction to have a 32-bit immediate argument by using an extra lane in a bundle. Though this reduces available instruction parallelism, it is an effective mechanism to build large constants, avoiding any access through the data memory, as we will see further.

The two previous features—*two pipelined multipliers* available along with *long immediate operands*—are key for efficient polynomial evaluation; see Section 5.

In addition to the usual operations, the ALUs also implement several specific one-cycle instructions dedicated to code optimization: *leading zero count* (which is key to subnormal numbers support), computation of *minimum* and *maximum*, arbitrary left and right *shifts*, and combined *shift and add*.

The core accesses the memory system through two separated L1 caches: a 32-Kbyte 4-way associative data cache (D-cache), and a 32-Kbyte direct-mapped instruction cache (I-cache).

The direct mapped organization of the I-cache creates a specific difficulty to obtain good and reproducible performance, since its caching performance is very sensitive to the code layout, due to conflict misses. This has been addressed by post link time optimizations [15] that are either driven by heuristics or by actual code profile.

Note also that in the code presented in this article, the choice was made not to use any in-memory table, to avoid any access through the D-cache side of the system, because a cold miss entails a significant performance hit and the prefetch mechanism cannot be efficient on these types of random accesses.

Finally, we conclude this section with a remark about latency and throughput. For instance, on IA64, Markstein [27] takes care to devise two variants of the emulated floating-point operators, one optimized for latency, one optimized for throughput, that are selected by compiler switches. This makes sense for 'open code generation,' where the operator low-level instructions are emitted directly by the compiler in the instruction stream and not available as library runtime support. This is made possible by the fact that the IA64 has a rich instruction set, that enables emulation of complex operations in a few instructions. This technique enables further optimization and scheduling by the compiler, at the cost of a higher code size. On the contrary for the ST200 where emulation of floating-point operations entails a significant code size, our operators are available only in their library variant, thus optimizing for latency is the criteria of choice.

2.3 ST200 VLIW compiler

The ST200 compiler is based on the Open64 technology,[2] open-sourced by SGI in 2001, and then further developed by STMicroelectronics and more generally by the Open64 community.

The Open64 compiler has been re-targeted to support different variants of the ST200 family of cores by a dedicated tool, called the Machine Description System (MDS), providing an automatic flow from the architectural description of the architecture to the compiler and other binary utilities.

Though the compiler has been developed in the beginning to achieve very high performance on embedded media C code, it has been further developed and is able to compile a fully functional Linux distribution, including C++ graphics applications based on WebKit.

It is organized as follows: the gcc-4.2.0 based front-end translates C/C++ source code into a first high level target independent representation called WHIRL, that is further lowered and optimized by the middle-end, including WOPT (WHIRL global Optimizer, based on SSA representations) and optionally LNO (Loop Nest Optimizer). It is then translated in a low-level target dependent representation called CGIR for code generation, including code selection, low level loop transformations, if-conversion, scheduling, and register allocation.

In addition the compiler is able to work in a specific Interprocedural Analysis (IPA) and Interprocedural Optimization (IPO) mode where the compiler builds a representation for a whole program, and optimizes it globally by global propagation, inlining, code cloning, and other optimizations.

Several additions have been done by STMicroelectronics to achieve high performance goals for the ST200 target:

- A dedicated Linear Assembly Optimizer (LAO) is in charge at the CGIR level of software pipelining, pre-

[2]`http://www.open64.net`

pass and post-pass scheduling. It embeds a nearly optimal scheduler based on an Integer Linear Programming (ILP) formulation of the pipelining problem [3]. As the problem instances are very large, a large neighborhood search heuristic is applied as described in [11] and the ILP problem is further solved by an embedded GLPK (GNU Linear Programming Kit) solver.

- A specific if-conversion phase, designed to transform control flow into 'select' operations [7].

- Some additions to the CGIR 'Extended Block Optimizer' (EBO), including a dedicated 'Range Analysis' and 'Range Propagation' phase.

- A proved and efficient out-of-SSA translation phase, including coalescing improvements [5].

Besides efficient code selection, register allocation, and instruction scheduling, the key optimizations contributing to the generation of the low-latency floating-point software are mostly the if-conversion optimization, and to a lesser extent the range analysis framework.

Note also that a compiler intrinsic (__builtin_clz) is used to select the specific 'count leading zero' instruction: this is a small restriction to portability since this builtin is supported in any gcc compiler.

3. HIGH LEVEL DESCRIPTIONS

As recalled in Section 2.1, IEEE 754 floating-point data can be either numbers or NaNs, finite or infinite, subnormal or normal, etc. Such a diversity typically entails many particular cases, and considering each of them separately may slow down both the implementation process and the resulting code.

In order to make the implementation process less cumbersome, a first step can be to systematically define which operands should be considered as *special cases*. This means to exhibit a sufficient condition C_{spec} on the input such that the IEEE result belongs to, say, $\{\text{NaN}, \pm\infty, \pm 0\}$. (For NaNs, this condition should be necessary as well.) Then it remains to perform the following three independent tasks:

(T1) Handle *generic cases* (inputs for which C_{spec} is false);

(T2) Handle special cases;

(T3) Evaluate the condition C_{spec}.

The output is produced by selecting the result of either T1 or T2, depending on the result of T3. These three tasks define our highest-level description of an operator implementation. At this level, ILP exposure is clear. To reduce the overall latency, we optimize T1 first, and then only optimize both T2 and T3 (in particular by reusing as much as we can the intermediate quantities used for T1). This is motivated by the fact that task T1 is the one where actual numerical computations, and especially rounding, take place. Thus, for most operators it can be expected that T1 will dominate the costs, and even allow for both T2 and T3 to be performed meanwhile.

The next level of description corresponds to a more detailed view of task T1. Handling generic input typically involves a range reduction step, an evaluation step on the reduced range, and a reconstruction step [33]. For example, for a unary real-valued operator f, the exact value of f at floating-point number x can always be written

$$f(x) = \pm \ell \cdot 2^d,$$

for some real number $\ell \in [1, 2)$ and some integer d. Here ℓ will in general have the form $\ell = F(s, t, \ldots)$, where F is a function either equal to f or closely related to it, and where $s, t, \ldots$ are parameters that encode both range reduction and reconstruction. In our case, s can be a real, non-rational number and t lies in a range smaller than that of x, like $[0, 1]$. Now assume for simplicity that $d \geq e_{\min}$. (The general case can be handled in a similar way at the expense of suitable scalings.) The correctly-rounded result to be returned has the form

$$r = \pm \text{RN}_p(\ell) \cdot 2^d, \tag{3}$$

where RN_p means rounding 'to nearest even' in precision p.

Task T1 can then be decomposed into three independent sub-tasks: compute the sign s_r of r, the integer d, and the floating-point number $\text{RN}_p(\ell)$. Although $\ell < 2$, its rounded value $\text{RN}_p(\ell)$ can be as large as 2. Again, ILP exposure is explicit at the level within task T1.

For the operators considered here, the most difficult of the three sub-tasks is the computation of $\text{RN}_p(\ell)$. Classically, this sub-task can itself be decomposed into three steps [12], yielding a third level of description: given f and x,

- compute (possibly approximate) values for $s, t, \ldots$;
- deduce from F, s, and t a "good enough" approximation v to ℓ; $(\star)$
- deduce $\text{RN}_p(\ell)$ be applying to v a suitable correction.

Unlike the two previous levels, this level has steps which are fully sequential. The good news, however, is that the computation of v allows many algorithmic choices, some of them leading to very high ILP exposure; see Section 5.

When the binary expansion $(1.\ell_1\ell_2\cdots)_2$ of ℓ is finite, as is the case for the addition and multiplication operators, correction is done via computing explicitly a rounding bit B such that

$$\text{RN}_p(\ell) = (1.\ell_1 \ldots \ell_{p-1})_2 + B \cdot 2^{1-p}. \tag{4}$$

When the expansion of ℓ can be infinite, as for square root or division, the situation is more complicated but one can proceed by correcting "one-sided truncated approximations" [12, 20, 21]. With u the truncated value of v after p bits, the correction to apply is now based on whether $u \geq \ell$ or not. For functions like square root or reciprocal, this predicate can be computed *exactly* by means of their inverse functions. (Note however that this kind of decision problem is much more involved for elementary functions (exp, cos, ...) because of the "tablemaker's dilemma" [33].)

The computation of B or the evaluation of $u \geq \ell$ can in general be simplified when the function f to be implemented has properties like

ℓ cannot be exactly halfway between two consecutive floating-point numbers.

Therefore, in order to get simpler and thus potentially faster rounding procedures, a thorough study of the properties of f in floating-point arithmetic can be necessary. Properties like the one above have been derived in [18, 22] for some commonly used algebraic functions.

4. EXPLOITING STANDARD ENCODINGS

The standard encoding of floating-point data into k-bit integers X as in (2) has several interesting properties, and especially a well-known ordering property (see for example [34, p. 330]). The two examples below show how such properties of the standard encodings of the operand(s) and the result can be exploited to optimize floating-point implementations.

First, the standard encoding can be used to obtain explicit and ready-to-implement formulas for evaluating the condition C_{spec} (task T3). Consider for example the square root operator $x \mapsto \sqrt{x}$. Its IEEE 754 specification implies that

$$C_{\text{spec}} = (x = \pm 0) \ \vee \ (x < 0) \ \vee \ (x = \pm\infty) \ \vee \ (x \text{ is NaN}).$$

A possible implementation of this predicate would thus consist in checking on X if $x = \pm 0$, and so on. However, the following more compact expression was shown in [20]:

$$C_{\text{spec}} = \Big[(X-1) \bmod 2^k \geq 2^{k-1} - 2^{p-1} - 1\Big], \qquad (5)$$

where the notation $[\mathcal{S}]$ means 1 if the statement $\mathcal{S}$ is true, and 0 otherwise. For the binary32 format, this amounts to comparing $(X-1) \bmod 2^{32}$ to the constant value $2^{31} - 2^{23} - 1$. Since on ST231 integer addition is done modulo 2^{32} the above formula can thus be immediately implemented as shown in the C fragment below:

```
Cspec = (X - 1) >= 0x7F7FFFFF;
```

In this case, exploiting the standard encoding allows us to filter out all special cases (task T3) in only 2 cycles and 2 instructions. A similar filter can be designed for addition, multiplication, and division. The overhead due to the fact that these are binary operators is quite reasonable: only 1 more cycle and 3 more instructions are used (see [34, §10]).

As a second example, let us now see how one can exploit the standard encoding at the end of task T1, when packing the result. Let

$$n = RN_p(\ell) = (1.n_1 \dots n_{p-1})_2.$$

For r as in (3), once we have computed its sign s_r as well as n and d, it remains to set up the k-bit integer R that corresponds to the standard encoding of r. One could have concatenated s_r with a biased value

$$D = d + e_{\max}$$

of d and with the fraction bits of n, but removing the leading 1 in n would have increased the critical path. To avoid this, it has been shown in [20] that one may prefer to compute $D-1$ instead of D (at no extra cost since it suffices to modify the value of the bias) and then deduce R as

$$R = \big(s_r \cdot 2^{k-1} + (D-1) \cdot 2^{p-1}\big) + n \cdot 2^{p-1}. \qquad (6)$$

Since the latency of n is in general higher than that of s_r and $D-1$, the evaluation order indicated by the parentheses in (6) may reduce the overall latency of R. Note also that the exponent field is automatically increased by 1 in the case where $n = 2$. In particular, when $n = 2$ and $D - 1 = 2e_{\max} - 1$ then the returned value of R is the encoding of $\pm\infty$, which means that overflow due to rounding is handled transparently thanks to the standard encoding.

Finally, in the cases where n is computed via adding the rounding bit B as in (4), since getting B is the most expensive step, one may rewrite (6) with the following evaluation order:

$$R = \Big(\big(s_r \cdot 2^{k-1} + (D-1) \cdot 2^{p-1}\big) + L\Big) + B, \qquad (7)$$

with L the integer given by $L = (1.\ell_1 \dots \ell_{p-1})_2 \cdot 2^{p-1}$.

5. PARALLEL POLYNOMIAL EVALUATION

When implementing operators like floating-point square root or division, the trickiest part is to write the code computing the approximation v to the real number ℓ; see ($\star$) in Section 3. The goal here is to achieve the lowest possible latency while being "accurate enough" (in a sense made precise in [20], for example $-2^{-p} < \ell - v \leq 0$).

Recall from Section 2.2 that the ST231 can issue up to four integer additions $(A+B) \bmod 2^{32}$ (latency of 1 cycle), up to two multiplications $\lfloor AB/2^{32} \rfloor$ (latency of 3 cycles, pipelined), and that these arithmetic instructions can have 32-bit immediate arguments. These features allow us to consider several methods, such as

- variants of Newton-Raphson and Goldschmidt iterative methods based on low-degree polynomial evaluation followed by 1 or 2 iterations [23, 36];
- methods based on evaluating piecewise, univariate polynomial approximants [19];
- methods based on the evaluation of a single bivariate polynomial [20, 21, 24, 38].

So far the highest ILP exposure and the smallest latencies have been obtained using the third approach: v is obtained by evaluating a very special bivariate polynomial $P(s,t)$ that approximates ℓ "well enough" and has the form

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \qquad a(t) = \sum_{i=0}^{d} a_i t^i.$$

For example, for the square root design introduced in [20], the degree d equals 8 and the coefficients a_i are such that $a_0 = 1$ and, for $1 \leq i \leq 8$, $a_i = (-1)^{i+1} A_i \cdot 2^{-31}$ with A_i a 32-bit positive integer. These numbers as well as a rigorous upper bound on the approximation error entailed by using $P(s,t)$ instead of the true square root function have been produced by the software tool Sollya [9].

Once the polynomial $P(s,t)$ is given, it remains to choose an evaluation scheme that will be fast on ST231 and to bound the rounding errors and check the absence of overflow. Rounding errors come from the fact that each multiplication does not give the full 64-bit product AB but only its highest part $\lfloor AB/2^{32} \rfloor$. For a given evaluation order, this analysis of rounding errors and overflows can be done automatically using the tool Gappa [29, 30, 10].

Because of distributivity and associativity, the number μ_d of all possible evaluation orders of $P(s,t)$ grows extremely fast with the degree d. The first values have been computed recently [31] and, in the above example of square root where $d = 8$, one has

$$\mu_8 = 1055157310305502607244946 \approx 10^{24}$$

different schemes. Among all these schemes, we want one having the lowest latency while satisfying a prescribed rounding error bound.

Since exhaustive search is out of reach, heuristics have been used instead, leading to the design of a tool called

CGPE (Code Generation for Polynomial Evaluation) [37, 38]. CGPE currently implements some heuristics allowing to quickly produce evaluation schemes (in the form of portable C code) that are accurate enough and have reduced latency (and a reduced number of multiplications) on a target like the ST231. It also computes lower bounds on the latency, which made it possible to conclude in our cases (square root, division) that the retained schemes are optimal (square root) or 1 cycle from the optimal (division). The accuracy/overflow certificates are produced by Gappa.

As an example, we show in Figure 1 and Listing 1 the scheme that CGPE has found in the case of square root and the corresponding C code it has generated. Its latency on

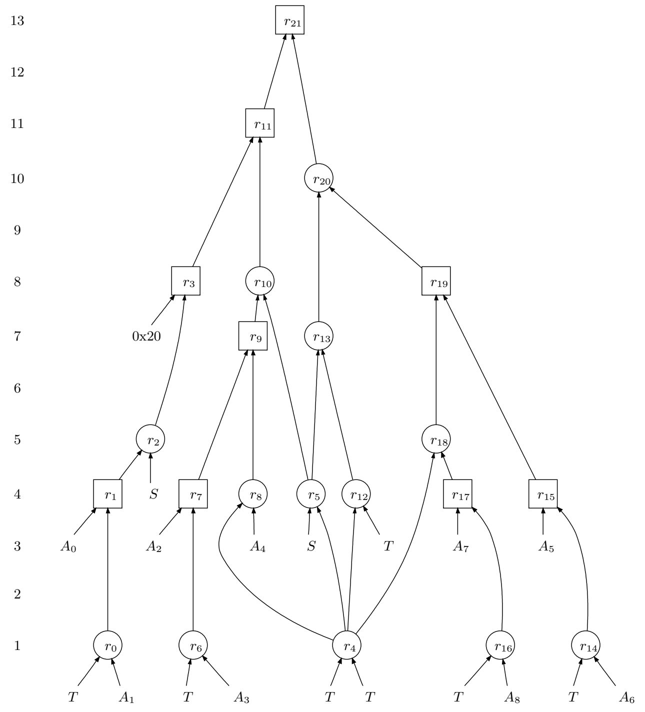


Figure 1: Generated evaluation scheme for square root using the bivariate polynomial approach of [20].

ST231 is of 13 cycles, which matches the computed lower bound. For comparison, Horner's rule, which is fully sequential, takes 36 cycles on ST231. Interestingly, the 13-cycle scheme uses only 4 more multiplications compared to Horner's rule (which is known to be the evaluation order minimizing the number of multiplications [8]).

Finally, it should be noted that the combination of the three tools Sollya, CGPE, and Gappa allows us to tackle more functions than just square root and division, and several other implementations have been [24, 38] or are currently being written using them.

6. COMPILER OPTIMIZATION

As we have seen in Section 2.3, the compiler is key to generate efficient target code. As many techniques are classical or have been described elsewhere (see for example [32]), we will focus on describing the 'Range Analysis' framework, and show a motivating case for its use in a specific optimization linked with the code generation for floating-point support.

Listing 1: Generated C code for the scheme of Fig. 1.

```
uint32_t r0  = mul(T, 0x3ffffafc);
uint32_t r1  = 0x80000007 + r0;
uint32_t r2  = mul(S, r1);
uint32_t r3  = 0x00000020 + r2;
uint32_t r4  = mul(T, T);
uint32_t r5  = mul(S, r4);
uint32_t r6  = mul(T, 0x07f9a6be);
uint32_t r7  = 0x0fff6f59 - r6;
uint32_t r8  = mul(r4, 0x04db72ce);
uint32_t r9  = r7 + r8;
uint32_t r10 = mul(r5, r9);
uint32_t r11 = r3 - r10;
uint32_t r12 = mul(T, r4);
uint32_t r13 = mul(r12, r5);
uint32_t r14 = mul(T, 0x0198e4c7);
uint32_t r15 = 0x0304d2f4 - r14;
uint32_t r16 = mul(T, 0x0019b4c0);
uint32_t r17 = 0x0093fa25 - r16;
uint32_t r18 = mul(r4, r17);
uint32_t r19 = r15 + r18;
uint32_t r20 = mul(r13, r19);
uint32_t r21 = r11 + r20;
```

Range analysis is a variant of the analysis used by constant propagation algorithm [39] operating on SSA where we bound the value (possibly) taken by any variable with a value (range) in a lattice.

We will briefly give an overview of this constant propagation algorithm.

For constant propagation the lattice contains, in addition to initial constant values, the following specific values:

- $\top$ meaning that the variable is unvisited,
- $\bot$ meaning that the value is unknown.

The algorithm proceeds by visiting instructions and lowering the lattice values on the variables according to the following meet ($\sqcap$) rules, as more information is discovered, until a fixed-point is reached:

$$\begin{aligned}&\text{any} \sqcap \top = \text{any}\\ &\text{any} \sqcap \bot = \bot\\ &c_1 \sqcap c_1 = c_1\\ &c_1 \sqcap c_2 = \bot \text{ if } c_1 \neq c_2\end{aligned}$$

In its simplest form, the range analysis uses an extension of the constant propagation lattice: the lattice represents the ranges of possible values of a variable, the meet rules are an extension of constant propagation, and the constant propagation algorithm can be used almost unchanged.

With $[x, y]$ used here to represent the range of integer values between x and y, the following rule replaces the constant rule:

$$[x_1, y_1] \sqcap [x_2, y_2] = [\min(x_1, x_2), \max(y_1, y_2)]$$

A whole family of range analyzes can be defined with slightly different range lattices: for instance analyzes operating on used bit values. In all cases the framework remains the same, only the lattice implementation changes.

It is also useful to have a backward analysis, using the same lattices as for the forward analysis but visiting the instructions backward. This enables for instance the computation of ranges of values needed by the use of a variable, useful to remove useless sign extensions.

In the Open64 compiler, these analyzes are split into generic and target specific parts. First target specific parts are used to handle unusual, target specific instructions (for instance the ST200 clz instruction creates values in the $[0, 32]$ range regardless of its input). Then target independent part acts on generic instruction types based on standard Open64 compiler predicates.

After the range analysis has assigned a value range to each variable, this information is used by the Range Propagation phase to perform various code improvements, that are first target specific, then generic.

For the ST200, the target specific parts include:

- re-selection of specific instructions, such as lower precision multiplication;
- generalized constant folding (cases leading to constant results whereas operands are not constant);
- long to short immediate transformations;
- shift-or transformations using the shift-add ST200 instruction;
- non-wrapping subtract to zero;
- constant result cases for special multiplications.

The generic parts generally tend to create dead code that is further removed by later phases:

- constant propagation (if the variable is found in the range $[x, x]$);
- removal of unnecessary sign/zero extensions;
- removal of unnecessary min/max instructions.

Range propagation is indeed very similar to a 'peepholing' transformation, where the knowledge of ranges on operands of operations enables more powerful and precise transformations.

One interesting example that we implemented in the compiler is the following, an oversimplified piece of code that we generated for emulating the single precision floating-point division:

```
 1 inline
 2 uint32_t minu(uint32_t a, uint32_t b)
 3                    { return a < b ? a : b ; }
 4
 5 uint32_t test5r(uint32_t x, uint32_t y,
 6                               int32_t z) {
 7   int32_t C,u;
 8   if(z>3) u = 6; else u = 1;
 9   C = minu(y,2);
10   return x >> (u+C);
11 }
```

On the ST200, the right shift expression line 10, $x >> (u + C)$ (or similarly with a left shift), can be transformed into $(x >> C) >> u$, which improves the parallelism by relaxing the data dependency on u, provided that the following conditions hold:

$$u \in [0, 31], \quad C \in [0, 31], \quad u + C \in [0, 31].$$

This transformation is done in the target specific part of the range analysis, since obviously we can enable it only when can prove the preconditions.

Then, instead of $x >> (u + C)$ that incurs the following computations:

[1] computation of u
[2] tmp $= u + C$
[3] $x >>$ tmp

we get a potentially better use of ILP (|| here means "in parallel with"):

[1] tmp $= x >> C$ || computation of u
[2] tmp $>> u$

For instance, the above `test5r` function now takes 3 cycles instead of 4 with this optimization.

7. CONCLUDING REMARKS

Let us conclude with remarks on the validation of the numerical quality of the codes produced by the techniques and tools presented so far. Validating a floating-point implementation that claims to be IEEE 754 compliant is often tricky. For the *binary32* format, for which every data can take 2^{32} different values, exhaustive testing is limited to univariate functions. For example, the square root code of FLIP can be compiled with Gcc under Linux and compared exhaustively against the square root functions of GNU C (glibc[3]) or GNU MPFR[4] within a few minutes.

However, this is not possible anymore for bivariate functions like $+$, $-$, $\times$, $/$. To get higher confidence, a first way is to use some existing test programs for IEEE floating-point arithmetic like the *TestFloat* package [16] and, for division in particular, the *Extremal Rounding Tests Set* [28].

A second, complementary way is to get higher confidence, already at the design stage, in the algorithms used for each subtask of the high-level descriptions of Section 3. The techniques and tools that have been reviewed make this possible as follows:

- for the most regular parts of the computation (*i.e.*, parallel polynomial evaluation schemes), we rely on the automatic error analysis functionalities offered by tools like Sollya and Gappa;
- for other subtasks (like special-value filtering and handling, rounding algorithms, computation of the sign and exponent of the result), we rely on proof-and-paper analysis written in terms of the parameters p, k, ... of the format. Typical examples are the symbolic expressions in (5), (6), and (7).

Our experience with the implementation of floating-point arithmetic shows that this kind of symbolic analysis can be really helpful to produce algorithms and codes that are not only faster but also *a priori* safer. Furthermore, establishing properties parameterized by the format should allow to scale easily from, say, binary32 to binary64 implementations. A future direction in this area could be to automate the derivation of such symbolic properties. Another direction is about automatic numerical error analysis: although CGPE produces C codes for polynomial evaluation that have a guaranteed accuracy, we have no guarantee that compilation will preserve the order of evaluation, thus potentially spoiling the accuracy and so the correctness of the whole implementation.

[3]`http://www.gnu.org/software/libc/`
[4]`http://www.mpfr.org/mpfr-current/`

Therefore, another direction is to explore the possibility of certifying numerical accuracy not only at the C level but also at the assembly level.

8. ACKNOWLEDGMENTS

This research was supported by “Pôle de compétitivité mondial” *Minalogic* (Sceptre Project) and by the ANR project *EVA-Flo*. This research effort is going on with an enlarged scope in the *Mediacom* project, which is part of the French government NANO-2012 R&D funding program.

9. REFERENCES

[1] IEEE standard for floating-point arithmetic. IEEE Std. 754-1985, 1985.

[2] IEEE standard for floating-point arithmetic. IEEE Std. 754-2008, pp.1-58, August 2008.

[3] C. Artigues, S. Demassey, and E. Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2008.

[4] C. Bertin, N. Brisebarre, B. D. de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. K. Raina, and A. Tisserand. A Floating-point Library for Integer Processors. In *Proceedings of SPIE* 49*th Annual Meeting International Symposium on Optical Science and Technology, Denver*, volume 5559, pages 101–111. SPIE, Aug. 2004.

[5] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*, pages 114–125. IEEE Computer Society Press, Mar. 2009.

[6] R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Mar. 2010. Version 0.5.1.

[7] C. Bruel. If-conversion for embedded VLIW architectures. *International Journal of Embedded Systems*, 4(1):2–16, 2009.

[8] B. Bürgisser, C. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1997.

[9] S. Chevillard and C. Lauter. Sollya. Available at `http://sollya.gforge.inria.fr/`.

[10] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1), 2009.

[11] B. Dupont de Dinechin. Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. In P. Baptiste, G. Kendall, A. Munier-Kordon, and F. Sourd, editors, *3rd Multidisciplinary International Scheduling conference: Theory and Applications (MISTA)*, 2007. http://www.cri.ensmp.fr/classement/2007.html.

[12] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[13] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[14] FLIP (Floating-point Library for Integer Processors). Available at `http://flip.gforge.inria.fr/`.

[15] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information: dealing with code size expansion. *Journal of Embedded Computing*, 1(4):437–459, 2005.

[16] J. Hauser. The SoftFloat and TestFloat Packages. Available at `http://www.jhauser.us/arithmetic/`.

[17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[18] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 233–240, Adelaide, Australia, 1999.

[19] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Faster floating-point square root for integer processors. In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, July 2007.

[20] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, Laboratoire de l'Informatique du Parallélisme (LIP), 46, allée d'Italie, F-69364 Lyon cedex 07, France, October 2008.

[21] C.-P. Jeannerod, H. Knochel, C. Monat, G. Revy, and G. Villard. A new binary floating-point division algorithm and its software implementation on the ST231 processor. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, Portland, Oregon, USA, June 2009.

[22] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. Technical Report RR2009-26, LIP, Aug. 2009.

[23] C.-P. Jeannerod, S. K. Raina, and A. Tisserand. High-radix floating-point division algorithms for embedded VLIW integer processors. In *17th World Congress on Scientific Computation, Applied Mathematics and Simulation IMACS*, Paris, France, July 2005.

[24] C.-P. Jeannerod and G. Revy. Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores. In *Asilomar'09: Proceedings of the 43rd Asilomar Conference on Signals, Systems, and Computers*, Washington, DC, USA, 2009. IEEE Computer Society.

[25] W. Kahan. IEEE 754: An interview with William Kahan. *Computer*, 31(3):114–115, Mar. 1998.

[26] W. Kahan. A brief tutorial on gradual underflow. Available as a PDF file at `http://www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf`, July 2005.

[27] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[28] D. W. Matula and L. D. McFearin. Extremal rounding test sets. Available at `http://engr.smu.edu/~matula/extremal.html`.

[29] G. Melquiond. Gappa - génération automatique de preuves de propriétés arithmétiques. Available at

`http://lipforge.ens-lyon.fr/www/gappa/`.

[30] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École normale supérieure de Lyon, France, November 2006.

[31] C. Mouilleron. Sequences A173157 and A169608. The On-line Encyclopedia of Integer Sequences (OEIS), February 2010. Available at `http://www.research.att.com/~njas/sequences/A173157`.

[32] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[33] J.-M. Muller. *Elementary functions: algorithms and implementation*. Birkhäuser, second edition, 2006.

[34] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[35] M. L. Overton. *Numerical computing with IEEE floating point arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[36] S.-K. Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, École normale supérieure de Lyon, France, September 2006.

[37] G. Revy. CGPE - Code Generation for Polynomial Evaluation. Available at `http://cgpe.gforge.inria.fr/`.

[38] G. Revy. *Implementation of binary floating-point arithmetic on embedded integer processors: polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, France, December 2009.

[39] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

Fifteen years after DSC and WLSS2 What parallel computations I do today* [Invited Lecture at PASCO 2010]

Erich L. Kaltofen
Dept. of Mathematics, North Carolina State University
Raleigh, North Carolina 27695-8205,USA
kaltofen@math.ncsu.edu
http://www.kaltofen.us

ABSTRACT

A second wave of parallel and distributed computing research is rolling in. Today's multicore/multiprocessor computers facilitate everyone's parallel execution. In the mid 1990s, manufactures of expensive main-frame parallel computers faltered and computer science focused on the Internet and the computing grid. After a ten year hiatus, the Parallel Symbolic Computation Conference (PASCO) is awakening with new vigor.

I shall look back on the highlights of my own research on theoretical and practical aspects of parallel and distributed symbolic computation, and forward to what is to come by example of several current projects. An important technique in symbolic computation is the evaluation/interpolation paradigm, and multivariate sparse polynomial parallel interpolation constitutes a keystone operation, for which we present a new algorithm. Several embarrassingly parallel searches for special polynomials and exact sum-of-squares certificates have exposed issues in even today's multiprocessor architectures. Solutions are in both software and hardware. Finally, we propose the paradigm of interactive symbolic supercomputing, a symbolic computation environment analog of the STAR-P Matlab platform.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algorithms; D.1.3 [Programming Techniques]: Parallel Programming

General Terms: algorithms, experimentation

Keywords: multiprocessor, multicore, memory bus contention, sparse interpolation, supersparse interpolation, parallel search, Lehmer's problem, single factor coefficient bound, interactive supercomputing

*This material is based on work supported in part by the National Science Foundation under Grants CCF-0830347 and DMS-0532140.

1. INTRODUCTION

1.1 A brief history of my research in parallel symbolic computation

In Fall 1982 as a newly minted Ph.D. hired by the theoretical computer science group at the University of Toronto, I was drawn into parallel computation. I shall briefly give an account of my research in parallel and distributed symbolic computation. The new parallel complexity class $\mathcal{NC}$ had been described by Stephen Cook (see [7]) and exact linear algebra problems were shown within it [5], in fact within uniform log-squared circuit depth. Since I worked on the problem of multivariate polynomial factorization, my first parallel result was on factoring polynomials over the complex numbers [24]. Because the integer GCD problem and the more general lattice basis reduction problem were (and are today) not known to be within $\mathcal{NC}$, the factorization of the defining polynomial for the field of definition of the complex factor coefficients had to be delayed until zero divisors appeared. Absolute irreducibility testing, however, was placed cleanly in $\mathcal{NC}$.* Independently, at the same time delayed factorization was used in sequential polynomial computation and called the D5 principle [12].

Work on polynomial matrix canonical forms with M. S. Krishnamoorthy and B. D. Saunders followed [29, 30], introducing random scalar matrix preconditioners for Smith form computation. Our general purpose algorithm for parallelizing straight-line programs for polynomials [47] and rational functions [25, Section 8] also increases the amount of total work of the parallel algebraic circuit, but our randomized algorithm for exact parallel linear system solving [37, 38], based on Wiedemann's algorithm [53], at last was processor-efficient, i.e., had only a poly-log factor more total work. No such solution is known today for Toeplitz systems or other structured systems (cf. [39]—a PASCO 1994 paper).

What was already apparent in the early 1990s, $\mathcal{NC}$ or even processor-efficient poly-logarithmic time algorithms are unrealizable on parallel computers: one does not have $O(n^3)$ processors for symbolic computation

*My paper [24] in its conclusion poses the problem of approximate polynomial factorization, which we only could solve 19 years later [17, 36].

tasks, and that without communication latency, for large n. Our 1991 implementation of a systolic polynomial GCD algorithm [6] on our own MasPar ("Massively Parallel") computer was slower then the sequential code on a top end workstation. The Thinking Machines Corporation filed for bankruptcy in 1994 and bit-wise parallelism disappeared. As far as I can judge, circuits of poly-logarithmic depth are only realized, on chip, for basic arithmetic operations such as look-ahead addition or Wallace-tree multiplication, which can be exploited for packed modular arithmetic with small moduli [15]. However, I still read today, in papers on exponential sequential algorithms in real algebraic geometry, for example, of "parallel polynomial-time" solutions. Those algorithms are not realizable.

The black box model for polynomials [40] and the parallel reconstruction algorithm via sparse interpolation seemed much more practicable—embarrassingly parallel. Thus in 1990 we began developing a run-time support tool, DSC (Distributed Symbolic Computation) [10, 8]. Several features in DSC seem unsupported in commonly used distributed run-time support tools, such as MPI/PVM. DSC could automatically distribute source code, produced possibly by the black box polynomial algorithms, to be *compiled remotely* before execution. There was a (weak) encryption protocol implemented to prevent malicious compilation requests on the remote computers. In today's world-wide hacker-attack environment our protocol no longer can meet required security standards. A second feature tried to model processor loads via time-series models and selectively choose computers with predicted short process queues [49]. We worked with the assumption that DSC was the *only daemon process* making such predictions on the LAN, as multiple such forecasters would possible select the same target computer. It appears to me that some of the instabilities in today's stock markets may be the result of similarly interfering prediction programs.

In the end, even my own Ph.D. students switched from our home made DSC to MPI, and our sparse multivariate polynomial interpolation algorithm in FoxBox is implemented with MPI [9, Section 2.8]. MPI was supported on a parallel computer available to us, the IBM SP-2. Our second application is the block Wiedemann algorithm [27] and Austin Lobo's implementation for entries in $\mathbb{Z}_2$ [45, 34, 35]. Lobo called his library WLSS (pronounced WiLiSyS—Wiedemann's Linear System Solver) and the SP-2 implementation WLSS2, on which systems arising from the RSA factoring challenged could be solved. Parallel symbolic computation had finally become reality. Those papers with Lobo should become, quite unpredictably, my last ones on the subject for 15 years, until this paper.

The second PASCO conference [21], for which I chaired the program committee. was held in Maui, Hawaii, before ISSAC 1997. Hoon Hong, the general chair, had broadened the subject to parallel automatic theorem proving. There were still several (strong) $\mathcal{NC}$ papers. Laurent Bernardin reported on a "massively" parallel implementation of Maple [4]. In contrast, Gaston Gonnet in his 2010 talk at Jo60, the conference in honor of Joachim von zur Gathen's 60th birthday, speculated that Maple's success was due in part that parallelism was *not* pursued in the early years.

On the other hand, Jean-Louis Roch, who attended our workshop "Parallel Algebraic Computation" at Cornell University in 1992 [55], with Thierry Gautier and others fully embraced parallelism and built several general purpose APIs and run-time systems: Athapascan-1 and Kaapi. The multicore multithreaded architectures of today make data parallel programming far beyond vector processing a reality. In 2007 Marc Moreno Maza single-handedly has revived the PASCO conference series. In his own symbolic computation research Moreno Maza systematically deploys parallel APIs, he also uses Cilk. A renaissance has begun. One paper at PASCO 2007 started at exactly the same place where we had left off: a parallel block Wiedemann algorithm [14].

1.2 Overview of results

We describe three separate topics. In Section 2 we investigate the important problem of sparse polynomial interpolation. Interpolation constitutes the reconstruction phase when computing by homomorphic images. The Ben-Or/Tiwari [3] breaks the sequentiality of Zippel's algorithm [54], but term exponent vectors need to be recovered from modular images. We give a method based on the discrete logarithm algorithm modulo primes p with smooth multiplicative orders $p-1$. Our algorithm can handle very large degrees, i.e., supersparse inputs [28], and thus allows for Kronecker substitution from many to a single variable.

In Section 3, we report on our year-long computer search for polynomials with large single factor coefficient bounds and small Mahler measure. Our search was successfully executed on multiprocessor MacPros. A second search, for sum-of-squares lower bound certificates in Rump's model problem [33, 43], for which each process requires several GBs of memory, was less successful on the older multiprocessor MacPros due to memory bus contention. Intel's new "Nehalem" architecture somewhat mitigates those issues.

In Section 4 we introduce the paradigm of interactive symbolic supercomputing.

2. SUPERSPARSE POLYNOMIAL INTERPOLATION

2.1 Term recovery by discrete logarithms

In [9, Section 4.1], as stated above, we implemented a modification of Zippel's [54] variable-by-variable multivariate sparse polynomial interpolation algorithm with modular coefficient arithmetic. The individual black box evaluation in each iteration are carried out in parallel, but the algorithm increases sequentially the number of variables in the interpolants. The algorithm works with relatively small moduli, and our subsequent hybridization [32] of the Zippel and Ben-Or/Tiwari [3] has further reduced the size of the required moduli.

An alternative that interpolates all variables in a single pass but requires a large modulus is already described in [26]. First, the multivariate interpolation problem for a black box polynomial $f(x_1, \ldots, x_n) \in$

$\mathbb{Q}[x_1, \ldots, x_n]$ is reduced to a univariate problem by Kronecker substitution $F(y) = f(y, y^{d_1+1}, y^{(d_1+1)(d_2+1)}, \ldots)$, where d_j is an upper bound on the degree of the black box polynomial in the variable x_j, which either is input or determined by a Monte Carlo method [40, Section 2, Step 1]. Each non-zero term $x_1^{e_{i,1}} \cdots x_n^{e_{i,n}}$ in f is mapped to y^{E_i} in F, where

$$E_i = \sum_j \Big(e_{i,j} \prod_{1 \le k \le j-1} (d_k + 1) \Big).$$

One interpolates $F(y)$ and recovers from the terms y^{E_i} and the bounds d_j all $e_{i,j}$.

The modulus p is then selected as a prime such that $p - 1 > \max_i E_i$ and that $p - 1$ is a smooth integer, i.e., has a factorization into small primes $q_1 \cdots q_l = p - 1$, where $q_1 = 2$. The prime p also must not divide any denominator in the rational coefficients of f, for which the black box call modulo p will throw an exception. For primes with smooth $p - 1$ there is a fast discrete logarithm algorithm [48] and primitive roots $g \in \mathbb{Z}_p$ are quickly constructed, for example by random sampling and testing. Such primes are quite numerous and easy to compute due to a conjectured effective version of Dirichlet's theorem: $\mu Q + 1$ is prime for $\mu = O((\log Q)^2)$ [20]. For instance,

$$37084 = \max_{m \le 12800} (\operatorname{argmin}_{\mu \ge 1}(\mu 2^m + 1 \text{ is prime})).$$

The values $a_k = F(g^k)$, $k = 0, 1, 2, \ldots$ are linearly generated by the minimal polynomial

$$\Lambda(z) = \Big(\prod_{i=1}^{t} (z - g^{E_i}) \bmod p \Big),$$

where t is the number of non-zero terms [3]. First, the $a_k \bmod p$ are computed in parallel via the black box for f for the needed k, and then the generator is determined (see Section 2.2 below). With the early termination strategy the number of terms t and Λ can be computed from $2t+1$ sequence elements a_k by a Monte Carlo algorithm that uses a random g [32]. Second, the polynomial $\Lambda(z)$ is factored into linear factors over $\mathbb{Z}_p$. The factorization is a simple variant of Zassenhaus's method: The $\operatorname{GCD}(\Lambda(z + r), z^{(p-1)/2} - 1)$ for a random residue r splits off about half of the factors. The first division $(z^{(p-1)/2} - 1) \bmod \Lambda(z + r)$ can utilize the factorization of $p - 1$ or simply use repeated squaring, all modulo $\Lambda(z + r)$. The two factors of approximately degree $t/2$ are handled recursively and in parallel. Third, from the linear factors $z - g^{E_i} \bmod p$ the E_i are computed in parallel with Pohlig's and Hellman's discrete logarithm algorithm. Fourth, the term coefficients are computed from the a_k by solving a transposed Vandermonde system [31, Section 5], which essentially constitutes the fine-grain parallel operations of univariate polynomial (tree) multiplication and multipoint evaluation. Fifth, the rational coefficients are determined from the modular images by rational vector recovery [41, Section 4].

A drawback of the above algorithm is the need for a large prime modulus, and this was our reason for implementing Zippel's interpolation algorithm in FoxBox in 1996. Recently, discrete logs modulo word-sized primes could be utilized to recover the term exponents $e_{i,j}$ [23]. We briefly mentioned our idea in an unpublished manuscript on sparse rational function interpolation in 1988 [26], without realizing that the algorithm was actually polynomial in $\log(\deg f)$ under Heath-Brown's conjecture. Algorithms for such supersparse (lacunary) inputs are today a rich research subject, and an unconditional polynomial-time supersparse interpolation algorithm is given in [18]. Supersparse interpolation of shifted univariate polynomials is described in [19], and the difficult problem of supersparse rational function interpolation is solved for a fixed number of terms in [44]. All algorithms have highly speeded parallel variants, which hopefully will become commonly available in symbolic computation systems in the near future.

2.2 Scalar generators via block generators

In Section 2.1, a scalar linear generator was needed for a linearly generated sequence a_k. The classical solution is to deploy the Berlekamp/Massey algorithm [46]. As the new values a_k trickle in, the algorithm can in a concurrent thread iteratively update the current generator candidate. As a specialization of the extended Euclidean algorithm [13] or a specialization of the σ-basis algorithm [2, 32], the classical algorithm seems hard to parallelize, even in a fine-grain shared memory model.

An alternative is to employ the *matrix* Berlekamp/Massey algorithm [11, 42] for purpose of computing a *scalar* linear generator. We shall describe the idea by example. Suppose the degree of the scalar linear recursion is $t = 18$. The scalar Berlekamp/Massey algorithm determines the minimal linear generator $\Lambda(z)$ from the $2t$ sequence elements $a_0, \ldots, a_{35}$. Instead, we use a blocking factor of $b = 2$ and consider the sequence of 2×2 matrices

$$\begin{bmatrix} a_i & a_{9+i} \\ a_{9+i} & a_{18+i} \end{bmatrix}, \quad i = 0, 1, \ldots, 18. \tag{1}$$

Each entry in (1) is linearly generated by Λ, so the scalar generator of the matrix sequence is also Λ. Instead, we compute the minimal right matrix generator. The (infinite) block Hankel matrix

$$\begin{bmatrix} a_0 & a_9 & a_1 & a_{10} & \cdots \\ a_9 & a_{18} & a_{10} & a_{19} & \cdots \\ & & & & \\ a_1 & a_{10} & a_2 & a_{11} & \cdots \\ a_{10} & a_{19} & a_{11} & a_{20} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{2}$$

has rank $t = 18$, which is achieved at the $(t/b) \times (t/b) = 9 \times 9$ blocks (dimensions 18×18), because a row and column permutation transforms the block Hankel matrix into the scalar Hankel matrix which has exactly rank t (cf. [27, Proof of Proposition 3]). Therefore the 2×2 matrix generator polynomial $\Gamma(z)$ has degree 9, whose determinant is $\Lambda(z)$. The latter follows because the highest degree invariant factor of $\Gamma(z)$ is the scalar linear generator [42, Theorem 2] and the degree $\deg(\det \Gamma) = t$ because the determinantal degree equals the rank of (2).

The method uses $b - 1$ more scalar sequence elements, but $\Gamma(z)$ is found after $2\,t/b + 1$ matrix sequence elements. The block algorithm seems to have greater locality as the polynomials are of lower degree. It requires, however, the computation of $\det(\Gamma)$ via interpolation, and therefore may not be competitive with the classical scalar Berlekamp/algorithm for parallel supersparse interpolation, unlike the block Wiedemann linear system solver, where the computation of a_k depends on a_{k-1}. But one could think of a black box polynomial where $F(g^k)$ is derived with the help of $F(g^{k-1})$.

3. MULTIPLE PROCESSORS/CORES–SINGLE MEMORY BUS

In Summer and Fall 2007 we acquired 2 newly Intel-based Apple MacPros with multiple CPUs and multiple cores, on which we installed Ubuntu Linux. At that time we pursued two large computations: first the search for polynomials with large single factor coefficient bounds [33, Remark 4] and the related search for polynomials with small Mahler measure (Lehmer's problem), and second the search for exact SOS certificates in Rump's model problem [33, Section 3.2]. Both searches are embarrassingly parallel and we launched multiple command-line Maple 11 and later Maple 12 processes in Linux detachable "screen"s to achieve full processor utilization.

Our search throughout the year of 2008 for pseudo-cyclotomic polynomials, for which we also used an older Apple G5, yielded no new polynomials. For the record, our own largest single factor bound is given by the irreducible integer polynomial

$$F(z) = a_{37} z^{37} + \sum_{i=0}^{36} a_i (z^{74-i} + z^i) \quad \text{with}$$

$$a_0 = 137244374035256035,\ a_1 = 6484943836830415168,$$
$$a_2 = 15319353170995914 1908,$$
$$a_3 = 241160750720080242 4907,$$
$$a_4 = 284529793856410798 41504,$$
$$a_5 = 268288753013473830301366,$$
$$a_6 = 2105372123573295644409420,$$
$$a_7 = 14138714883963898462151808,$$
$$a_8 = 82921677184320004630302040,$$
$$a_9 = 431329478501438585427465254,$$
$$a_{10} = 2014156747639672791329597498,$$
$$a_{11} = 8526069501131479222465282376,$$
$$a_{12} = 32979342592280651952625919221,$$
$$a_{13} = 117343525840400678593760923162,$$
$$a_{14} = 386220797646892832924725343578,$$
$$a_{15} = 1181540655003118732221772208453,$$
$$a_{16} = 3373539469466421210816098963801,$$
$$a_{17} = 9021882900472427122636284167235,$$
$$a_{18} = 22669166923589015905675502095077,$$
$$a_{19} = 53664552516356435243212903922539,$$
$$a_{20} = 119977087506448109730882947309906,$$
$$a_{21} = 253858560921214782055361032381920,$$
$$a_{22} = 509315086548136054993905167906615,$$
$$a_{23} = 970529828476535410874212141091985,$$
$$a_{24} = 1759154390161310454823643987118589,$$
$$a_{25} = 3036998343927337089144845248619604,$$
$$a_{26} = 4999639546259331050695892101743471,$$
$$a_{27} = 7856622081008872596932525992122154,$$
$$a_{28} = 11795932815522505668032581481982119,$$
$$a_{29} = 16934616571889545324916521185499766,$$
$$a_{30} = 23263087926382581409159452491840837,$$
$$a_{31} = 30596272432934117736562047551265003,$$
$$a_{32} = 38547804638104808779028751533424518,$$
$$a_{33} = 46541915513845439185592821100345340,$$
$$a_{34} = 53870405856198473740160765586055008,$$
$$a_{35} = 59790381075274084971155248471629182,$$
$$a_{36} = 63645538749721902787135528353527656,$$
$$a_{37} = 64984262804935950161468248039123157,$$

where $\|F(z)\|_\infty = \|F(-z)\|_\infty = a_{37}$ and $\|F(z) \cdot F(-z)\|_\infty = 18920209630100132696430504439191918$. Thus the single factor coefficient growth is

$$\frac{\|F(z)\|_\infty}{\|F(z)\, F(-z)\|_\infty} > 3.43464813.$$

The polynomial was constructed from the minimizers in Rump's model problem and we believed it to be the largest single factor bound known as David Boyd's construction only yielded ratios below 3. When presenting our bounds in Summer 2008, John Abbott showed us polynomials with much lower degree and coefficient size. His subsequent paper [1] contains an irreducible F with $\deg(F) = 20$, $\|F\|_\infty = 495$, and $\|F(z)\, F(-z)\|_\infty = 36$. Differences between nearby polynomials with large single (reducible) factor coefficient bounds yield pseudo-cyclic polynomials. Michael Mossinghoff's web site lists the top 100 non-cyclotomic irreducible polynomials with a small Mahler measure, the first being Lehmer's [http://www.cecm.sfu.ca/~mjm/Lehmer/]. In Figure 1 we list how many times we have discovered each of the top 50 polynomials, mostly those of high sparsity. Unfortunately, the search yielded no new polynomials, perhaps because we used relatively low degree minimizing polynomials.

In terms of parallel execution, in our Lehmer polynomials search we achieved a very good utilization of all available 16 cores. The same, surprisingly, was not true for our second search for sum-of-squares certificates for lower bounds in Rump's model problem [43, Section 3]. The main difference is that each SOS searche required a substantial amount of memory, about 5GB, due to the size of the arising matrices in the Newton optimizers. As Table 2 in [43] indicates, for $n = 17$ one command line Maple process required almost twice the time per iteration for a lesser lower bound, that because we executed it concurrently with second such independent command line Maple process since we had sufficient real memory for both. Reminiscent to parallel computing 15 years ago, the slowdown was caused by contention on the memory bus, which may be considered a hardware logic fault. Figure 2 depicts the 2007 MacPro, 4 cores with each L1 I cache: 32K, L1 D cache: 32K, L2 cache: 4096K, with one of the memory cards pulled out. In contrast, Figure 3 shows our new Intel "Nehalem" MacPro, 16 cores with each L1 I cache: 32K, L1 D cache: 32K, L2 cache: 256K, L3 cache: 8192K, and with the 32GB memory card and its massive dual controllers pulled out. Both cabinets have the same size. Note the large gray multipin interface at the bottom of the Nehalem card, which lies sideways on the cabinet. We could verify that for at least 2 processes the memory contention problem was solved.

Figure 1: Michael Mossinghoff's Top 50 pseudo-cyclotomic polynomials

MM's	deg	Mahler measure	count
1.	10	1.176280818260	2248
2.	18	1.188368147508	—
3.	14	1.200026523987	1
4.	18	1.201396186235	8804
5.	14	1.202616743689	105
6.	22	1.205019854225	10
7.	28	1.207950028412	—
8.	20	1.212824180989	4
9.	20	1.214995700776	—
10.	10	1.216391661138	198
11.	20	1.218396362520	1598
12.	24	1.218855150304	—
13.	24	1.219057507826	—
14.	18	1.219446875941	—
15.	18	1.219720859040	—
16.	34	1.220287441693	—
17.	38	1.223447381419	—
18.	26	1.223777454948	—
19.	16	1.224278907222	1779
20.	18	1.225503424104	35
21.	30	1.225619851977	—
22.	30	1.225810532354	—
23.	26	1.226092894512	17
24.	36	1.226493301473	—
25.	20	1.226993758166	194
26.	12	1.227785558695	77
27.	30	1.228140772740	—
28.	36	1.229482810173	—
29.	22	1.229566456617	1
30.	34	1.229999039697	—
31.	38	1.230263271363	—
32.	42	1.230295468643	—
33.	10	1.230391434407	27995
34.	46	1.230743009076	—
35.	18	1.231342769993	—
36.	48	1.232202952743	—
37.	20	1.232613548593	133
38.	28	1.232628775929	—
39.	38	1.233672001767	—
40.	52	1.234348374876	—
41.	24	1.234443834873	—
42.	26	1.234500336789	—
43.	16	1.235256705642	72
44.	46	1.235496042193	—
45.	22	1.235664580390	—
46.	42	1.235761099712	—
47.	32	1.236083368052	—
48.	32	1.236198469859	—
49.	32	1.236227922245	—
50.	40	1.236249557349	—

Figure 2: Dual processor dual core Xeon 3.0GH/7GB 2007 MacPro

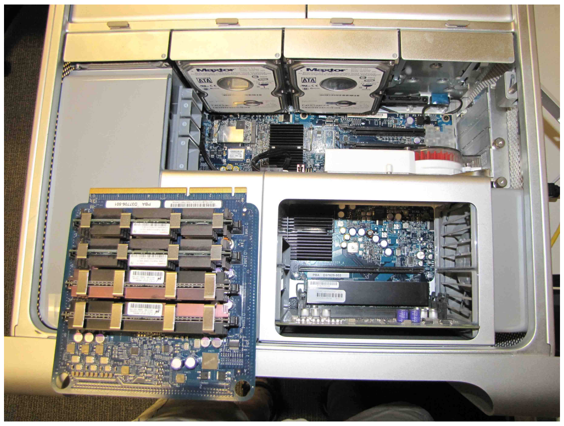


Figure 3: Quad processor quad core Xeon 2.67GH/32GB 2009 Intel Nehalem MacPro

In addition, the authors of [16] report avoidance of bus contention by using Google's cached heap allocation scheme TCmalloc. Perhaps symbolic computation system vendors should also offer software compiled with such malloc schemes. We shall add that we also tried to tune the Maple garbage collection parameters.

Marc Moreno Maza has inquired with Maplesoft in response to our remarks, and a problem area seems to be the memory management strategy of Maple's garbage collector in a setting of parallel independent processes.

4. INTERACTIVE SYMBOLIC SUPERCOMPUTING

During my sabbatical at MIT in Spring 2006, with Alan Edelman we have investigated the use of genericity to create interfaces from symbolic computation platforms to Star-P servers [22] and other parallel implementations.

Laptops and desktops do not have hundreds of processors and large clusters of computers are housed in labs. The Internet makes it possible to access such high performance computers and networks from almost everywhere. The idea of interactive supercomputing is to place the data and computations of a Matlab or Mathematica/Maple/SAGE [52] session remotely on such a compute server and control the session from the local GUI interactively in such a way *that the supercomputing session is indistinguishable from what would be a locally run session.*

Alan Edelman's solution in Matlab is to overload the "`*`" operator and pseudo-postmultiply any value by a global variable `p` of a special type so that the resulting type is a reference to the remote storage. The relevant Matlab functions are then overloaded so that any arguments of the Star-P type delegate execution of the function to the remote supercomputer on the remote data.

In Figure 4 we give a code fragment how Maple's `LinearAlgebra` package could be overloaded for a special Star-P type. We anticipate to experiment with links to parallel implementations of the LinBox exact linear algebra library as soon as they are available. We add that the INTER*CTIVE supercomputing company was recently acquired by Microsoft. Note that SAGE's philosophy is to place its user interface above the symbolic computation system, while Star-P places the interface underneath it.

Acknowledgments: I thank Jean-Guillaume Dumas for sharing with me his experience with the memory bus contention problem and TCmalloc, and Marc Moreno Maza for his reviewing comments.

5. REFERENCES

[1] Abbott, J. Bounds on factors in Z[x]. *Mathematics Research Repository abs/0904.3057* (2009). URL: http://arxiv.org/abs/0904.3057.

[2] Beckermann, B., and Labahn, G. A uniform approach for fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Applic. 15*, 3 (July 1994), 804–823.

[3] Ben-Or, M., and Tiwari, P. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. Twentieth Annual ACM Symp. Theory Comput.* (New York, N.Y., 1988), ACM Press, pp. 301–309.

[4] Bernardin, L. Maple on a massively parallel, distributed memory machine. In Hitz and Kaltofen [21], pp. 217–222.

[5] Borodin, A., von zur Gathen, J., and Hopcroft, J. E. Fast parallel matrix and GCD computations. *Inf. Control 52* (1982), 241–256.

[6] Brent, R. P., and Kung, H. T. Systolic VLSI arrays for linear-time GCD computation. In *Proc. VLSI '83* (1983), pp. 145–154.

[7] Cook, S. A. A taxonomy of problems with fast parallel algorithms. *Inf. Control 64* (1985), 2–22.

[8] Díaz, A., Hitz, M., Kaltofen, E., Lobo, A., and Valente, T. Process scheduling in DSC and the large sparse linear systems challenge. *J. Symbolic Comput. 19*, 1–3 (1995), 269–282. URL: EKbib/95/DHKLV95.pdf.

[9] Díaz, A., and Kaltofen, E. FoxBox a system for manipulating symbolic objects in black box representation. In *Proc. 1998 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'98)* (New York, N. Y., 1998), O. Gloor, Ed., ACM Press, pp. 30–37. URL: EKbib/98/DiKa98.pdf.

[10] Díaz, A., Kaltofen, E., Schmitz, K., and Valente, T. DSC A system for distributed symbolic computation. In *Proc. 1991 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'91)* (New York, N. Y., 1991), S. M. Watt, Ed., ACM Press, pp. 323–332. URL: EKbib/91/DKSV91.ps.gz.

[11] Dickinson, B. W., Morf, M., and Kailath, T. A minimal realization algorithm for matrix sequences. *IEEE Trans. Automatic Control* AC-*19*, 1 (Feb. 1974), 31–38.

[12] Dicrescenzo, C., and Duval, D. *Le système D5 de calcul formel avec des nombres algébriques.* Univ. Grenoble, 1987, Doctoral Thesis by Dominique Duval, Chapter 1. Jean Della-Dora (Thesis Adisor).

[13] Dornstetter, J. L. On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE Trans. Inf. Theory* IT-*33*, 3 (1987), 428–431.

[14] Dumas, J.-G., Elbaz-Vincent, P., Giorgi, P., and Urbanska, A. Parallel computation of the rank of large sparse matrices from algebraic K-theory. In *PASCO'07 Proc. 2007 Internat. Workshop on Parallel Symbolic Comput.* (2007), pp. 43–52.

[15] Dumas, J.-G., Fousse, L., and Salvy, B. Simultaneous modular reduction and Kronecker substitution for small finite fields. *J. Symbolic Comput. to appear* (2010).

[16] Dumas, J.-G., Gautier, T., and Roch, J.-L. Generic design of Chinese remaindering schemes. In *PASCO'10 Proc. 2010 Internat. Workshop on Parallel Symbolic Comput.* (New York, N. Y., 2010), M. Moreno Maza and J.-L. Roch, Eds., ACM.

[17] Gao, S., Kaltofen, E., May, J. P., Yang, Z., and Zhi, L. Approximate factorization of multivariate polynomials via differential equations. In *ISSAC 2004 Proc. 2004 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 2004), J. Gutierrez, Ed., ACM Press, pp. 167–174. ACM SIGSAM's ISSAC 2004 Distinguished Student Author Award (May and Yang). URL: EKbib/04/GKMYZ04.pdf.

[18] Garg, S., and Schost, Éric. Interpolation of polynomials given by straight-line programs. *Theoretical Computer Science 410*, 27-29 (2009), 2659 – 2662.

[19] Giesbrecht, M., and Roche, D. S. Interpolation of shifted-lacunary polynomials. *Computing Research Repository abs/0810.5685* (2008). URL: http://arxiv.org/abs/0810.5685.

[20] Heath-Brown, D. R. Almost-primes in arithmetic progressions and short intervals. *Math. Proc. Camb. Phil. Soc. 83* (1978), 357–375.

Figure 4: Overloading a Maple package procedure

```
> LAstarp:=module()
>    export RandomMatrix;
>    local __RandomMatrix;
>    unprotect(LinearAlgebra:-RandomMatrix);
>    __RandomMatrix := eval(LinearAlgebra:-RandomMatrix);
>    LinearAlgebra:-RandomMatrix:=proc()
>       if type(args[1],string) then RETURN("Calling starp with " || args);
>                                 else RETURN(__RandomMatrix(args));
>       fi;
>    end; # RandomMatrix
> end; # LAstarp
                   LAstarp := module() local __RandomMatrix; export RandomMatrix;  end module
> LinearAlgebra:-RandomMatrix(2,2,generator=0..1.0);
                                 [0.913375856139019393    0.905791937075619225]
                                 [                                            ]
                                 [0.126986816293506055    0.814723686393178936]
> LinearAlgebra:-RandomMatrix("overloading");
                                        "Calling starp with overloading"
```

[21] Hitz, M., and Kaltofen, E., Eds. *Proc. Second Internat. Symp. Parallel Symbolic Comput. PASCO '97* (New York, N. Y., 1997), ACM Press.

[22] INTER*CTIVE supercomputing. Star-P overview. Web page, 2008. URL http://www.interactivesupercomputing.com/.

[23] Javadi, S. M. M., and Monagan, M. On sparse polynomial interpolation over finite fields. Manuscript, 2010.

[24] Kaltofen, E. Fast parallel absolute irreducibility testing. *J. Symbolic Comput. 1*, 1 (1985), 57–67. Misprint corrections: *J. Symbolic Comput.* vol. 9, p. 320 (1989). URL: EKbib/85/Ka85_jsc.pdf.

[25] Kaltofen, E. Greatest common divisors of polynomials given by straight-line programs. *J. ACM 35*, 1 (1988), 231–264. URL: EKbib/88/Ka88_jacm.pdf.

[26] Kaltofen, E. Unpublished article fragment, 1988. URL http://www.math.ncsu.edu/~kaltofen/bibliography/88/Ka88_ratint.pdf.

[27] Kaltofen, E. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comput. 64*, 210 (1995), 777–806. URL: EKbib/95/Ka95_mathcomp.pdf.

[28] Kaltofen, E., and Koiran, P. Finding small degree factors of multivariate supersparse (lacunary) polynomials over algebraic number fields. In *ISSAC MMVI Proc. 2006 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 2006), J.-G. Dumas, Ed., ACM Press, pp. 162–168. URL: EKbib/06/KaKoi06.pdf.

[29] Kaltofen, E., Krishnamoorthy, M. S., and Saunders, B. D. Fast parallel computation of Hermite and Smith forms of polynomial matrices. *SIAM J. Alg. Discrete Math. 8* (1987), 683–690. URL: EKbib/87/KKS87.pdf.

[30] Kaltofen, E., Krishnamoorthy, M. S., and Saunders, B. D. Parallel algorithms for matrix normal forms. *Linear Algebra and Applications 136* (1990), 189–208. URL: EKbib/90/KKS90.pdf.

[31] Kaltofen, E., and Lakshman Yagati. Improved sparse multivariate polynomial interpolation algorithms. In *Symbolic Algebraic Comput. Internat. Symp. ISSAC '88 Proc.* (Heidelberg, Germany, 1988), P. Gianni, Ed., vol. 358 of *Lect. Notes Comput. Sci.*, Springer Verlag, pp. 467–474. URL: EKbib/88/KaLa88.pdf.

[32] Kaltofen, E., and Lee, W. Early termination in sparse interpolation algorithms. *J. Symbolic Comput. 36*, 3–4 (2003), 365–400. Special issue Internat. Symp. Symbolic Algebraic Comput. (ISSAC 2002). Guest editors: M. Giusti & L. M. Pardo. URL: EKbib/03/KL03.pdf.

[33] Kaltofen, E., Li, B., Yang, Z., and Zhi, L. Exact certification of global optimality of approximate factorizations via rationalizing sums-of-squares with floating point scalars. In *ISSAC 2008* (New York, N. Y., 2008), D. Jeffrey, Ed., ACM Press, pp. 155–163. URL: EKbib/08/KLYZ08.pdf.

[34] Kaltofen, E., and Lobo, A. Distributed matrix-free solution of large sparse linear systems over finite fields. In *Proc. High Performance Computing '96* (San Diego, CA, 1996), A. M. Tentner, Ed., Society for Computer Simulation, Simulation Councils, Inc., pp. 244–247. Journal version in [35]. URL: EKbib/96/KaLo96_hpc.pdf.

[35] Kaltofen, E., and Lobo, A. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica 24*, 3–4 (July–Aug. 1999), 331–348. Special Issue on "Coarse Grained Parallel Algorithms". URL: EKbib/99/KaLo99.pdf.

[36] Kaltofen, E., May, J., Yang, Z., and Zhi, L. Approximate factorization of multivariate polynomials using singular value decomposition. *J. Symbolic Comput. 43*, 5 (2008), 359–376. URL:

EKbib/07/KMYZ07.pdf.

[37] Kaltofen, E., and Pan, V. Processor efficient parallel solution of linear systems over an abstract field. In *Proc. SPAA '91 3rd Ann. ACM Symp. Parallel Algor. Architecture* (New York, N.Y., 1991), ACM Press, pp. 180–191. URL: EKbib/91/KaPa91.pdf.

[38] Kaltofen, E., and Pan, V. Processor-efficient parallel solution of linear systems II: the positive characteristic and singular cases. In *Proc. 33rd Annual Symp. Foundations of Comp. Sci.* (Los Alamitos, California, 1992), IEEE Computer Society Press, pp. 714–723. URL: EKbib/92/KaPa92.pdf.

[39] Kaltofen, E., and Pan, V. Parallel solution of Toeplitz and Toeplitz-like linear systems over fields of small positive characteristic. In *Proc. First Internat. Symp. Parallel Symbolic Comput. PASCO '94* (Singapore, 1994), H. Hong, Ed., World Scientific Publishing Co., pp. 225–233. URL: EKbib/94/KaPa94.pdf.

[40] Kaltofen, E., and Trager, B. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput. 9*, 3 (1990), 301–320. URL: EKbib/90/KaTr90.pdf.

[41] Kaltofen, E., and Yang, Z. On exact and approximate interpolation of sparse rational functions. In *ISSAC 2007 Proc. 2007 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 2007), C. W. Brown, Ed., ACM Press, pp. 203–210. URL: EKbib/07/KaYa07.pdf.

[42] Kaltofen, E., and Yuhasz, G. On the matrix Berlekamp-Massey algorithm, Dec. 2006. Manuscript, 29 pages. Submitted.

[43] Kaltofen, E. L., Li, B., Yang, Z., and Zhi, L. Exact certification in global polynomial optimization via sums-of-squares of rational functions with rational coefficients, Jan. 2009. Accepted for publication in J. Symbolic Comput. URL: EKbib/09/KLYZ09.pdf.

[44] Kaltofen, E. L., and Nehring, M. Supersparse black box rational function interpolation, Jan. 2010. Manuscript, 23 pages.

[45] Lobo, A. A. *Matrix-Free Linear System Solving and Applications to Symbolic Computation*. PhD thesis, Rensselaer Polytechnic Instit., Troy, New York, Dec. 1995.

[46] Massey, J. L. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory* IT-*15* (1969), 122–127.

[47] Miller, G. L., Ramachandran, V., and Kaltofen, E. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput. 17*, 4 (1988), 687–695. URL: EKbib/88/MRK88.pdf.

[48] Pohlig, C. P., and Hellman, M. E. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Inf. Theory* IT-*24* (1978), 106–110.

[49] Samadani, M., and Kaltofen, E. Prediction based task scheduling in distributed computing. In *Proc. 14th Annual ACM Symp. Principles Distrib. Comput.* (New York, N. Y., 1995), ACM Press, p. 261. Brief announcement of [51, 50].

[50] Samadani, M., and Kaltofen, E. On distributed scheduling using load prediction from past information. Unpublished paper, 1996.

[51] Samadani, M., and Kaltofen, E. Prediction based task scheduling in distributed computing. In *Languages, Compilers and Run-Time Systems for Scalable Computers* (Boston, 1996), B. K. Szymanski and B. Sinharoy, Eds., Kluwer Academic Publ., pp. 317–320. Poster session paper of [50]. URL: EKbib/95/SaKa95_poster.ps.gz.

[52] Stein, W., et al. SAGE: Open Source mathematics software. Web page, Feb. 2008. URL http://www.sagemath.org.

[53] Wiedemann, D. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory* IT-*32* (1986), 54–62.

[54] Zippel, R. Interpolating polynomials from their values. *J. Symbolic Comput. 9*, 3 (1990), 375–403.

[55] Zippel, R. E., Ed. *Proc. 2nd Internat. Workshop on Computer Algebra and Parallelism* (Heidelberg, Germany, 1992), vol. 584 of *Lect. Notes Comput. Sci.*, Springer Verlag.

Exploiting Multicore Systems with Cilk

[Extended Abstract]

Stephen Lewin-Berlin
Quanta Research Cambridge
One Kendall Square B2201, Cambridge, MA 02139

Categories and Subject Descriptors

D.3.2 [**Software**]: Programming Languages

General Terms

Languages

Keywords

Cilk, Spawn, Sync, Hyperobject

1. INTRODUCTION

The increasing prevalence of multicore processors has led to a renewed interest in parallel programming. Cilk is a language extension to C and C++ designed to simplify programming shared-memory multiprocessor systems. The work-stealing scheduler in Cilk is provably efficient and maintains well-defined space bounds. [1, 2] A deterministic program (that is, a race-free Cilk program that uses no lock constructs) maintains serial semantics, and such a Cilk program running on P processors will use no more than P times the stack space required by the corresponding serial program.

In Cilk, parallelism is expressed using three keywords: `cilk_spawn`, `cilk_sync`, and `cilk_for`. These keywords describe the logical parallel structure of the program, and leave it to the runtime system to dynamically schedule the parallel work onto the available processors. A `cilk_spawn` indicates that the "spawned" routine is allowed (but not required) to run in parallel with the continuation of the routine. A `cilk_sync` creates a barrier, requiring all spawned routines in the syntactic scope of the current procedure to return before execution continues beyond the sync. An implicit `cilk_sync` exists at the end of each procedure. The `cilk_for` keyword indicates a parallel loop, allowing the iterations of the loop to run in parallel.

The work-stealing scheduler in the runtime system adapts well to dynamic conditions, creating a parallel model that allows for composability of parallel Cilk modules without performance degradation or increased system resource demands regardless of the number of available processors.

Cilk programs retain a property that we call "serializability". That is, if the `cilk_spawn` and `cilk_sync` keywords are removed, and `cilk_for` replaced by the standard "for" keyword, the resulting program has the same semantics as the Cilk program. We call this the "serialization" of the Cilk program.

Cilk programs do not require multiple processors to run, and the execution of a Cilk program on a single processor is equivalent to the execution of the serialization. Cilk keywords introduce minimal overhead, allowing the serialization to run nearly as fast as the execution of single-processor parallel program.

The strict fork-join nature of Cilk permits a structured analysis of the parallel performance of a Cilk program, and a performance model based on "work" and "span" has been developed. The "Cilkview" tool [4] has been developed to analyze and display the parallel performance and to predict how a Cilk program will scale to run on multiple processors.

Typically, the most difficult aspect of debugging parallel programs is to identify and resolve race conditions. The "Cilkscreen" tool identifies race conditions exposed in the dynamic execution of a Cilk program. Several techniques exist to resolve such race conditions, including a new data construct that we call "hyperobjects" [3].

In this talk, I will provide a brief introduction to Cilk programming, describing the use of the Cilk keywords and addressing performance implications of various programming strategies. I will explain the work/span performance model, and show how the Cilkview tool can be used to identify performance bottlenecks and to predict how performance will scale.

I will also describe a kind of hyperobject that we call a "reducer" and explain how reducers are implemented in the Cilk runtime system. I will show several examples of reducers and illustrate how reducers can be used to write lock-free deterministic programs whose output would otherwise be schedule dependent.

The version of the Cilk system described in this talk is the commercially released Cilk++ system developed by Cilk Arts between 2006 and 2009. In August of 2009, Cilk Arts was acquired by Intel Corporation.

2. REFERENCES

[1] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of*

computing, pages 362–371, New York, NY, USA, 1993. ACM.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[3] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90, New York, NY, USA, 2009. ACM.

[4] Charles Leiserson Yuxiong He and William Leiserson. The cilkview scalability analyzer. In *SPAA '10: Proceedings of the twenty-second annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2010. ACM.

Automated Performance Tuning

[Extended Abstract]

Jeremy R. Johnson
Computer Science Dept.
Drexel University
Philadelphia, PA, USA
jjohnson@cs.drexel.edu

ABSTRACT

This tutorial presents automated techniques for implementing and optimizing numeric and symbolic libraries on modern computing platforms including SSE, multicore, and GPU. Obtaining high performance requires effective use of the memory hierarchy, short vector instructions, and multiple cores. Highly tuned implementations are difficult to obtain and are platform dependent. For example, Intel Core i7 980 XE has a peak floating point performance of over 100 GFLOPS and the NVIDIA Tesla C870 has a peak floating point performance of over 500 GFLOPS, however, achieving close to peak performance on such platforms is extremely difficult. Consequently, automated techniques are now being used to tune and adapt high performance libraries such as ATLAS (`math-atlas.sourceforge.net`), PLASMA (`icl.cs.utk.edu/plasma`) and MAGMA (`icl.cs.utk.edu/magma`) for dense linear algebra, OSKI (`bebop.cs.berkeley.edu/oski`) for sparse linear algebra, FFTW (`www.fftw.org`) for the fast Fourier transform (FFT), and SPIRAL (`www.spiral.net`) for wide class of digital signal processing (DSP) algorithms. Intel currently uses SPIRAL to generate parts of their MKL and IPP libraries.

Categories and Subject Descriptors

D.1.2 [**Software**]: Programming Techniques, Automatic Programming; G.4 [**Mathematics of Computing**]: Mathematical Software, Efficiency; I.1.3 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation, Languages and Systems

General Terms

Algorithms, Performance

Keywords

Code generation and optimization, high-performance computing, vectorization, parallelism, autotuning

1. INTRODUCTION

The first part of the tutorial will review implementation techniques from high-performance computing [1] including SSE, multicore, and GPU processors and survey techniques for automated performance tuning of linear algebra [3, 12] and FFT kernels [7, 11]. The second part of the tutorial presents a detailed case study using the Walsh-Hadamard transform, a simple DSP transform related to the FFT[8], and uses symbolic formula manipulation to optimize code.

2. WALSH-HADAMARD TRANSFORM

The Walsh–Hadamard Transform is the matrix–vector product $y = \mathbf{WHT}_N \cdot x$ where, $N = 2^n$,

$$\mathbf{WHT}_N = \bigotimes_{i=1}^{n} \mathbf{WHT}_2 = \overbrace{\mathbf{WHT}_2 \otimes \cdots \otimes \mathbf{WHT}_2}^{n}, \qquad (1)$$

$$\mathbf{WHT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad (2)$$

and $\otimes$ is the Kronecker product. A large class of algorithms for computing the **WHT** can be derived by factoring the **WHT** matrix into a product of sparse structured matrices.

Let $N = N_1 \cdots N_t$, where $N_i = 2^{n_i}$, then

$$\mathbf{WHT}_N = \prod_{i=1}^{t} \left(\mathrm{I}_{N_1 \cdots N_{i-1}} \otimes \mathbf{WHT}_{N_i} \otimes \mathrm{I}_{N_{i+1} \cdots N_t} \right). \qquad (3)$$

Two special cases of Equation 3 correspond to the standard recursive and iterative algorithms

$$\mathbf{WHT}_{2^n} = (\mathbf{WHT}_2 \otimes \mathrm{I}_{2^{n-1}})(\mathrm{I}_2 \otimes \mathbf{WHT}_{2^{n-1}}), \qquad (4)$$

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{n} (\mathrm{I}_{2^{i-1}} \otimes \mathbf{WHT}_2 \otimes \mathrm{I}_{2^{n-i}}). \qquad (5)$$

3. AUTOTUNING

Let $x^N_{b,s}$ denote the vector access pattern $[x_b, x_{b+s}, \ldots, x_{b+(N-1)s}]$. Then the evaluation of $\mathbf{WHT}_N \cdot x$ using Equation 3 can be expressed as a triply nested loop

```
$R = N, \quad S = 1$
for $i = 1, \ldots, t$
  $R = R/N_i$
  for $j = 0, \ldots, R-1$
    for $k = 0, \ldots, S-1$
      $x^{N_i}_{jN_iS+k,S} = \mathbf{WHT}_{N_i} \cdot x^{N_i}_{jN_iS+k,S}$
  $S = S \cdot N_i$,                                    (6)
```

where $\mathbf{WHT}_{N_i}$ is an algorithm for computing smaller size transforms. The computation of each $\mathbf{WHT}_{N_i}$ can be evaluated recursively in the same fashion or directly with unrolled code. Implementations are tuned to a given platform, trading off iteration, recursion, straight-line code and different memory access patterns, by intelligently searching over the space of formulas.

4. VECTORIZATION

Vectorized implementations of the **WHT** are obtained through the observation that formulas of the form $A \otimes \mathrm{I}_\nu$ can be interpreted as vector operations on vectors of length ν [10, 6]. For example, $y = (\mathbf{WHT}_2 \otimes \mathrm{I}_2)x$, corresponds to a vector add and subtract

$$\begin{aligned} y_0^2 &= x_0^2 + x_2^2 \\ y_2^2 &= x_0^2 - x_2^2 \end{aligned}$$

The key challange is to convert formulas into this form while maintaining efficient memory access patterns. For example, using properties of the tensor product and stride permutations Equation 4 can be manipulated into the vector form with $\nu = 2$.

$$\begin{aligned} \mathbf{WHT}_N &= (\mathbf{WHT}_2 \otimes \mathrm{I}_{N/2})(\mathrm{I}_2 \otimes \mathbf{WHT}_{N/2}) \\ &= ((\mathbf{WHT}_2 \otimes \mathrm{I}_{N/4}) \otimes \mathrm{I}_2)L_2^N(\mathbf{WHT}_{N/2} \otimes \mathrm{I}_2)L_{N/2}^N \\ &= ((\mathbf{WHT}_2 \otimes \mathrm{I}_{N/4}) \otimes \mathrm{I}_2) \\ &\quad (L_2^{N/2} \otimes \mathrm{I}_2)(\mathrm{I}_{N/4} \otimes L_2^4) \\ &\quad (\mathbf{WHT}_{N/2} \otimes \mathrm{I}_2) \\ &\quad (\mathrm{I}_{N/4} \otimes L_2^4)(L_{N/4}^{N/2} \otimes \mathrm{I}_2). \end{aligned}$$

5. PARALLELIZATION

Formula manipulation techniques can be used to obtain load-balanced shared memory parallel implementations without false sharing [10, 2, 5, 4]. For example, assuming p processors and cache line of size μ

$$\begin{aligned} \mathbf{WHT}_N &= (\mathbf{WHT}_R \otimes \mathbf{WHT}_S) \\ &= ((L_R^{Rp} \otimes \mathrm{I}_{S/p\mu}) \otimes \mathrm{I}_\mu) \\ &\quad (\mathrm{I}_p \otimes (\mathbf{WHT}_R \otimes \mathrm{I}_{S/p}) \\ &\quad ((L_p^{Rp} \otimes \mathrm{I}_{S/p\mu}) \otimes \mathrm{I}_\mu) \\ &\quad (\mathrm{I}_p \otimes (\mathrm{I}_{R/p} \otimes \mathbf{WHT}_S) \end{aligned}$$

consists of parallel operations $\mathrm{I}_p \otimes A$ and permutations that move entire cache blocks. Similar techniques extend to distributed memory and streaming computations [10, 9].

6. REFERENCES

[1] S. Chellappa, F. Franchetti, and M. Püschel. How to write fast numerical code: A small introduction. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 5235 of *Lecture Notes in Computer Science*, pages 196–259. Springer, 2008.

[2] K. Chen and J. R. Johnson. A prototypical self-optimizing package for parallel implementation of fast signal transforms. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 36, Washington, DC, USA, 2002. IEEE Computer Society.

[3] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, volume 92, pages 293–312, 2005.

[4] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on Signal Processing on Platforms with Multiple Cores*, 26(6):90–102, 2009.

[5] F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: Smp and multicore. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 115, New York, NY, USA, 2006. ACM.

[6] F. Franchetti, Y. Voronenko, and M. Püschel. A rewriting system for the vectorization of signal transforms. In *VECPAR '06: Proceedings of the 7th international conference on High performance computing for computational science*, pages 363–377, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, volume 92, pages 216–231, 2005.

[8] J. Johnson and M. Püschel. In search of the optimal Walsh-Hadamard transform. In *ICASSP '00: Proceedings of the Acoustics, Speech, and Signal Processing, 2000. on IEEE International Conference*, pages 3347–3350, Washington, DC, USA, 2000. IEEE Computer Society.

[9] J. R. Johnson and K. Chen. A self-adapting distributed memory package for fast signal transforms. In *IPDPS '04: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 44, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9(4):449–500, 1990.

[11] M. Püschel, J. M. F. Moura, D. Padua J. R. Johnson, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for DSP transforms. In *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, volume 92, pages 232–275, 2005.

[12] K. Yotov, G. Ren X. Li, M. J. Garzaran, D. Padua K. Pingali., and P. Stodghill. Is search really necessary to generate high-performance BLAS. In *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, volume 92, pages 358–386, 2005.

Roomy: A System for Space Limited Computations

Daniel Kunkle
Northeastern University
360 Huntington Ave.
Boston, Massachusetts 02115
kunkle@ccs.neu.edu

ABSTRACT

There are numerous examples of problems in symbolic algebra in which the required storage grows far beyond the limitations even of the distributed RAM of a cluster. Often this limitation determines how large a problem one can solve in practice. Roomy provides a minimally invasive system to modify the code for such a computation, in order to use the local disks of a cluster or a SAN as a transparent extension of RAM.

Roomy is implemented as a C/C++ library. It provides some simple data structures (arrays, unordered lists, and hash tables). Some typical programming constructs that one might employ in Roomy are: map, reduce, duplicate elimination, chain reduction, pair reduction, and breadth-first search. All aspects of parallelism and remote I/O are hidden within the Roomy library.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features — *Dynamic storage management*; E.1 [**Data Structures**]: *Distributed data structures*

General Terms: Algorithms, Languages

Keywords: parallel, disk-based, programming model, open source library

1. INTRODUCTION

This paper provides a brief introduction to Roomy [1], a new programming model and open source library for parallel disk-based computation. The primary purpose of Roomy is to solve space limited problems without significantly increasing hardware costs or radically altering existing algorithms and data structures.

Roomy uses disks as the main working memory of a computation, instead of RAM. These disks can be disks attached to a single shared-memory system, a storage area network (SAN), or the locally attached disks of a compute cluster. Particularly in the case of using the local disks of a cluster, disks are often underutilized and can provide several order of magnitude more working memory than RAM for essentially no extra cost.

There are two fundamental challenges in using disk-based storage as main memory:

Bandwidth: roughly, the bandwidth of a single disk is 50 times less than that of a single RAM subsystem (100 MB/s versus 5 GB/s). The solution is to use many disks in parallel, achieving an aggregate bandwidth comparable to RAM.

Latency: even worse than bandwidth, the latency of disk is many orders of magnitude worse than RAM. The solution is to avoid latency penalties by using streaming data access, instead of costly random access.

Roomy hides from the programmer both the complexity inherent in parallelism and the techniques needed to convert random access patterns into streaming access patterns. In doing so, the programming model presented to the user closely resembles that of traditional RAM-based serial computation.

Some previous work that provided a foundation for the development of Roomy include: the use of parallel disks to prove that 26 moves suffice to solve Rubik's Cube [2]; and disk-based methods for enumerating large implicit state spaces [4, 5]. The largest Roomy-based project to date is a package for manipulating large binary decision diagrams [3], which appears in the same proceedings as this paper, and includes an experimental analysis of the package.

The rest of this paper briefly describes the data structures provided by Roomy, some example programming constructs that can be implemented using Roomy, and some general performance considerations. Complete documentation, and instructions for obtaining the Roomy open source library, can be found on the Web at `roomy.sourceforge.net`.

2. ROOMY DATA STRUCTURES

Roomy data structures are transparently distributed across many disks, and the operations on these data structures are transparently parallelized across the many compute nodes of a cluster. Currently, there are three Roomy data structures:

`RoomyArray`: a fixed size, indexed array of elements (elements can be as small as one bit).

`RoomyHashTable`: a dynamically sized structure mapping *keys* to *values*.

`RoomyList`: a dynamically sized, unordered list of elements.

There are two types of Roomy operations: delayed and immediate. If an operation requires random access, it is delayed. Otherwise, it is performed immediately. To initiate the processing of delayed operations for a given Roomy data structure, the programmer makes an explicit call to *synchronize* that data structure. By delaying random access operations they can be collected and performed more efficiently in batch.

Table 1 describes some of the basic Roomy operations. Some operations are specific to one type of Roomy data structure, while others apply to all three. The operations are also identified as either immediate (I) or delayed (D).

For performance reasons, it is often best to use a `RoomyArray` or `RoomyHashTable` instead of a `RoomyList`, where possible. Computations using `RoomyList`s are often dominated by the time to sort the list and any delayed operations. `RoomyArray`s and `RoomyHashTable`s avoid sorting by organizing data into *buckets*, based on indices or keys.

3. PROGRAMMING CONSTRUCTS

Because Roomy provides data structures and operations similar to traditional programming models, many common programming constructs can be implemented in Roomy without significant modification. The one major difference is in the use of delayed random operations. To ensure efficient computation, it is important to maximize the number of delayed random operations issued before they are executed (by calling `sync` on the data structure).

Below are Roomy implementations of six programming constructs: map, reduce, set operations, chain reduction, pair reduction, and breadth-first search. Both map and reduce are primitive operations in Roomy. The others are built using Roomy primitives.

First, note that the code given here uses a simplified syntax. For example, the `doUpdate` method from the *chain reduction* programming construct below would be implemented in Roomy as:

```
void doUpdate(uint64 localIndex, void* localVal,
              void* remoteVal) {
  *(int*)localVal =
      *(int*)localVal + *(int*)remoteVal;
}
```

The simplified version given here eliminates the type casting, and appears as:

```
int doUpdate(int localIndex, int localVal,
             int remoteVal) {
  return localVal + remoteVal;
}
```

A future C++ version of Roomy is planned that would use templates to make the simplified version legal code.

See the online Roomy documentation and API [1] for the exact syntax and function definitions.

Map.

The `map` operator applies a user-defined function to every element of a Roomy data structure. As an example, the following converts a `RoomyArray` into a `RoomyHashTable`, with array indices as keys and the associated elements as values.

```
RoomyArray ra;       // elements of type T
RoomyHashTable rht; // pairs of type (int, T)

// Function to map over RoomyArray ra.
void makePair(int i, T element) {
  RoomyHashTable_insert(rht, i, element);
}

// Perform map, then complete delayed inserts
RoomyArray_map(ra, makePair);
RoomyHashTable_sync(rht);
```

Reduce.

The `reduce` operator produces a result based on a combination of all elements in a data structure. It requires two user-defined functions. The first combines a partially computed result and an element of the list. The second combines two partially computed results. The order of reductions is not guaranteed. Hence, these functions must be associative and commutative, or else the result is undefined.

As an example, the following computes the sum of squares of the elements in a `RoomyList`.

```
RoomyList rl;  // elements of type int

// Function to add square of an element to sum.
int mergeElt(int sum, int element) {
  return sum + element * element;
}

// Function to compute sum of two partial answers.
int mergeResults(int sum1, int sum2) {
  return sum1 + sum2;
}

int sum =
    RoomyList_reduce(rl, mergeElt, mergeResults);
```

The type of the result does not necessarily have to be the same as the type of the elements in the list, as it is in this case. For example, the result could be the k largest elements of the list.

Set Operations.

Roomy can support certain set operations through the use of a `RoomyList`. Some of these operations (particularly intersection) are sub-optimal when built using the current set of primitives. Future work is planned to add a native `RoomySet` data structure.

A `RoomyList` can be converted to a set by removing duplicates.

```
RoomyList A;  // can contain duplicate elements
RoomyList_removeDupes(A);  // now a set
```

Performing set union, $A = A \cup B$, is also simple.

```
RoomyList A, B;
RoomyList_addAll(A, B);
RoomyList_removeDupes(A);
```

Set difference, $A = A - B$, is performed by using just the `removeAll` operation, assuming A and B are already sets.

```
RoomyList A, B;
RoomyList_removeAll(A, B);
```

Finally, set intersection is implemented as a union, followed set differences: $C = (A+B) - (A-B) - (B-A)$. Set intersection may become a Roomy primitive in the future.

Table 1: Some basic Roomy operations. If an operation is specific to one type of data structure, it is listed under `RoomyArray`, `RoomyHashTable`, or `RoomyList`. Otherwise, it is listed as "common to all". Also, the type of each operation is given as either immediate (I) or delayed (D).

Data Structure	Name	Type	Description
`RoomyArray`	`access`	D	apply a user-defined function to an element
	`update`	D	update an element using a user-defined function
`RoomyHashTable`	`insert`	D	insert a given (key, value) pair in the table
	`remove`	D	given a key, remove the corresponding (key, value) pair from the table
	`access`	D	given a key, apply a user-defined function to the corresponding value
	`update`	D	given a key, update a the corresponding value using a user-defined function
`RoomyList`	`add`	D	add a single element to the list
	`remove`	D	remove all occurrences of a single element from the list
	`addAll`	I	adds all elements from one list to another
	`removeAll`	I	removes all elements in one list from another
	`removeDupes`	I	removes duplicate elements from a list
Common to all	`sync`	I	process all outstanding delayed operations for the data structure
	`size`	I	returns the number of elements in the data structure
	`map`	I	applies a user-defined function to each element
	`reduce`	I	applies a user-defined function to each element and returns a value (e.g. the ten largest elements of the list)
	`predicateCount`	I	returns the number of elements that satisfy a given property (Note: this does not require a separate scan, the count is kept current as the data is modified)

```
// input sets
RoomyList A, B;
// initially empty sets
RoomyList AandB, AminusB, BminusA, C;

// create three temporary sets
RoomyList_addAll(AandB, A);
RoomyList_addAll(AandB, B);
RoomyList_removeDupes(AandB);
RoomyList_addAll(AminusB, A);
RoomyList_removeAll(AminusB, B);
RoomyList_addAll(BminusA, B);
RoomyList_removeAll(BminusA, A);

// compute intersection
RoomyList_addAll(C, AandB);
RoomyList_removeAll(C, AminusB);
RoomyList_removeAll(C, BminusA);
```

Chain Reduction.

Chain reduction combines each element in a sequence with the element after it. In this example, we compute the following function for an array of integers `a` of length `N`

```
for i = 1 to N-1
  a[i] = a[i] + a[i-1]
```

where all array elements on the right-hand side are accessed before updating any array elements on the left-hand side.

In the following code, `val_i` represents `a[i]` and `val_iMinus1` represents `a[i-1]`.

```
RoomyArray ra;  // array of ints, length N

// Function to complete updates
int doUpdate(int i, int val_i, int val_iMinus1) {
  return val_i + val_iMinus1;
}

// Function to be mapped over ra, issues updates
void callUpdate(int iMinus1, int val_iMinus1) {
  int i = iMinus1 + 1;
  if i < N
    RoomyArray_update(
        ra, i, val_iMinus1, doUpdate);
}

RoomyArray_map(ra, callUpdate); // issue updates
RoomyArray_sync(ra);            // complete updates
```

The computation is deterministic. The new array values are based only on the old array values because Roomy guarantees that none of the delayed update operations are executed until `sync` is called. The code above is implemented internally through a traditional scatter-gather operation.

Parallel Prefix.

The chain reduction programming construct can also be used as the basis for a parallel prefix computation. At a high level, the parallel prefix computation is defined as

```
for (k = 1; k < N; k = k * 2)
  if i-k >= 0
    a[i] = a[i] + a[i-k];
```

Pair Reduction.

Pair reduction applies a function to each pair of elements in a collection. For an array `a` of length `N`, pair reduction is defined as

```
for i = 0 to N-1
  for j = 0 to N-1
    f(a[i], a[j]);
```

The following example inserts each pair of elements from a `RoomyArray` into a `RoomyList`. The variable `outerVal` represents `a[i]` and the variable `innerVal` represents `a[j]`.

```
RoomyArray ra;  // array of int, length N
RoomyList rl;   // list containing Pair(int, int)

// Access function, adds a pair to  the list
void doAccess(int innerIndex, int innerVal,
              int outerVal) {
```

```
  RoomyList_add(
      rl, new Pair(innerVal, outerVal));
}

// Map function, sends access to all other elts
void callAccess(int outerIndex, int outerVal) {
  for innerIndex = 0 to N-1
    RoomyArray_access(
        ra, innerIndex, outerVal, doAccess);
}

RoomyArray_map(ra, callAccess);
RoomyArray_sync(ra);  // perform delayed accesses
RoomyList_sync(rl);   // perform delayed adds
```

One can think of the `RoomyArray_map` method as the outer loop, the `callAccess` method as the inner loop, and the `doAccess` method as the function being applied to each pair of elements.

Breadth-first Search.

Breadth-first search enumerates all of the elements of a graph, exploring elements closer to the starting point first. In this case, the graph is implicit, defined by a starting element and a generating function that returns the neighbors of a given element.

```
// Lists for all elts, current, and next level
RoomyList* all = RoomyList_make("allLev", eltSize);
RoomyList* cur = RoomyList_make("lev0", eltSize);
RoomyList* next = RoomyList_make("lev1", eltSize);

// Function to produce next level from current
void genNext(T elt) {
  /* User-defined code to compute neighbors ... */
  for nbr in neighbors
    RoomyList_add(next, nbr);
}

// Add start element
RoomyList_add(all, startElt);
RoomyList_add(cur, startElt);

// Generate levels until no new states are found
while(RoomyList_size(cur)) {
  // generate next level from current
  RoomyList_map(cur, genNext);
  RoomyList_sync(next);

  // detect duplicates within next level
  RoomyList_removeDupes(next);

  // detect duplicates from previous levels
  RoomyList_removeAll(next, all);

  // record new elements
  RoomyList_addAll(all, next);

  // rotate levels
  RoomyList_destroy(cur);
  cur = next;
  next = RoomyList_make(levName, eltSize);
}
```

One of the initial tests of Roomy was to use breadth-first search to solve the *pancake sorting problem.* Pancake sorting operates using a sequence of prefix reversals (reversing the order of the first k elements of the sequence). The sequence can be thought of as a stack of pancakes of varying sizes, with the prefix reversal corresponding to flipping the top k pancakes. The goal of the computation is to determine the number of reversals required to sort any sequence of length n.

Using Roomy, the entire application took less than one day of programming and less than 200 lines of code. Breadth-first search was implemented using a `RoomyArray`, similar to the `RoomyList`-based version presented above. It was able to solve the 13-pancake problem in 70 minutes using the locally attached disks of a 30 node cluster.

Three different solutions to the pancake sorting problem, each using one of the three Roomy data structures, is available in the Roomy online documentation [1].

4. PERFORMANCE CONSIDERATIONS

Parallelism: It is anticipated that most applications will use one Roomy process per compute node. In some cases, however, disk bandwidth may not be fully utilized by a single process. In this case, each compute node can start several Roomy processes. Alternatively, the user application itself can be multi-threaded, as long as a single thread issues all Roomy operations. A future version of Roomy is planned that provides full multi-threading support.

Maximum data structure size: The maximum size of a Roomy data structure is limited only by the aggregate available disk space. The use of load balancing techniques ensures that each Roomy process stores approximately the same amount of data. In cases where compute nodes have significantly different amounts of free space, aggregate space can be increased by starting additional Roomy processes on those nodes with more space.

Choice of data structure: Roomy uses two primary methods for converting random access patterns into streaming access: buckets and sorting. The bucket-based method is used by `RoomyArray`s and `RoomyHashTable`s. This method splits the data structure into RAM-sized *chunks*, and co-locates delayed operations with the corresponding chunk. The sorting method is used by `RoomyList`s, where there is no index or key that can be used to define buckets. In this case, both the list and the delayed operations are maintained in a sorted order. Because the cost of sorting often dominates the running time of `RoomyList`-based programs, it is recommended that a `RoomyArray` or `RoomyHashTable` be used instead, where possible.

5. ACKNOWLEDGMENTS

Many thanks to Gene Cooperman and Vlad Slavici, whose input has greatly improved both Roomy and this paper.

6. REFERENCES

[1] Daniel Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. `http://roomy.sourceforge.net/`.

[2] Daniel Kunkle and Gene Cooperman. Harnessing parallel disks to solve Rubik's cube. *Journal of Symbolic Computation*, 44(7):872–890, 2009.

[3] Daniel Kunkle, Vlad Slavici, and Gene Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *Parallel Symbolic Computation (PASCO '10)*. ACM Press, 2010.

[4] Eric Robinson. *Large Implicit State Space Enumeration: Overcoming Memory and Disk Limitations.* PhD thesis, Northeastern University, Boston, MA, 2008.

[5] Eric Robinson, Daniel Kunkle, and Gene Cooperman. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Parallel Symbolic Computation (PASCO '07)*, pages 78–87. ACM Press, 2007.

Generic design of Chinese remaindering schemes

Jean-Guillaume Dumas[*]
Université de Grenoble
Laboratoire Jean Kuntzmann,
umr CNRS 5224, BP 53X,
F38041 Grenoble, France.
Jean-Guillaume.Dumas@imag.fr

Thierry Gautier[†] Jean-Louis Roch
Université de Grenoble
Laboratoire LIG
51, av. Jean Kuntzmann,
F38330 Montbonnot, France.
Thierry.Gautier@inrialpes.fr,JLRoch@imag.fr

ABSTRACT

We propose a generic design for Chinese remainder algorithms. A Chinese remainder computation consists in reconstructing an integer value from its residues modulo coprime integers. We also propose an efficient linear data structure, a radix ladder, for the intermediate storage and computations. Our design is structured into three main modules: a black box residue computation in charge of computing each residue; a Chinese remaindering controller in charge of launching the computation and of the termination decision; an integer builder in charge of the reconstruction computation. We show that this design enables many different forms of Chinese remaindering (for example deterministic, early terminated, distributed, etc.); easy comparisons between these forms and user-transparent parallelism at different parallel grains.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Algorithms*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Algorithm design and analysis*

General Terms

Algorithms, Design, Performance, Experimentation.

Keywords

Chinese remainder; generic design; early termination; radix ladder; parallel modular arithmetic.

[*]Part of this work was done while the first author was visiting the Claude Shannon Institute and the University College Dublin, Ireland, under a CNRS grant.

[†]Part of this work was done while the second author was visiting the ArTeCS group of the University Complutense, Madrid, Spain.

1. INTRODUCTION

Modular methods are largely used in computer algebra to reduce the cost of coefficient growth of the integer, rational or polynomial coefficients. Chinese remaindering (or interpolation) can be used to recover the large coefficients from their modular evaluations by reconstructing an integer value from its residues modulo coprime integers.

LinBox[1][9] is an exact linear algebra library providing some of the most efficient methods for linear systems over arbitrary precision integers.

For instance, to compute the determinant of a large dense matrix over the integers one can use linear algebra over word size finite fields [10] and then use a combination of system solving and Chinese remaindering to lift the result [13].

The Frobenius normal form and the characteristic polynomial of a matrix can be used to test two matrices for similarity [25]. The Smith normal form of an integer matrix is useful, for example, in the computation of homology groups and its computation can be done via the integer minimal polynomial [12]. In both cases, the polynomials are computed first modulo several prime numbers and then reconstructed via Chinese remaindering using precise bounds on the integer coefficients of the integer characteristic or minimal polynomials [18, 8].

A first approach is to use deterministic remaindering using a priori bounds on the reconstructed output. An alternative is to terminate the reconstruction early when the actual integer result is smaller than the estimated bound [14, 12, 20]. If the reconstruction stabilizes for some modular iterations, the computation is stopped and gives the correct answer with high probability. Early termination can also be used in a deterministic way, e.g. for integer polynomial factorization, when the norms of the reconstructed output satisfy some inequality. In probabilistic and deterministic early, the computational complexity becomes output-sensitive.

We propose in section 2 a linear space data structure enabling fast computation of Chinese reconstruction, alternative to subproduct trees. In section 3 we structure the design of a generic pattern of Chinese remaindering into three main modules: a black box residue computation in charge of computing each residue; a Chinese remaindering controller in charge of launching the computation and of the termination decision; an integer builder in charge of the reconstruction computation. We show in section 4 that this design enables many different forms of Chinese remaindering (deterministic, early terminated, distributed, etc.) and easy comparisons between these forms. Finally, in section 5 we provide

[1]`http://linalg.org`

an easy and efficient user-transparent parallelism at different parallel grains using this design. Any parallel paradigm can be implemented, provided that it fulfills the defined controller interface. We use KAAPI[2][16] to show the efficiency of our approach on distributed/shared architectures and compare it to OPENMP[3].

2. RADIX LADDER: LINEAR STRUCTURE FOR FAST CHINESE REMAINDERING

2.1 Generic reconstruction

We are given a black box function which computes the evaluation of an integer R modulo any number m (often a prime number).

To reconstruct R, we must have enough evaluations $r_j \equiv R \mod m_j$ modulo coprimes m_j. To perform this reconstruction, we need two by two liftings with $U \equiv R \mod M$ and $V \equiv R \mod N$ as follows:

$$R_{MN} = U + (V - U) \times (M^{-1} \mod N) \times M. \qquad (1)$$

We will need this combination most frequently in two different settings: when M and N have the same size, and when N is of size 1. The first generic aspect of our development is that for both cases, the same implementation can be fast.

We first need a complexity model. We do not give much details on fast integer arithmetic in this paper, instead our point is to show the genericity of our approach and that it facilitates experiments in order to obtain practical efficiency independently of the underlying arithmetic. Therefore we propose to use a very simplified model of complexity where division/inverse/modulo/gcd are slower than multiplication. We denote by $d_\alpha \ell^\alpha$ the complexity of the gcd of integers of size ℓ with $1 < \alpha \le 2$, and ranging from $O(\ell^2)$ for classical multiplication to $O(\ell^{1+\epsilon})$ for FFT-like algorithms. Then the cost of the division and of modular multiplication is also bounded by $d_\alpha \ell^\alpha$. Moreover, there exists m_α such that the complexity of integer multiplication of size ℓ can be bounded by $m_\alpha \ell^\alpha$ (e.g. $m_2 = 2$). We refer to e.g. the GMP manual[4] or [19, 15] for more accurate estimates.

With this in mind we compute formula (1) with one modular multiplication as follows:

Algorithm 1 RECONSTRUCT

```
Input: $U \equiv R \mod M$ and $V \equiv R \mod N$.
Output: $R_{MN} \equiv R \mod M \times N$.
1: $U_N \equiv V - U \mod N$;
2: $M_N \equiv M^{-1} \mod N$;
3: $U_N \equiv U_N \times M_N \mod N$;
4: $R_{MN} = U + U_N \times M$;
5: if $R_{MN} > M \times N$ then $R_{MN} = R_{MN} - M \times N$ end if
```

Now, if the formula (1) is computed via algorithm 1 and the operation count uses column "Mul." for multiplication and "Div./Gcd." for division/inverse/modulo/gcd, then we have the complexities given in column "CRT" of table 1.

2.2 Radix ladder

Fast algorithms for Chinese remaindering rely on reconstructing pairs of residues of the same size. A usual way of implementing this is via a binary tree structure (see e.g. figure 1 left). But Chinese remaindering is usually an iterative procedure and residues are added one after the other. Therefore it is possible to start combining them two by two before the end of the iterations. Furthermore, when a combination has been made it contains all the information of its leaves. Thus it is sufficient to store only the partially recombined parts and cut its descending branches. We propose to use a *radix ladder* for that task.

[2]http://kaapi.gforge.inria.fr
[3]http://openmp.org
[4]http://gmplib.org/gmp-man-5.0.1.pdf

Size of operands	Mul.	Div. Gcd.	CRT
$\ell \times 1$	ℓ	3ℓ	$9\ell + O(1)$
$\ell \times \ell$	$m_\alpha \ell^\alpha$	$d_\alpha \ell^\alpha$	$2(m_\alpha + d_\alpha)\ell^\alpha + O(\ell)$

Table 1: Integer arithmetic complexity model

DEFINITION 1. *A* radix ladder *is a ladder composed of successive shelves. A* shelf *is either empty or contains a modulus and an associated residue, denoted respectively M_i and U_i at level i. Moreover, at level i, are stored only residues or moduli of* size 2^i.

New pairs of residues and moduli can be inserted anywhere in the ladder. If the shelf corresponding to its size is empty, then the pair is just stored there, otherwise it is combined with occupant of the shelf, the latter is dismissed and the new combination tries to go one level up as shown on algorithm 2.

Algorithm 2 RADIXLADDER.**insert**(U, M)

```
Input: $U \equiv R \mod M$ and a Radix ladder
Output: Insertion of $U$ and $M$ in the ladder.,
1: for $i = size(M)$ while Shelf[$i$] is not empty do
2:     $U, M$ :=RECONSTRUCT($U \mod M, U_i \mod M_i$);
3:     Pop Shelf[$i$];
4:     Increment $i$;
5: end for
6: Push $U, M$ in Shelf[$i$];
```

Then if the new level is empty the combination is stored there, otherwise it is combined and goes up ... An example of this procedure is given on figure 1.

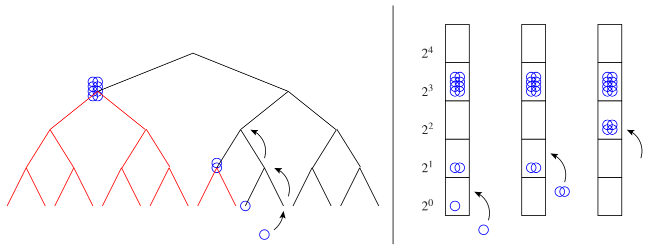


Figure 1: A residue going up the radix ladder

Then to recover the whole reconstructed number it is sufficient to iterate through the ladder from the ground level and make all the encountered partial results go to up one level after the other to the top of the ladder. As we will see in section 3.3, LINBOX-1.1.7 contains such a data structure, in `linbox/algorithms/cra-full-multip.h`.

An advantage of this structure is that it enables insertion of any size pair with fast arithmetic complexity: any reconstruction occurs with two pairs of the same size. Now recall that the sizes of the shelves are powers of 2 to get that the overall complexity is given by:

$$\sum_{i=1}^{\log_2(\ell)-1} O\left((2^i)^\alpha\right) = O(\ell^\alpha)$$

Moreover, the merge of two ladders, shown on algorithm 3, is straightforward and we will be used in a parallel setting in section 5 and algorithm 12.

Algorithm 3 RadixLadder.**merge**

```
Input: Two radix ladders $RL_1$ and $RL_2$.
Output: In place merge of $RL_1$ and $RL_2$.
1: for $i = 0$ to size($RL_2$) do
2:     $RL_1$.insert($RL_2$.Shelf[$i$]);
3: end for
4: Return $RL_1$
```

3. A CHINESE REMAINDERING DESIGN PATTERN

The generic design we propose here comes from the observation that there are in general two ways of computing a reconstruction: a deterministic way computing all the residues until the product of moduli reaches a bound on the size of the result; or a probabilistic way using early termination. We thus propose an abstraction of the reconstruction process in three layers: a black box function produces residues modulo small moduli, an integer builder produces reconstructions using algorithm 2, and a Chinese remaindering controller commands them both.

Here our point is that the controller is completely generic where the builder may use e.g. the radix ladder data structure proposed in section 2 and has to implement the termination strategy.

3.1 Black box residue computation

In general this consists in mapping the problem from $\mathbb{Z}$ to $\mathbb{Z}/m\mathbb{Z}$ and computing the result modulo m. Such black boxes, mapping from integer data structures to finite fields and computing the residue results are defined in the `linbox/solutions` directory of LinBox-1.1.7. There, determinant, valence, minpoly, charpoly or linear system solve are function objects `IntegerModular*` (where `*` is one of the latter functions). We describe an automated alternative for these mappings in section 3.4.

3.2 Chinese remaindering controller

The pattern we propose here is generic with respect to the termination strategy, the integer reconstruction scheme and the residue computation. It somewhat extends the scheme proposed in [21, §4.1] which was independent only of the integer reconstruction scheme.

Our controller must be able to initialize the data structure via the builder; generate some coprime moduli; apply the black box function; update the data structure; test for termination and output the reconstructed element. The generations of moduli and the black box are parameters and the other functionalities are provided by any builder. Then the control is a simple loop. Algorithm 4 shows this loop which contains also the whole interface of the Builder.

Algorithm 4 CRA-Control

```
1: $Builder$.initialize();
2: while $Builder$.notTerminated() do
3:     $p := Builder$.nextCoPrime();
4:     $v := BlackBox$.apply($p$);
5:     $Builder$.update($v, p$);
6: end while
7: Return $Builder$.reconstruct();
```

LinBox gives an implementation of such a controller, parameterized by a builder and a black box function as the class `ChineseRemainder` in `linbox/algorithms/cra-domain.h`.

The interface of a controller is to be a function class: it contains a constructor with a builder as argument and a functional operator taking as argument a BlackBox, computing e.g. a determinant modulo m, and a moduli generator. This functional operator returns an integer reconstructed from the modular computations. Algorithm 5 shows the specifications of the LinBox-1.1.7 controller.

Algorithm 5 C++ **ChineseRemainder** class

```
template<class Builder> struct ChineseRemainder
{
 ChineseRemainder(const Builder& b):bldr(b) {}

 template<class Function> Integer& operator() (
                 Integer          & res,
          const  Function         & BlackBox) {

            // CRA-Control ...

      return res;
 }

protected: Builder bldr;
};
```

Any higher-level algorithm then just has to choose its builder and its controller and pass them the modular BlackBox iteration it wants to lift over the integers.

Definition 2. *The* Chinese remaindering controller *is the algorithm combining the launch of BlackBox iterations with the reconstruction scheme.*

3.3 Integer builders

The role of the builder is to implement the interface defined by algorithm 4:

- `void` *Builder*.**initialize**(): sets e.g. precomputed, mixed radix, representations ...
- `bool` *Builder*.**notTerminated**(): tests a determinist bound or early termination ...
- `Integer` *Builder*.**nextCoPrime**(): produces the next modulus.
- `void` *Builder*.**update**(*residue*, *Integer*): gives the new residue(s) to the builder.
- `Integer` *Builder*.**reconstruct**(): returns the lifted result.

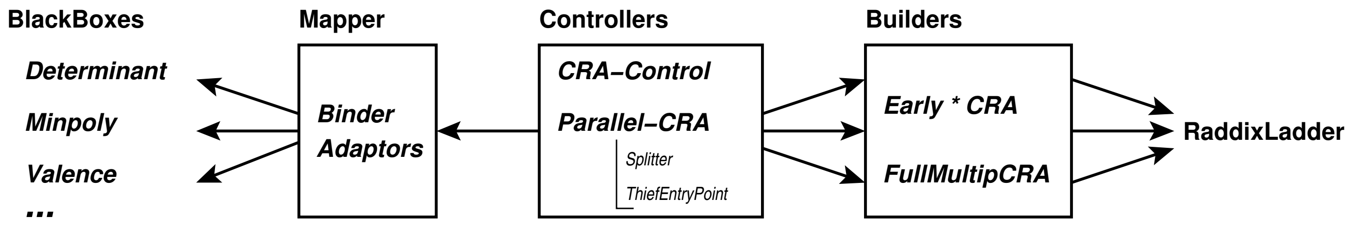


Figure 2: Generic Chinese remaindering scheme

Definition 3. *The* Builder *defines the termination strategy by implementing this interface.*

There are three of these implementations in LinBox-1.1.7: an early terminated version for a single residue, an early terminated version for a vector of residues and a deterministic version for a vector of residues (respectively, the files `cra-early-single.h`, `cra-early-multip.h` and `cra-full-multip.h` in the `linbox/algorithms` directory). Up till now the radix ladder is not a separate class. Indeed we use only this data structure for the underlying computations and it is simple enough to inherit from one of the latter builder implementations and modify the behavior of the methods.

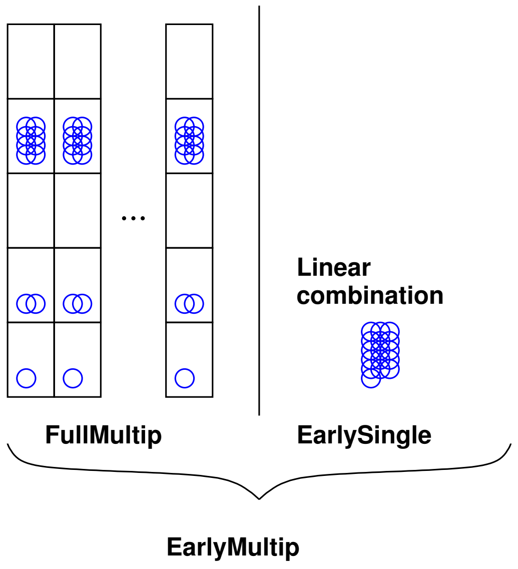


Figure 3: Early termination of a vector of residues via a linear combination

Actually, in the current implementation, `EarlyMultipCRA` inherits from both `EarlySingleCRA` and `FullMultipCRA` as it uses the radix ladder of `FullMultipCRA` for its reconstruction and the early termination of `EarlySingleCRA` to test a linear combination of the residues to be reconstructed as shown on figure 3. The `FullMultipCRA` has been implemented so that when a vector/matrix is reconstructed the moduli and some computations are shared among the ladders.

We give more implementation details on the early termination strategies in sections 4 and 5.

3.4 Mappers and binders

To further enhance genericity, the mapping between integer and field operations can also be automated. If binder adaptors are enclosed within the data structure storing the data, generic mappers can be designed. This is the case for the sparse and dense matrices of LinBox. A generic converter, using the Givaro/LinBox Field's methods `init` and `convert`, can be found in `linbox/field/hom.h` and `linbox/algorithm/matrix-hom.h`.

Then, to map any function class to the field representation one can use the generic mapper, shown as algorithm 6, which defines the *BlackBox*.**apply** method.

Algorithm 6 C++ **Mapper** class

```
template<class Data, class Function>
struct Mapper {
 Mapper(const Data &a, const Function& g)
          : _A(a), _g(g) {}

 template<class Field>
 typename Field::Element& apply (
     typename Field::Element& d,
     const Field& F) const {

  typename Data::template
                 rebind<Field>::other Ap;

  Homomorphism::map(Ap, this->_A, F);

  return this->_g( d, Ap );
 }

protected: const Data& _A; const Function& _g;
};
```

An example of the design usage, here computing a determinant via Chinese remaindering, is then simply algorithm 7.

Definition 4. *The* Mapper *automatically maps and rebinds the Integer data structure to a modular data structure and then calls the BlackBox function on the mapped objects.*

The whole design and the interactions between the presented classes are given on figure 2.

4. TERMINATION STRATEGIES

We sketch here several termination strategies and show that our design enables the modification of this strategy, and only that, while the rest of the implementation is unchanged.

4.1 Deterministic strategy

By deterministic strategy, we denote a strategy where termination is decided when the product of primes so far exceeds a bound on the size of the reconstructed values.

In this case, the implementation is usually straightforward: *FullMultipCRA*.**update**(v, p) just adds the residues

Algorithm 7 C++ Chinese remaindering scheme

```
// [...] matrix initializations etc.

// Defines the termination strategy: here early terminated, for a single integer,
//      using Modular<double> as finite fields
EarlySingleCRA< Modular<double> > ETcra_Builder;

// Defines the Chinese remaindering algorithm
ChineseRemainder< EarlySingleCRA< Modular<double> > > ETcra_Control(ETcra_Builder);

// Defines an apply which given a SparseMatrix<Integer>:
//      maps it generically to a SparseMatrix<FiniteField>
//      and calls the Determinant algorithm on the new matrix
Mapper< SparseMatrix<Integer>, Determinant> BlackBox(A, Det);

// Calls to the controller which launches the BlackBox applications and the Builder reconstructions
Integer d; ETcra_Control(d, BlackBox);
```

to the ladder; where $FullMultipCRA$.**notTerminated**() tests if the product of primes exceeds the bound.

This bound can be precomputed a priori, or refined with properties discovered during the computation.

4.2 Earliest termination

In a sequential mode, depending on the actual speed of the different routines of table 1 on a specific architecture or if the cost of $BlackBox$.**apply** is largely dominant, one can choose to test for termination after each call to the black box. A way to implement the probabilistic test of [12, Lemma 3.1] and to reuse every black box apply is to use random primes as the moduli generator. Indeed then the probabilistic check can be made with the incoming black box residue computed modulo a random prime. The reconstruction algorithm of section 3 is then only slightly modified as shown in algorithm 8.

Algorithm 8 EarlySingleCRA.**update**(v, p)

```
Global: $U \equiv R \mod M$.
Global: A variable $Stabilization$ initially set to 0.
Input: $v \equiv R \mod p$.
Output: $R_{MN} \equiv R \mod M \times p$.
1: $u \equiv U \mod p$;
2: if $u == v$ then
3:     Increment $Stabilization$;
4:     Return $(U, M \times p)$;
5: else
6:     $Stabilization = 0$;
7:     Return Reconstruct$(U \mod M, v \mod p)$;
8: end if
```

Now, $EarlyTerminationThreshold$ is the number of successive stabilizations required to get a probabilistic estimate of failures. It will be denoted ET for the rest of the paper. Then, the termination test becomes simply algorithm 9: test wether the actual number of stabilizations exceeds this threshold.

Algorithm 9 EarlySingleCRA.**notTerminated**()

```
1: Return $Stabilization < EarlyTerminationThreshold$;
```

This is the strategy implemented in LinBox-1.1.7 in `linbox/algorithms/cra-early-single.h`.

For this strategy, the cost of one iteration of algorithm 4 is one BlackBox application and one $i \times 1$ reconstruction. With the estimates of table 1, the cost of the whole reconstruction thus becomes

$$\sum_{i=1}^{t+ET} (\mathbf{apply} + 9i + O(1)) =$$

$$(t + ET)\mathbf{apply} + \frac{9}{2}(t + ET)^2 + O(t) \quad (2)$$

where $t = \lceil \log_{2^\beta}(R) \rceil$ is the size of the result, β is the word size, and **apply** the cost of the BlackBox application.

This strategy enables the least possible number of calls to $BlackBox$.**apply**. It it thus useful when the latter dominates the cost of the reconstruction.

4.3 Balanced termination

Another classic case is when one wants to use fast integer arithmetic for the reconstruction. Then the balanced computations are mandatory and the radix ladder becomes handy.

The problem now becomes the early termination. There a simple strategy could be to test for termination only when the number of computed residues is a power of two. In that case the reconstruction is guaranteed to be balanced and fast Chinese remaindering is also guaranteed.

Moreover random moduli are not any more necessary for all the residues, only those testing for early termination need be randomly generated. This induces another saving if one fixes the other primes and precomputes all the factors $M_i \times (M_i^{-1} \mod M_{i+1})$. There the cost of the reconstruction drops by a factor of 2 from $2(m_\alpha + d_\alpha)l^\alpha$ to $(m_\alpha + d_\alpha)l^\alpha$. The drawback is an extension of the number of black box applications from $\lceil \log_{2^\beta}(R) \rceil + ET$ to the largest power of two immediately superior and thus up to a factor of 2 in the number of black box applies.

For the $Builder$, the update becomes just a push in the ladder as shown on algorithm 10.

Algorithm 10 EarlyBalancedCRA.**update**(v, p)

```
1: RadixLadder.insert$(v, p)$;
```

The termination condition, on the contrary tests only when the number of residues is power of two as shown on algorithm 11.

Algorithm 11 EarlyBalancedCRA.**notTerminated**()

```
1: if Only one Shelf, Shelf[$i$], is full then
2:    Set $U_i$ to Shelf[$i$] residue;
3:    for $j = 1$ to $EarlyTerminationThreshold$ do
4:       $p$ :=PrimeGenerator();
5:       if ($U_i \mod p$) $!=$ $BlackBox$.apply($p$) then
6:          Return $false$;
7:       end if
8:    end for
9:    Return $true$;
10: else
11:    Return $false$;
12: end if
```

Then, the whole reconstruction of algorithm 4 now requires:

$$ET \cdot (\mathbf{apply} + 3 \cdot 2^k) + \sum_{i=0}^{k-1} \frac{2^k}{2^{i+1}} \left(\mathbf{apply} + (m_\alpha + d_\alpha)2^{i\alpha}\right) + (\mathbf{apply} + 3 \cdot 2^i) = (2^k + k + ET - 1) \cdot \mathbf{apply} + \left(2^k\right)^\alpha \frac{m_\alpha + d_\alpha}{2^\alpha - 2} + O(2^k) \quad (3)$$

operations, where now $k = \lceil \log_2(\log_{2^\beta}(R)) \rceil$.

Despite the increase in the number of black box applications, the latter can be useful, in particular when multiple values are to be reconstructed.

Example 1. *Consider the Gaußian elimination of an integer matrix where all the matrix entries are bounded in absolute value by $A_\infty > n$. Let $a_\infty = \log_{2^\beta}(A_\infty)$ and suppose one would like to compute the rational coefficients of the triangular decomposition only by Chinese remaindering (there exist better output dependant algorithms, see e.g. [24], but usually with the same worst-case complexity). Now, Hadamard bound gives that the resulting numerators and denominators of the coefficients are bounded by $\sqrt{nA_\infty^2}^n$. Then the complexity of the earliest strategy would be dominated by the reconstruction where the balanced strategy or the hybrid strategy of figure 3 could benefit from fast algorithms. Indeed the naive earliest strategy would use n^2 reconstructions, each one of cost $O(t^2) = O(\log^2(\sqrt{nA_\infty^2}^n)) = O(n^2a_\infty^2)$, by eq. (2); overall this would dominate even the elimination cost. Then the hybrid strategy would use the earliest strategy only on a linear combination, of single cost $O(\log^2(n\sqrt{nA_\infty^2}^n)) = O(n^2a_\infty^2)$, and the n^2 reconstructions would be computed with fast integer arithmetic. Finally the balanced strategy removes the dependency in a_∞^2 by performing only fast integer arithmetic. Table 2 compares all three complexities.*

`EarlySingleCRA`	$O(n^4a_\infty^2)$
`EarlyMultipCRA`	$O(n^{\omega+1}a_\infty + n^{2+\alpha}a_\infty^\alpha + n^2a_\infty^2)$
`EarlyBalancedCRA`	$O(2n^{\omega+1}a_\infty + 2n^{2+\alpha}a_\infty^\alpha)$

Table 2: Early termination strategies complexities for Chinese remaindered Gaußian elimination with rationals

In the case of small matrices with large entries the reconstruction dominates and then a balanced strategy is preferable. Now if both complexities are comparable it might be useful to reduce the factor of 2 overhead in the black box applications. This can be done via amortized techniques, as shown next.

4.4 Amortized termination

A possibility is to use the ρ-amortized control of [2]: instead of testing for termination at steps 2^1, 2^2, ..., 2^i, ... the tests are performed at steps $\rho^{g(1)}$, $\rho^{g(2)}$, ..., $\rho^{g(i)}$, ... with $1 < \rho < 2$ and g satisfies $\forall i$, $g(i) \leq i$. If the complexity of the modular problem is C and the number of iterations to get the output is b, [2] give choices for ρ and g which enable to get the result with only $b + \frac{f(b)}{b}$ iterations and extra $O(f(b))$ termination tests where $f(b) = log_\rho(b)$.

In example 1 the complexity of the modular problem is n^ω, the size of the output and the number of iterations is na_∞ so that strategy would reduce the iteration complexity from $2n^{\omega+1}a_\infty$ to $(na_\infty + o(na_\infty))n^\omega$ and the overall complexity would then become:

`EarlyAmortizedCRA`	$O(n^{\omega+1}a_\infty + n^{2+\alpha}a_\infty^\alpha + \log(na_\infty)n^\alpha a_\infty^\alpha)$

Indeed, we suppose that the amortized technique is used only on a linear combination, and that the whole matrix is reconstructed with a `FullMultipCRA`, as in figure 3. Then the linear combination has size $2\log(n) + n \cdot a_\infty$ which is still $O(n \cdot a_\infty)$. Nonetheless, there is an overhead of a factor $\log(na_\infty)$ in the linear combination reconstruction since there might be up to $O(\log(na_\infty))$ values $\rho^{g(i)}$, $\rho^{g(i+1)}$, ... between any two powers of two. Overall this gives the above estimate. Now one could use other g functions as long as eq. 4 is satisfied.

$$\begin{cases} \left(\rho^{g(i+1)} - \rho^{g(i)}\right) = o(\rho^{g(i)}) \\ \left(\rho^{g(i+k(i))} - \rho^{g(i)}\right) \sim 2^{\lceil \log_2(\rho^{g(i)}) \rceil}, \quad k(i) = o(\rho^{g(i)}) \end{cases} \quad (4)$$

5. PARALLELIZATION

All parallel versions of these sequential algorithms have to consider the parallel merge of radix ladders and the parallelization of the loop of the CRA-control algorithm 4. Many parallel libraries can be used, namely OpenMP or Cilk would be good candidates for the parallelization of the embarrassingly parallel `FullMultipCRA`. Now in the early termination setting, the main difficulty comes from the distribution of the termination test. Indeed, the latter depends on data computed during the iterations. To handle this issue we propose an adaptive parallel algorithm [5, 26] and use the Kaapi library [6, 16]. Its expressiveness in an adaptive setting guided our choice, together with the possibility to work on heterogenous networks.

5.1 Kaapi overview

Kaapi is a task based model for parallel computing. It was targeted for distributed and shared memory computers. The scheduling algorithm uses work-stealing [3, 1, 4, 17]: an idle processor tries to steal work to a randomly selected victim processor.

The sequential execution of a Kaapi program consists in pushing and popping tasks to dequeue the current running processor. Tasks should declare the way they access the memory, in order to compute, at runtime, the data flow dependencies and the ready tasks (when all their input values are produced). During a parallel execution, a ready task, in the queue but not executed, may be entirely theft and executed on an other processor (possibly after being communicated through the network). These tasks are called *dfg tasks* and their schedule by work-stealing is described in [16, 17].

A task being executed by a processor may be *only partially* stolen if it interacts with the scheduler, in order to e.g. decide which part of the work is to be given to the thieves. Such tasks are called *adaptive tasks* and allows fine grain loop parallelism.

To program an adaptive algorithm with Kaapi, the programmer has to specify some points in the code (using `kaapi_stealpoint`) or sections of the code (`kaapi_stealbegin` and `kaapi_stealend`) where thieves may steal work. To guarantee that the parallel computation is completed, the programmer has to wait for the finalization of the parallel execution (using `kaapi_steal_finalize`). Moreover, in order to better balance the work load, the programmer may also decide to preempt the thieves (send an event via `kaapi_preempt_next`).

5.2 Parallel earliest termination

Algorithm 12 lets thieves steal any sequence of primes.

Algorithm 12 ParallelCRA-Control

```
 1: Builder.initialize();
 2: while Builder.notTerminated() do
 3:    p := Builder.nextCoPrime();
 4:    kaapi_stealbegin( splitter, Builder);
 5:    v := BlackBox.apply(p);
 6:    Builder.update(v, p);
 7:    kaapi_finalize_steal();
 8:    kaapi_stealend();
 9:    if require synchronization step then
10:       while kaapi_nomore_thief() do
11:          (list of v, list of p) :=kaapi_preempt_next();
12:          Builder.update(list of v, list of p);
13:       end while
14:    end if
15: end while
16: Return Builder.reconstruct();
```

At line 12, the code allows the scheduler to trigger the processing of steal requests by calling the *splitter* function. The parameters of `kaapi_stealbegin` are the *splitter* function and some arguments to be given to its call. These arguments[5] can e.g. specify the state of the computation to modify (here the builder object plays this role).

Then, on the one hand, concurrent modifications of the state, of computation by thieves, must be taken care of during the control flow between lines 12 and 12: here the computation of the residue could be evaluated by multiple threads without any critical section[6]. On the other hand, after line 12, the scheduler guarantees that no concurrent thief can modify the computational state when they steal some work. We remark that both branches of the conditional `if` at line 12 must be executed without concurrency: the iteration of the list of thieves or the generation of the next random modulus are not reentrant.

[5]in or out

[6]This depends on the implementation, most of the LinBox library functions are reentrant

The role of the *splitter* function is to distribute the work among the thieves. In algorithm 13, each thief receives a `coPrimeGenerator` object and the *entrypoint* to execute.

Algorithm 13 Splitter($Builder, N, requests[]$)

```
1: for i = 0 to N − 1 do
2:    kaapi_request_reply(request[i], entrypoint,
            Builder.getCoPrimeGenerator() );
3: end for
```

The `coPrimeGenerator` depends on the `Builder` type and allows the thief to generate a sequence of moduli. For instance the `coPrimeGenerator` for the earliest termination contains at one point a single modulus M which is returned by the next call of `nextCoPrime()` by the `Builder`.

The *splitter* function knows the number N of thieves that are trying to steal work to the same victim. Therefore it allows for a better balance of the work load. This feature is unique to Kaapi when compared to other tools having a work-stealing scheduler.

5.3 Synchronization

Now, the victim periodically tests the global termination of the computation (line 12 of algorithm 12). Depending on the chosen termination method (`Early*CRA`, etc.), the synchronization may occur at every iteration or after a certain number of iterations. The choice is made in order to e.g. amortize the cost of this synchronization or reduce the arithmetic cost of the reconstruction.

Then each thief is preempted (line 12) and the code recovers its results before giving them to the `Builder` for future reconstruction (line 12).

The preemption operation is a two way communication between a victim and a thief: the victim may pass parameters *and* get data from one thief. Note that the preemption operation assumes cooperation with the thief code. The latter being responsible for polling incoming events at specific points (e.g. where the computational state is safe preemption-wise).

On the one hand, to amortize the cost of this synchronization, more primes should be given to the thieves. In the same way, the victim code works on a list of moduli inside the critical section (at line 12 returns a list of moduli, and at lines 12-12 the victim iterates over this list by repeatedly calling `apply` and `update` methods). On the other hand, to avoid long waits of the victim during preemption, each thief should test if it has been preempted to return quickly its results (see next section).

5.4 Thief entrypoint

Finally, algorithm 14 returns both the sequence of residues and the sequence of primes that where given to the BlackBox. This algorithm is very similar to algorithm 12.

Lines 14 and 14 define a section of code that could be concurrent with steal requests. At line 14, the code tests if a preemption request has been posted by algorithm 12 at line 12. If this is the case, then the thief aborts any further computation and the result is only a partial set of the initial work allocated by the *splitter* function.

5.5 Efficiency

These parallel versions of the Chinese remaindering have been implemented using Kaapi transparently from the LinBox library: one has just to change the sequential controller

Matrix	$\mathbf{d}, \mathbf{r}$	$\mathbf{T_{seq}}$	$[k]$	$\mathbf{T_{p=8}}$	$[k]$	$\mathbf{T_{p=16}}$	$[k]$	speed-up	Naive
$EX1$	$560 \times 560,\ 8736$	$0.29s$	[4]	$0.16s$	[9]	$0.22s$	[16.8]	1.32	**1.38**
$EX3$	$2600 \times 2600,\ 71760$	$837.80s$	[184]	$123.56s$	[193]	$77.99s$	[193]	**10.74**	10.66
$T150$	$150 \times 150,\ 2040$	$0.21s$	[59]	$0.046s$	[63.4]	$0.036s$	[63.6]	**5.83**	2.10
$T300$	$300 \times 300,\ 4678$	$2.52s$	[138]	$0.36s$	[144.8]	$0.24s$	[144.7]	**10.50**	8.52
$T500$	$500 \times 500,\ 8478$	$15.19s$	[249]	$2.05s$	[257]	$1.31s$	[256.3]	**11.60**	11.29
$T700$	$700 \times 700,\ 12654$	$52.59s$	[367]	$6.50s$	[368.9]	$4.19s$	[371.2]	12.55	12.55
$T2000$	$2000 \times 2000,\ 41907$	$2978.23s$	[1274]	$384.43s$	[1281]	$236.59s$	[1281]	**12.59**	12.48

Table 3: Timings in seconds for the computation of the determinant. d is the dimension of the matrix, r the number of non-zero coefficients, $[k]$ is the mean number of primes observed for the Chinese remaindering using p cores. The speed-ups are for 16 cores of algorithm 12 with Kaapi compared to a naive approach with OpenMP.

Algorithm 14 Thief's EntryPoint(M)

```
1: $Builder$.initialize();
2: $list\ of\ v$.clear();
3: $list\ of\ p$.clear();
4: while $Builder$.CoPrimeGenerator() not empty do
5:     if kaapi_preemptpoint() then break; end if
6:     $p := Builder$.nextCoPrime();
7:     kaapi_stealbegin( $splitter$, $Builder$);
8:     $list\ of\ p$.push_back($p$);
9:     $list\ of\ v$.push_back($BlackBox$.apply($p$));
10:    kaapi_stealend();
11: end while
12: kaapi_stealreturn ($list\ of\ v, list\ of\ p$);
```

`cra-domain.h` to the parallel one.

In LinBox-1.1.7 some of the sequential algorithms which make use of some Chinese remaindering are the determinant, the minimal/characteristic polynomial and the valence, see e.g. [20, 12, 11, 8] for more details.

We have performed these preliminary experiments on an 8 dual core machine (Opteron 875, 1MB L2 cache, 2.2Ghz, with 30GBytes of main memory). Each processor is attached to a memory bank and communicates to its neighbors via an hypertransport network. We used g++ 4.3.4 as C++ compiler and the Linux kernel was the 2.6.32 Debian distribution.

All timings are in seconds. In the following, we denote by T_{seq} the time of the sequential execution and by T_p the time of the parallel execution for $p = 8$ or $p = 16$ cores. All the matrices are from "Sparse Integer Matrix Collection" (SIMC)[7].

Table 3 gives the performance of the parallel computation of the determinant for small invertible matrices (less than a second) and larger ones (an hour CPU) of the `SIMC/SPG` and `SIMC/Trefethen` collections.

The small instance (EX1) needed very few primes to reconstruct the integer solution. There, we can see the overhead of parallelism: this is due to some extra synchronizations and also to the large number of unnecessary modular computations before realizing that early termination was needed. Despite this we do achieve some speed-up and compare them with a naive approach using OpenMP: for p the number available cores, launch the computations by blocks of p iterations and test for terminaison after each block is completed.

For large computations the speed-up is quite the same since the computation is largely dominant, and the better for OpenMP when the actual number of modular computation is a multiple of the number of processors. For smaller instances we see the advantage of reducing the number of synchronizations. On e.g. multi-user environments the advantage should be even greater.

6. CONCLUSION

We have proposed a new data structure, the radix ladder, capable of managing several kinds of Chinese reconstructions while still enabling fast reconstruction.

Then, we have defined a new generic design for Chinese remaindering schemes. It is summarized on figure 2. Its main feature is the definition of a builder interface in charge of the reconstruction. This interface is such that any termination (deterministic, early terminated, distributed, etc.) can be handled by a CRA controller. It enables to define and test remaindering strategies while being transparent to the higher level routines. Indeed we show that the Chinese remaindering can just be a plug-in in any integer computation.

We also provide in LinBox-1.1.7 an implementation of the ladder, several implementations for different builders and a sequential controller. Then we tested the introduction of a parallel controller, written with Kaapi, without any modification of the LinBox library. The latter handles the difficult issue of distributed early termination and shows good performance on a SMP machine.

In parallel, some improvement could be made to the early termination strategy in particular when the BlackBox is fast compared to the reconstruction and when balanced and amortized techniques are required. Also, output sensitive early termination is very useful for rational reconstruction, see e.g. [22] and thus the latter should benefit from this kind of design.

7. REFERENCES

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01), Puerto Vallarta*, pages 119–129, 2001.

[2] O. Beaumont, E. M. Daoudi, N. Maillard, P. Manneback, and J.-L. Roch. Tradeoff to minimize

[7]`http://ljk.imag.fr/CASYS/SIMC`

extra-computations and stopping criterion tests for parallel iterative schemes. In *3rd International Workshop on Parallel Matrix Algorithms and Applications (PMAA04)*, CIRM, Marseille, France, Oct. 2004.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[4] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In P. B. Gibbons and P. G. Spirakis, editors, *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms (SPAA'05), Las Vegas, Nevada, USA*, pages 21–28. ACM, July 2005.

[5] V. D. C. Cung, V. Danjean, J.-G. Dumas, T. Gautier, G. Huard, B. Raffin, C. Rapine, J.-L. Roch, and D. Trystram. Adaptive and hybrid algorithms: classification and illustration on triangular system solving. In Dumas [7], pages 131–148.

[6] V. Danjean, R. Gillard, S. Guelton, J.-L. Roch, and T. Roche. Adaptive loops with kaapi on multicore and grid: Applications in symmetric cryptography. In Moreno-Maza and Watt [23], pages 33–42.

[7] J.-G. Dumas, editor. *TC'2006. Proceedings of Transgressive Computing 2006, Granada, España.* Universidad de Granada, Spain, Apr. 2006.

[8] J.-G. Dumas. Bounds on the coefficients of the characteristic and minimal polynomials. *Journal of Inequalities in Pure and Applied Mathematics*, 8(2):art. 31, 6 pp, Apr. 2007.

[9] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: A generic library for exact linear algebra. In A. M. Cohen, X.-S. Gao, and N. Takayama, editors, *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, pages 40–50. World Scientific Pub., Aug. 2002.

[10] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over prime fields. *ACM Transactions on Mathematical Software*, 35(3):1–42, Nov. 2008.

[11] J.-G. Dumas, C. Pernet, and Z. Wan. Efficient computation of the characteristic polynomial. In M. Kauers, editor, *Proceedings of the 2005 ACM International Symposium on Symbolic and Algebraic Computation, Beijing, China*, pages 140–147. ACM Press, New York, July 2005.

[12] J.-G. Dumas, B. D. Saunders, and G. Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computation*, 32(1/2):71–99, July–Aug. 2001.

[13] J.-G. Dumas and A. Urbańska. An introspective algorithm for the determinant. In Dumas [7], pages 185–202.

[14] I. Z. Emiris. A complete implementation for computing general dimensional convex hulls. *International Journal of Computational Geometry and Applications*, 8(2):223–253, Apr. 1998.

[15] J. v. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, New York, NY, USA, 1999.

[16] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In Moreno-Maza and Watt [23], pages 15–23.

[17] T. Gautier, J. L. Roch, and F. Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In *Workshop PAPP 2007 - Practical Aspects of High-Level Parallel Programming in International Conference on Computational Science 2007 (ICCS2007)*, Beijing, China, may 2007. IEEE.

[18] A. Goldstein and R. Graham. A Hadamard-type bound on the coefficients of a determinant of polynomials. *SIAM Review*, 15:657–658, 1973.

[19] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, Orlando, Florida, June 20–24, 1994.

[20] E. Kaltofen. An output-sensitive variant of the baby steps/giant steps determinant algorithm. In T. Mora, editor, *Proceedings of the 2002 ACM International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 138–144. ACM Press, New York, July 2002.

[21] E. Kaltofen and M. Monagan. On the genericity of the modular polynomial GCD algorithm. In S. Dooley, editor, *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, Vancouver, Canada*, pages 59–66. ACM Press, New York, July 1999.

[22] S. Khodadad and M. Monagan. Fast rational function reconstruction. In J.-G. Dumas, editor, *Proceedings of the 2006 ACM International Symposium on Symbolic and Algebraic Computation, Genova, Italy*, pages 184–190. ACM Press, New York, July 2006.

[23] M. Moreno-Maza and S. Watt, editors. *Parallel Symbolic Computation'07.* Waterloo University, Ontario, Canada, July 2007.

[24] C. Pernet and W. Stein. Fast computation of hermite normal form of random integer matrices. Technical report, 2009. `http://modular.math.washington.edu/papers/hnf/hnf.pdf`.

[25] C. Pernet and A. Storjohann. Faster algorithms for the characteristic polynomial. In C. W. Brown, editor, *Proceedings of the 2007 ACM International Symposium on Symbolic and Algebraic Computation, Waterloo, Canada.* ACM Press, New York, July 29 – August 1 2007.

[26] D. Traore, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Adaptive parallel algorithms and applications to STL. In Springer-Verlag, editor, *EUROPAR 2008*, Las Palmas, Spain, August 2008.

A complete modular resultant algorithm targeted for realization on graphics hardware

Pavel Emeliyanenko
Max-Planck Institute for Informatics
Saarbrücken, Germany
asm@mpi-sb.mpg.de

ABSTRACT

This paper presents a complete modular approach to computing bivariate polynomial resultants on Graphics Processing Units (GPU). Given two polynomials, the algorithm first maps them to a prime field for sufficiently many primes, and then processes each modular image individually. We evaluate each polynomial at several points and compute a set of univariate resultants for each prime in parallel on the GPU. The remaining "combine" stage of the algorithm comprising polynomial interpolation and Chinese remaindering is also executed on the graphics processor. The GPU algorithm returns coefficients of the resultant as a set of Mixed Radix (MR) digits. Finally, the large integer coefficients are recovered from the MR representation on the host machine. With the approach of displacement structure [16] and efficient modular arithmetic [8] we have been able to achieve more than 100x speed-up over a CPU-based resultant algorithm from Maple 13.[1]

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Algebraic algorithms*

General Terms

Algorithms, Experimentation, Measurement, Performance, Theory

Keywords

CUDA, GPU, modular algorithm, parallel computing, polynomial resultants

1. INTRODUCTION

Resultants is a powerful algebraic tool in the quantifier elimination theory. Among their numerous and widespread applications, resultants play an important role in topological study of algebraic curves and computer graphics. However, despite the fact that this problem received a good deal of attention in the literature, resultants still constitute a major bottleneck for many geometric algorithms. This is mainly due to the rapid growth of coefficient bit-length and the degree of the resultant polynomial with respect to the initial parameters as well as the necessity to work in a complicated domain. We find it, therefore, very advantageous to try to utilize the incredible computational horsepower of Graphics Processing Units (GPUs) for this problem.

A classical resultant algorithm is the one of Collins [6]. The algorithm employs modular and evaluation homomorphisms to deal with expression swell during computation of resultants. Following the "divide-conquer-combine" strategy, it reduces the coefficients of input polynomials modulo sufficiently many primes. Then, several evaluation homomorphisms are applied recursively reducing the problem to the univariate case. Finally, a set of univariate resultants are computed using polynomial remainder sequences (PRS), see [11]. The final result is recovered by means of polynomial interpolation and the Chinese remainder algorithm (CRA).

This idea gave rise to the whole spectrum of modular algorithms: including sequential [20, 17], and parallel ones specialized for workstation networks [4] or shared memory machines [21, 15]. However, neither of these algorithms admits a straightforward realization on the GPU. The reason for that is because the modular approach exhibits only a *coarse-grained* parallelism since the PRS algorithm, lying in its core, hardly admits any parallelization. This is a good choice for traditional parallel platforms such as workstation networks or multi-core machines but not for massively-threaded architecture like that of GPU. That is why, we have decided to use an alternative method to solve the problem in the univariate case, namely, the approach of *displacement structure* [16]. In the essence, this method reduces computation of the resultant to the triangular factorization of a structured matrix. Operations on matrices generally map very well to the GPU's threading model. The displacement structure approach is traditionally applied in floating-point arithmetic, however using square-root and division-free modifications [10] we have been able to adapt it for a prime field. It is worth mentioning that *even though the PRS algorithm can also be made division-free* (which is probably the method of choice for a modular approach) this does not facilitate its the realization on the GPU.

In this work we extend our previous results [9] by porting the remaining stages of the resultant algorithm (polynomial interpolation and partly the CRA) to the graphics processor, thereby, minimizing the amount of work to be done on

[1] `www.maplesoft.com`

the CPU. For the sake of completeness, we present the full approach here. We also use an improved version of the univariate resultant algorithm given in [9] which is based on a modified displacement equation, see Section 2.2. Additionally, we have developed an efficient stream compaction algorithm based on parallel reductions in order to eliminate "bad" evaluation points right on the GPU, see Section 4.1. What concerns the CRA, we compute the coefficients of the resultant polynomial using Mixed Radix (MR) representation [22] on the GPU without resorting to multi-precision arithmetic, and finally recover the large integers on the host machine. Foundation of our algorithm is a fast 24-bit modular arithmetic developed in [8]. The arithmetic rests on mixing floating-point and integer computations, and widely exploits the GPU multiply-add capabilities.

The organization of the paper is as follows. In Section 2 we formulate the problem in a mathematically concise way and give an introduction to displacement structure and the generalized Schur algorithm. Section 3 describes the GPU architecture and CUDA framework. Section 4 focuses on the algorithm itself including the main aspects of the GPU realization. In Section 5 we present experimental results and draw conclusions.

2. THEORETICAL BACKGROUND

In this section we give a definition of the resultant of two polynomials, and describe the displacement structure approach in application to univariate resultants and polynomial interpolation. We also briefly consider Chinese remaindering and the Mixed Radix representation of the numbers.

2.1 Polynomial resultants

Let f and g be two polynomials in $\mathbb{Z}[x,y]$ of y-degrees p and q respectively: $f(x,y) = \sum_{i=0}^{p} f_i(x)y^i$ and $g(x,y) = \sum_{i=0}^{q} g_i(x)y^i$. Let $R = res_y(f,g)$ denote the resultant of f and g with respect to y. The resultant R is the determinant of $(p+q)\times(p+q)$ Sylvester matrix S:

$$R = det(S) = det\begin{bmatrix} f_p & f_{p-1} & \dots & f_0 & 0\dots & 0 \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \dots 0 & f_p & f_{p-1} & \dots & f_0 \\ g_q & g_{q-1} & \dots & g_0 & 0\dots & 0 \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \dots 0 & g_q & g_{q-1} & \dots & g_0 \end{bmatrix}.$$

From the definition it follows that the resultant R is a polynomial in $\mathbb{Z}[x]$. However, using modular techniques one can reduce the problem to the univariate case. Computing univariate resultants is discussed in the next section.

2.2 Displacement structure of Sylvester matrix and the generalized Schur algorithm

Suppose we are given two polynomials $f,g \in \mathbb{Z}[x]$ of degrees p and q respectively ($p \geq q$) and the associated Sylvester matrix $S \in \mathbb{Z}^{n\times n}$ ($n = p+q$). Let us first assume that S is a strongly regular matrix.[1] The matrix S is structured with a *displacement rank* 2 because it satisfies the *displacement equation*:

$$S - FSA^T = GB^T \text{ where } F = Z_n \text{ and } A = Z_q \oplus Z_p.$$

[1]Meaning that its leading principal minors are non-singular.

Here $Z_n \in \mathbb{Z}^{n\times n}$ is a down-shift matrix zeroed everywhere except for 1's on the first subdiagonal. Accordingly, *generators* $G, B \in \mathbb{Z}^{n\times 2}$ are matrices whose entries can be deduced from the matrix S directly by inspection:

$$G^T = \begin{bmatrix} f_p & f_{p-1} & \dots & f_0 & 0\dots & 0 \\ g_q & q_{q-1} & \dots & g_0 & 0 & \dots & 0 \end{bmatrix} \quad \begin{matrix} B \equiv 0 \text{ except for} \\ B_{0,0} = B_{q,1} = 1 \end{matrix}$$

Consequently, the matrix S can fully be described by its generators. Our goal is to obtain an LDU^T-factorization of S where the matrices L and U are triangular with unit diagonals, and D is a diagonal matrix. Having this factorization, the resultant is: $det(S) = det(D) = \prod_i^n d_{ii}$ (the product of diagonal entries of D).

The generalized Schur algorithm [16, 5] computes the matrix factorization by iteratively computing the *Schur complements* of leading submatrices. The Schur complement R of a submatrix M_{00} in $M = [M_{ij}]$, $(i,j = \{0,1\})$ is defined as: $R = M_{11} - M_{10}M_{00}^{-1}M_{01}$. The idea of the algorithm is to rely on a low-rank displacement representation of a matrix. The displacement equation, preserved under all transformations, allows us to derive the matrix factorization by operating solely on matrix generators. As a result, *the matrix factorization can be computed in $\mathcal{O}(n^2)$ arithmetic operations*, see [16, p. 323].

In each step, the generators are transformed to a *proper* form. Let us denote the generator matrices in step i by (G_i, B_i). A generator $\overline{G}_i$ is said to be in a proper form if it has only *one* non-zero entry in its first row. The transformation is done by applying non-Hermitian rotation matrices Θ_i and Γ_i[2] such that $\overline{G}_i = (G_i\Theta_i)$ and $\overline{B}_i = (B_i\Gamma_i)$:

$$\overline{G}_i^T = \begin{bmatrix} \delta^i & a_1^i & a_2^i & \dots \\ 0 & b_1^i & b_2^i & \dots \end{bmatrix}, \overline{B}_i^T = \begin{bmatrix} \zeta^i & c_1^i & c_2^i & \dots \\ 0 & d_1^i & d_2^i & \dots \end{bmatrix}.$$

Once the generators are in proper form, the displacement equation yields: $d_{ii} = \delta^i\zeta^i$. To obtain the next generator G_{i+1} (and by analogy B_{i+1}) from $\overline{G}_i$ (or $\overline{B}_i$) we multiply its first column (the one with the entry δ^i or ζ^i) by corresponding down-shift matrix (F for $\overline{G}_i$ and A for $\overline{B}_i$) while keeping the other column intact, see [16]. In case of $\overline{G}_i$ this corresponds to shifting down all elements of the first column by one position, while for $\overline{B}_i$, in addition to the down-shift, the q-th entry of the first column (c_q^i) must be zeroed.[3] Accordingly, the length of generators decreases by one in each step of the algorithm.

2.3 Division-free rotations in non-Hermitian case

Note that the term non-Hermitian stands for the fact that we apply rotations to an asymmetric generator pair. To bring the generators to a proper form we need to find matrices Θ and Γ that satisfy: $\begin{bmatrix} a_0 & b_0 \end{bmatrix}\Theta = \begin{bmatrix} \delta & 0 \end{bmatrix}$, $\begin{bmatrix} c_0 & d_0 \end{bmatrix}\Gamma = \begin{bmatrix} \zeta & 0 \end{bmatrix}$, with $\Theta\Gamma^T = I$. This holds for the following matrices:

$$\Theta = \begin{bmatrix} c_0 & b_0 \\ d_0 & -a_0 \end{bmatrix}, \Gamma = \frac{1}{D}\begin{bmatrix} a_0 & d_0 \\ b_0 & -c_0 \end{bmatrix}, D = a_0c_0 + b_0d_0.$$

To get rid of expensive divisions, we use an idea similar to the one introduced for Givens rotations [10]. Namely, we postpone the division until the end of the algorithm by

[2]These matrices must satisfy: $\Theta\Gamma^T = I$, which preserves the displacement equation since: $G\Theta(B\Gamma)^T = GB^T$.

[3]This is because the matrix $A = Z_q \oplus Z_p$ is formed of two down-shift matrices.

keeping a common denominator for each generator column. Put it differently, we express the generators as follows:

$$\mathbf{a}^T = 1/l_a \cdot (a_0,\ a_1,\ \dots) \quad \mathbf{b}^T = 1/l_b \cdot (b_0,\ b_1,\ \dots),$$
$$\mathbf{c}^T = 1/l_c \cdot (c_0,\ c_1,\ \dots) \quad \mathbf{d}^T = 1/l_d \cdot (d_0,\ d_1,\ \dots),$$

where $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$. Accordingly, the generator update $(\overline{G}, \overline{B}) = (G\Theta, B\Gamma)$ proceeds in the following way:

$$\overline{a}_i = l_a(a_i c_0 + b_i d_0) \quad \overline{b}_i = l_b(a_i b_0 - b_i a_0),$$
$$\overline{c}_i = l_c(c_i a_0 + d_i b_0) \quad \overline{d}_i = l_d(c_i d_0 - d_i c_0),$$

here $\overline{G} = (\overline{\mathbf{a}}, \overline{\mathbf{b}})$ and $\overline{B} = (\overline{\mathbf{c}}, \overline{\mathbf{d}})$. Moreover, it follows that the denominators are *pairwise* equal, thus, we can keep only two of them. They are updated as follows: $\overline{l}_a = \overline{l}_d = \overline{a}_0$, $\overline{l}_c = \overline{l}_b = l_a l_c^2$.

Note that the denominators must be non-zero to prevent the algorithm from breaking down. It is guaranteed by a *strong-regularity* assumption introduced at the beginning. Yet, this is not always the case for Sylvester matrix. In Section 4.1 we discuss how to alleviate this problem.

2.4 Polynomial interpolation

The task of polynomial interpolation is to find a polynomial $f(x)$, $deg(f) < n$, satisfying the set of equations: $f(x_i) = y_i$, for $0 \le i < n$. The coefficients a_i of f are given by the solution of the system: $V\mathbf{a} = \mathbf{y}$, where $V \in \mathbb{Z}^{n\times n}$ is a Vandermonde matrix: $V_{ij} = x_i^j$ $(i, j = 0, \dots, n-1)$. If we apply the generalized Schur algorithm to the following matrix $M \in \mathbb{Z}^{2n\times(n+1)}$:

$$M = \begin{bmatrix} V & -\mathbf{y} \\ I_n & \mathbf{0} \end{bmatrix}, \text{ where } I_n \text{ is } n\times n \text{ identity matrix,}$$

then after n steps we obtain the Schur complement R of V, such that: $R = \mathbf{0} - I_n V^{-1}(-\mathbf{y}) = V^{-1}\mathbf{y}$, i.e., the desired solution. The matrix M has a displacement rank 2 and satisfies the equation:

$$M - FMA^T = GB^T,$$

here $F = diag(x_0 \dots x_{n-1}) \oplus Z_n$, $A = Z_{n+1}$, with $Z_n \in \mathbb{Z}^{n\times n}$ is a down-shift matrix. The generators $G \in \mathbb{Z}^{2n\times 2}$ and $B \in \mathbb{Z}^{(n+1)\times 2}$ have the following form:

$$G^T = \begin{bmatrix} 1 & \dots & 1 & 1 & 0 & \dots & 0 \\ y_0 & \dots & y_{n-1} & 0 & 0 & \dots & 0 \end{bmatrix} \quad \begin{matrix} B \equiv 0 \text{ except for} \\ B_{0,0} = 1,\ B_{n,1} = -1 \end{matrix}.$$

Again, in each step of the algorithm we bring the generators to a proper form as outlined in Section 2.3. The next generator pair $(G_{i+1},\ B_{i+1})$ is computed in a slightly different manner (see [16]):

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i \overline{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = \Psi_i \overline{B}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{B}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix},$$

here $\Phi_i = F_i - f_i I_i$, $\Psi_i = A_i(I_i - f_i A_i)^{-1}$, where F_i, A_i and I_i are obtained by deleting the first i rows and columns of matrices F, A and I_n respectively, f_i is an i-th diagonal element of F. Although, the equations look complicated at first glance, it turns out that the generator B does *not* need to be updated due to its trivial structure, see Section 4.3. After n steps of the algorithm, the product GB^T yields the solution of the system.

2.5 Chinese Remainder Algorithm (CRA)

The task of CRA is to reconstruct a number X from its residues $(x_1, x_2, \dots, x_k)$ modulo a set of primes $(m_1, m_2, \dots, m_k)$. A classical approach is to associate X with the *mixed-radix* (MR) digits $(\alpha_1, \dots, \alpha_k)$:

$$X = \alpha_1 V_1 + \alpha_2 V_2 + \dots + \alpha_k V_k,$$

where $V_1 = 1$, $V_j = m_1 m_2 \dots m_{j-1}$ $(2 \le j \le k)$. We use the algorithm [22] to compute the digits α_i as follows $(1 \le i \le k)$:

$$\begin{aligned} \alpha_1 &= x_1,\ \alpha_2 = (x_2 - \alpha_1)c_2 \bmod m_2, \\ \alpha_3 &= ((x_3 - \alpha_1)c_3 - (\alpha_2 V_2 c_3 \bmod m_3)) \bmod m_3,\ \dots \\ \alpha_i &= ((x_i - \alpha_1)c_i - (\alpha_2 V_2 c_i \bmod m_i) - \dots \\ &\qquad -(\alpha_{i-1} V_{i-1} c_i \bmod m_i)) \bmod m_i, \end{aligned}$$

where $c_i = (m_1 m_2 \dots m_{i-1})^{-1} \bmod m_i$ can be precomputed in advance. It is easy to see that one can compute the MR digits on the GPU because the algorithm exposes some parallelism.

3. CUDA PROGRAMMING MODEL

Starting with the G80 series, NVIDIA GPUs do not have separated fragment and vertex processors. Instead, they are unified in Streaming Multiprocessors (SMs) capable of running shader programs as well as general purpose parallel programs. For instance, the GTX 280 contains 30 SMs. The SM can be regarded as a 32-lane SIMD vector processor because a group of 32 threads called a *warp* always executes same instruction. Accordingly, a data-dependent branch causes a warp to *diverge*, i.e., the SM has to serialize execution of all taken branch paths. Different warps are free from the divergence problem.

CUDA [1] is a heterogeneous serial-parallel programming model. In other words, a CUDA program executes serial code on the host interleaved with parallel threads execution on the GPU. To manage large number of threads that can work cooperatively, CUDA groups them in *thread blocks.* Each block contains up to 512 threads that can share data in fast on-chip memory and synchronize with barriers. Thread blocks are arranged in a *grid* that is launched on a single CUDA program or *kernel.* The blocks of a grid execute independently from each other. Hence, sequentially dependent algorithms must be split up in two or more kernels.

CUDA memory hierarchy is built on 6 memory spaces. These include: *Register file* which is a set of physical registers (16Kb per SM) split evenly between all active threads of a block.[1] *Local memory* is a private space used for per-thread temporary data and register spils. Each SM has 16Kb of low-latency on-chip *shared memory* can be accessed by all threads of a block and is organized in 16 banks to speed-up concurrent access. Bank conflicts are resolved by warp serialization and broadcast mechanism. Read-only *texture* and *constant* memory spaces as well as read-write *global* memory have lifetime of an application and are visible to all thread blocks of a grid. Texture and constant memory spaces are cached on the device. Global memory has no on-chip cache and is of much higher latency than shared memory. To use bandwidth efficiently, the graphics hardware tries to combine

[1] The advantage of static register allocation is that context switching comes almost for free, however this incurs register pressure – a formidable problem in GPU programming.

separate thread memory accesses to a single wide memory access which is also known as *memory coalescing.* The second generation NVIDIA Tesla cards (GT200 series) are much less restrictive in what concerns the memory access patterns for which coalescing can be achieved.

4. THE ALGORITHM

We start with a high-level description of the algorithm. Then, we consider subalgorithms for univariate resultants and polynomial interpolation in detail. Next, we briefly discuss the realization of fast modular arithmetic on the GPU. Finally, in the last subsection we outline the main implementation details of the algorithm.

4.1 Algorithm overview

As mentioned in the beginning, our approach follows the ideas of Collins' algorithm. To compute the resultant of two polynomials in $\mathbb{Z}[x, y]$, we map them to a prime field using several modular homomorphisms. The number of prime moduli in use depends on the resultant coefficients' bit-length given by Hadamard's bound [17]. For each modular image (for each prime m_i) we compute resultants at $x = \alpha_0, x = \alpha_1, \dots \in \mathbb{Z}_{m_i}$. Each univariate resultant is computed using the displacement structure approach, see Section 4.2. The degree bound (or the number of evaluation points) can be obtained using the rows and columns of Sylvester matrix. As shown in [17], one can use both lower and upper bounds for the resultant degree which is advantageous for sparse polynomials. In the next step, resultants over $\mathbb{Z}_{m_i}[x]$ are recovered through polynomial interpolation, see Section 4.3. Afterwards, the modular images of polynomials are lifted using the Chinese remaindering giving the final solution.

An important issue is how to deal with "bad" primes and evaluation points. With the terminology from [17], *a prime m is said to be bad if $f_p \equiv 0 \bmod m$ or $g_q \equiv 0 \bmod m$.* Similarly, *an evaluation point $\alpha \in \mathbb{Z}_m$ is bad if $f_p(\alpha) \equiv 0 \bmod m$ or $g_q(\alpha) \equiv 0 \bmod m$.* Dealing with "bad" primes is easy: we can eliminate them right away during the initial modular reduction of polynomial coefficients performed on the CPU. To handle "bad" evaluation points, we run the GPU algorithm with an *excess* amount of points (typically 1–2% more than required). The same idea is applied to deal with *non-strongly regular* Sylvester matrices.[1] In the essence, non-strong regularity indicates that there is a non-trivial relation between polynomial coefficients which, as our tests confirm, is a rare case on the average(see [9, Section 5] for experiments).

That is why, if for some $\alpha_k \in \mathbb{Z}_m$ the denominators vanish, instead of using some sophisticated methods, we simply *ignore* the result and take another evaluation point. In a very "unlucky" case when we cannot reconstruct the resultant because of the lack of points, we launch another grid to compute extra information. This can be exemplified as follows. Consider two polynomials:

$$\begin{aligned} f &= y^8 + y^6 - 3y^4 - 3y^3 + (x+6)y^2 + 2y - 5x \\ g &= (2x^3 - 13)y^6 + 5y^4 - 4y^2 - 9y + 10x + 1. \end{aligned}$$

Now, if we evaluate them at points $x = 0 \dots 100000$, it is easy to check that the corresponding Sylvester matrix is *non-strongly regular* only for a single point $x = 2$. Hence, we have enough information to recover the result.

[1]In fact, both cases correspond to zero denominator, and therefore, are indistinguishable from the algorithm's perspective.

4.2 Computing univariate resultants

Let $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$ be the generator matrices associated with two polynomials f and g as defined in Section 2.2. In each iteration we update these matrices and collect one factor of the resultant at a time. After n iterations ($n = p + q$) the generators vanish completely, and the product of collected factors yields the resultant.[2] The optimized algorithm is given below:

```
 1: procedure resultant_univariate(f : Polynomial, g : Polynomial)
 2:     p = degree(f), q = degree(g), n = p + q
 3:     f ← f/f_p                                    ▷ convert f to monic form
 4:     let G = (a, b), B = (c, d)                   ▷ set up the generators
 5:     for j = 0 to q − 1 do                        ▷ simplified iterations
 6:         b_i ← b_i − a_i b_j  for ∀i = j + 1 . . . p + j   ▷ rotations
 7:         c_{2q−j} = b_j                           ▷ update a single entry of c
 8:         a_{i+1} ← a_i  for ∀i = j . . . n − 2    ▷ shift-down
 9:     end for
10:     l_a = 1, l_c = 1                             ▷ denominators and
11:     res = 1, l_res = 1                           ▷ the resultant are set to 1
12:     for j = q to n − 1 do                        ▷ the remaining p iterations
13:         for ∀i = j . . . n − 1                   ▷ multiply by rotation matrices
14:             a_i ← l_a(a_i c_j + b_i d_j), b_i ← l_c(a_i b_j − b_i a_j),
15:             c_i ← l_c(c_i a_j + d_i b_j), d_i ← l_a(c_i d_j − d_i c_j)
16:
17:         l_c = l_a l_c^2, l_a = a_j              ▷ update denominators and
18:         res = res · c_j, l_res = l_res · l_c     ▷ the resultant
19:                                   ▷ shift-down the first generator columns
20:         a_{i+1} ← a_i, c_{i+1} ← c_i  for ∀i = j . . . n − 2
21:     end for
22:     return res · (f_p)^q / l_res                 ▷ return the resultant
23: end procedure
```

In what follows, we will denote the iterations $j = 0 \dots q-1$ and $j = q \dots n-1$ as type S and T iterations respectively. Note that, the division in lines 3 and 22 is realized by the Montgomery inverse algorithm [7] with improvements from [19]. Though the algorithm is serial, the number of iterations is bounded by the moduli bit-length (24 bits), for details see [9, Appendix A].

Our algorithm is based on the observation that $B \equiv 0$ at the beginning of the algorithm, except for $c_0 = d_q = 1$. Moreover, if we ensure that the polynomial f is monic, i.e., $f_p \equiv 1$, we can observe that the vectors $\mathbf{a}$, $\mathbf{c}$ and $\mathbf{d}$ remain *unchanged* during the first q iterations of the algorithm (with the exception of a single entry c_q). Indeed, if f is monic we have that: $D = a_0c_0 + b_0d_0 = a_0 \equiv 1$ (see Section 2.3), hence we can get rid of the denominators completely which greatly simplifies the vector update. By the same token, the computed resultant factors are *unit* during the first q iterations. Thus, we do not need to collect them. At the end, we have to multiply the resultant by $(f_p)^q$ as to compensate for running the algorithm on monic f.

4.3 Polynomial interpolation

Suppose $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$ are the generators defined in Section 2.4. Again, we take into account that B has only two non-zero entries: $c_0 = 1$ and $d_n = -1$.[3] As a result, we can skip updating B throughout all n iterations of the algorithm. Furthermore, the vector $\mathbf{a}$ does not need to be multiplied by the rotation matrix because:

[2]Recall that, in each iteration the size of generators decreases by 1.

[3]Here n denotes the number of interpolation points.

Listing 1 24-bit modular arithmetic on the GPU

```
 1: procedure mul_mod(a, b, m, invm)                          ▷ computes $a \cdot b \bmod m$
 2:     hf = uint2float_rz(umul24hi(a, b))                   ▷ compute 32 MSB of the product, convert to floating-point
 3:     prodf = fmul_rn(hf, invm)                            ▷ multiply by invm=$2^{16}/m$ in floating-point
 4:     l = float2uint_rz(prodf)                             ▷ integer truncation: $l = \lfloor hi \cdot 2^{16}/m \rfloor$
 5:     r = umul24(a, b) − umul24(l, m)                      ▷ now $r \in [-2m + \varepsilon; m + \varepsilon]$ with $0 \le \varepsilon < m$
 6:     if $r < 0$ then $r = r +$ umul24(m, 0x1000002) fi    ▷ single multiply-add instruction: $r = r + m \cdot 2$
 7:     return umin(r, r − m)                                ▷ return $r = a \cdot b \bmod m$
 8: end procedure
 9: procedure sub_mul_mod(x1, y1, x2, y2, m, invm1, invm2)    ▷ computes $(x_1y_1 - x_2y_2) \bmod m$
10:     hf1 = uint2float_rz(umul24hi(x1, y1))
11:     hf2 = uint2float_rz(umul24hi(x2, y2))                ▷ two inlined MUL_MOD operations
12:     pf1 = fmul_rn(hf1, invm1), pf2 = fmul_rn(hf2, invm1) ▷ multiply by invm1 = $2^{16}/m$ in floating-point
13:     l1 = float2uint_rz(pf1), l2 = float2uint_rz(pf2)     ▷ truncate the results to nearest integer
14:     r = mc + umul24(x1, y1) − umul24(l1, m) − umul24(x2, y2) + umul24(l2, m)   ▷ intermediate product r, $mc = m \cdot 100$
15:     rf = uint2float_rn(r) ∗ invm2 + e23                  ▷ multiply by invm2 = $1/m$ and truncate, e23 = $2^{23}$, rf = $\lfloor r/m \rfloor$
16:     r = r − umul24(float_as_int(rf), m)
17:     return $r < 0$ ? r + m : r
18: end procedure
```

$\overline{a}_i = l_a(a_ic_0 + b_id_0) \equiv l_aa_i$ (see Section 2.3). Also, observe that only n entries of the generator G are *non-zero* at a time. Thus, we can use some sort of a "sliding window" approach, i.e., only n relevant entries of G are updated in each iteration. The pseudocode for polynomial interpolation is given below:

```
 1: procedure interpolate($\mathbf{x}$ : Vector, $\mathbf{y}$ : Vector, n : Integer)
 2:         ▷ returns a polynomial f, s.t., $f(x_i) = y_i$, $0 \le i < n$
 3:     let $G = (\mathbf{a}, \mathbf{b})$                ▷ set up the generator matrix
 4:     $l_{int} = 1$                                      ▷ set the denominator to 1
 5:     for $j = 0$ to $n - 1$ do
 6:                                   ▷ multiply by the rotation matrix
 7:         $\mathbf{b}_i \leftarrow \mathbf{b}_i\mathbf{a}_j - \mathbf{a}_i\mathbf{b}_j$ for $\forall i = j + 1 \ldots j + n - 1$
 8:         $l_{int} = l_{int} \cdot \mathbf{a}_j$         ▷ update the denominator
 9:              ▷ update the last non-zero entries of $\mathbf{a}$ and $\mathbf{b}$
10:         $\mathbf{b}_{j+n} = -\mathbf{b}_j$, $\mathbf{a}_{n+j+1} = 1$, $s = 0$, $t = 0$
11:         if $(i > j$ and $i < n)$ then $s = \mathbf{a}_i$, $t = \mathbf{x}_i$
12:         elif $(i > n$ and $i \le j + n)$ then $s = \mathbf{a}_{i-1}$, $t = 1$ fi
13:                                   ▷ multiply $\mathbf{a}$ by the matrix $\Phi_j$
14:         $\mathbf{a}_i \leftarrow s \cdot t - \mathbf{a}_i \cdot \mathbf{x}_j$ for $\forall i = j + 1 \ldots j + n$
15:     end for              ▷ divide the result by the denominator
16:     $\mathbf{b}_i \leftarrow -\mathbf{b}_i/l_{int}$ for $\forall i = n \ldots 2n - 1$
17:     return $b_n \ldots b_{2n-1}$     ▷ return the coefficients of f(x)
18: end procedure
```

Note that, we write the update of $\mathbf{a}$ in line 13 in a "linearized" form (using s and t) to avoid thread divergence because in the GPU realization one thread is responsible for updating a single entry of each of vectors $\mathbf{a}$ and $\mathbf{b}$.[1]

4.4 Modular arithmetic

Modular multiplication still constitutes a big problem on modern GPUs due to the limited support for integer arithmetic and particularly slow modulo ('%') operation. The GPU natively supports 24-bit integer multiplication realized by two instructions: mul24.lo and mul24.hi.[2] CUDA only provides an intrinsic for mul24.lo while the latter instruction is not available in a high-level API. To overcome this limitation, the authors of [13] suggest to use 12-bit residues because the reduction can proceed in floating-point without overflow concerns. In the other paper [3], 280-bit residues are partitioned in 10-bit limbs to facilitate multiplication. Hence, neither paper exploits the GPU multiplication capabilities at full. We access the "missing intrinsic" directly using the PTX inline assembly [2] and realize the modular reduction in floating-point, see also [8].

The procedure MUL_MOD in Listing 1 computes $a \cdot b \bmod m$. Here umul24 and umul24hi denote the intrinsics for mul24.lo and mul24.hi respectively. First, we partition the product as follows: $a \cdot b = 2^{16}hi + lo$ (32 and 16 bits parts), and use the congruence:

$$2^{16}hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo = 2^{16}hi + lo - \\ -m \cdot l = a \cdot b - l \cdot m = r, \text{ where } 0 \le \lambda < m.$$

It can be shown that $r \in [-2m + \varepsilon; m + \varepsilon]$ for $0 \le \varepsilon < m$. As a result, r fits into a 32-bit word. Thus, we can compute it as the difference of 32 *least significant bits* of the products $a \cdot b$ and $m \cdot l$ (see line 5). Finally, the reduction in lines 6–7 maps r to the valid range $[0; m - 1]$.

The next procedure SUB_MUL_MOD evaluates the expression: $(x_1y_1 - x_2y_2) \bmod m$ used in rotation formulas (see Section 2.3). It runs two MUL_MOD's in parallel with the difference that the final reduction is deferred until the subtraction in line 13 takes place. The advantage is that line 13 produces 4 multiply-add (MAD) instructions.[3] Lines 14–16 are taken from REDUCE_MOD procedure in [8] with a minor change: namely, in line 14 we use a mantissa trick [14] to multiply by $1/m$ and round the result down in a single multiply-add instruction. We have studied the efficiency of our realization using the `decuda` tool[4]. According to disassembly, the SUB_MUL_MOD operation maps to 16 GPU instructions where 6 of them are *multiply-adds*.

4.5 GPU realization

In this section we go through the main aspects of our implementation. Suppose we are given two polynomials $f, g \in \mathbb{Z}[x, y]$ with y-degrees p and q respectively, and $p \ge q$. The algorithm comprising four kernel launches is depicted in Figure 1. Grid configuration for each kernel is shown to the left.

Before going through the realization details of each GPU kernel separately we would like to outline some features

[1]Short conditional statements are likely to be replaced by predicated instructions which do not introduce branching in the GPU code.

[2]They return 32 least and most significant bits of the product of 24-bit integer operands respectively.

[3]The compiler favors MAD instructions by aggressively merging subsequent multiply and adds.

[4]`http://wiki.github.com/laanwj/decuda`

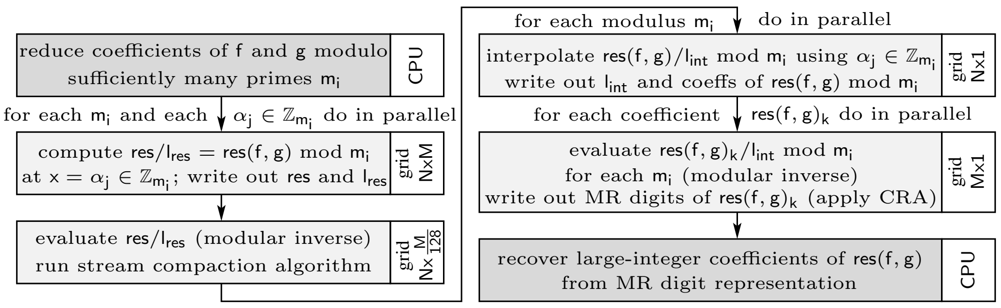

reduce coefficients of f and g modulo sufficiently many primes m_i
CPU
for each m_i and each $\alpha_j \in \mathbb{Z}_{m_i}$ do in parallel
compute $res/l_{res} = res(f, g) \bmod m_i$ at $x = \alpha_j \in \mathbb{Z}_{m_i}$; write out res and l_{res}
grid NxM
evaluate res/l_{res} (modular inverse) run stream compaction algorithm
grid Nx M/128
for each modulus m_i do in parallel
interpolate $res(f, g)/l_{int} \bmod m_i$ using $\alpha_j \in \mathbb{Z}_{m_i}$ write out l_{int} and coeffs of $res(f, g) \bmod m_i$
grid Nx1
for each coefficient $res(f, g)_k$ do in parallel
evaluate $res(f, g)_k/l_{int} \bmod m_i$ for each m_i (modular inverse) write out MR digits of $res(f, g)_k$ (apply CRA)
grid Mx1
recover large-integer coefficients of $res(f, g)$ from MR digit representation
CPU

Figure 1: Schematic view of the resultant algorithm with N is the number of moduli, and M is the number of evaluation points

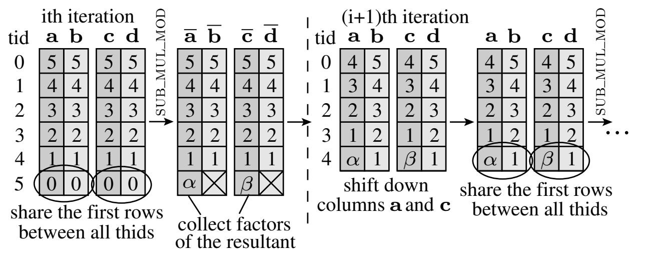

ith iteration
tid a b c d
SUB_MUL_MOD
(i+1)th iteration
share the first rows between all thids
collect factors of the resultant
shift down columns a and c
share the first rows between all thids

Figure 2: Vector updates during the type T iterations of the resultant algorithm, where tid denotes the thread ID

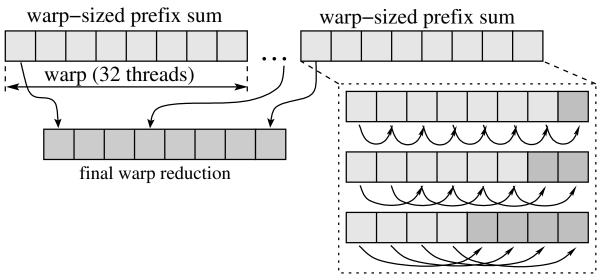


Figure 3: Warp-sized parallel reduction (prefix sum)

shared by all kernels. In our implementation we have used a number of standard optimization techniques including constant propagation via templates, loop unrolling, exploiting warp-level parallelism (parallel prefix sum), favoring small thread blocks over the large ones, etc.

An important aspect of GPU optimization is dealing with *register pressure*. We decrease register usage partly by kernel templetizations and by declaring frequently used local variables with `volatile` keyword. The latter technique forces the compiler to really keep those variables in registers and reuse them instead of inlining the expressions.

We keep the moduli set m_i and corresponding reciprocals `invm` used for reduction in *constant memory space*. This is because one thread block (except for CRA kernel) works with a *single* modulus. Hence, all threads of a block read the same value which is what the constant memory cache optimized for. Moreover, direct access from constant memory has a positive effect on reducing register usage.

4.5.1 Resultant kernel

The resultant kernel evaluates polynomials at $x = \alpha_j \in \mathbb{Z}_{m_i}$ and computes univariate resultants. It is launched on a 2D grid $N \times M$, see Figure 1. The evaluation points are implicitly given by the block indices. Four kernel instantiations with 32×2, 64, 96 and 128 threads per block are specialized for different polynomial degrees. A kernel specialization with P threads per block covers the range of degrees: $p \in [P/2;\ P-1]$.[1]

This configuration is motivated by the fact that we use $p+1$ threads to substitute x in each of $p+1$ coefficients of f (and the same for g) in parallel. The resultant algorithm consists of one outer loop split up in iterations of types S and T, see Section 4.2. In each iteration the generators are transformed using the SUB_MUL_MOD procedure, see Figure 2. The inner loop is completely vectorized: this is another reason for our block configuration. The type S iterations are unrolled by the factor of 2 for better thread occupancy, so that each thread runs two SUB_MUL_MOD operations in a row. In this way, we double the maximal degree of polynomials that can be handled, and ensure that all threads are occupied. Moreover, at the beginning of type T iterations we can guarantee that not less than half of threads are in use in the corner case ($p = P/2$).

The column vectors a and c of generators $G = (a, b)$ and $B = (c, d)$ are stored in shared memory because they need to be shifted down in each iteration. The vectors b and d reside in a register space. Observe that, the number of working threads decreases with the length of generators in the outer loop. We run the type T iterations until at least half of all threads enter an idle state. When this happens, we rebalance the workload by switching to a "lighter" version where all threads do half of a job. Finally, once the generator size descends below the warp boundary, we switch to iterations *without* sync.[2] The factors of the resultant and the denominator are collected in shared memory, then the final product $(f_p)^q \cdot \prod_i d_{ii}$ is computed efficiently using "warp-sized" parallel reduction based on [12]. The idea of reduction is to run prefix sums for several warps separately omitting synchronization barriers, and then combine the results in a final reduction step, see Figure 3. Listing 2 computes a prefix sum of 256 values stored in registers. It slightly differs from that of used in [12]. Namely, our algorithm requires less amount of shared memory per warp ($\mathsf{WS} + \mathsf{HF} + 1 = 49$ words in-

[1]The first kernel instantiation – 32×2 – calculates two resultants at a time.

[2]Recall that, the warp, as a minimal scheduling entity, is always executed synchronously on the GPU.

Listing 2 "warp-sized" parallel reduction with 256 threads

```
procedure warp_scan(x)    ▷ parallel prefix of 256 values x
            ▷ abbrev.: OP: prefix operation, TID: thread-id
                ▷ WS: warp-size (32), HF: half-warp (16),
              ▷ initialize shared memory space for reduction:
    volatile type *scan = data + HF+
        +(TID%32) + (TID/32) * (WS + HF + 1)
    scan[−HF] = Ident      ▷ ident. symbol: OP(Ident,x)=x
    scan[0] = x            ▷ save elements to shared memory
                          ▷ run warp-scans independently:
    t = OP(t, scan[−1]),  scan[0] = t
    t = OP(t, scan[−2]),  scan[0] = t
    t = OP(t, scan[−4]),  scan[0] = t
    t = OP(t, scan[−8]),  scan[0] = t
    t = OP(t, scan[−HF]),  scan[0] = t
                 ▷ "post-scan" leading elements of each warp
    volatile type *postscan = data + HF+
        +(WS * 8/32) * (WS + HF + 1)
    syncthreads()                     ▷ thread synchronize
    if TID < 8  then        ▷ post-scan 8 leading elements
        volatile type *scan2 = postscan + TID
        scan2[−HF] = Ident              ▷ put identity symbol
                  ▷ load high-order elements from shared mem
        t = data[HF + WS − 1 + TID * (WS + HF + 1)]
        scan2[0] = t,  t = OP(t, scan2[−1])
        scan2[0] = t,  t = OP(t, scan2[−2])
        scan2[0] = t,  t = OP(t, scan2[−4])
        scan2[0] = t
    fi
    syncthreads()                     ▷ thread synchronize
    t = scan[0]    ▷ read out and update scanned elements
    t = OP(t, postscan[TID/32 − 1]▷ postscan[-1] = Ident
    return t
end procedure
```

stead of 64 words) and uses fewer number of *sync* operations (2 instead of 3).

4.5.2 Modular inverse and stream compaction kernel

This kernel is also launched on a 2D grid with 128 threads per block, see Figure 1. For each modulus m_i it eliminates the resultants corresponding to "bad" evaluation points,[1] and computes the quotient res/l_{res} for each non-zero denominator l_{res} using the modular inverse. The input sequence (the set of M evaluation points for each modulus) is partitioned into 128-element chunks such that one thread computes one Montgomery modular inverse. In this way, we achieve the full occupancy due to sequential nature of the Montgomery algorithm.

To realize the efficient stream compaction we have made the following observations. **1.** The number of evaluation points M per modulus can be quite large (typically on the order of one thousand), hence it is inefficient/impossible to process all of them in one thread block. **2.** Running hierarchical stream compaction in global memory (several kernel launches) seems to be unreasonable because "bad" evaluation points occur quite rarely on the average.[2] **3.** The actual order of evaluation points does not matter for interpolation. As a result, we found the following solution optimal: Each block runs the stream compaction on its 128-element chunk using warp-sized reduction in shared memory. The reduction computes an exclusive prefix sum of a sequence of 0's and 1's

[1]In other words, those points for which denominators vanish, see Section 4.1.

[2]Our experiments show that for random polynomials typically 3–4 evaluation points are "bad" out of 10–50 thousand.

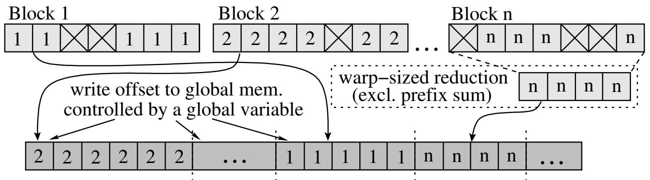

Block 1
Block 2
Block n
write offset to global mem. controlled by a global variable
warp–sized reduction (excl. prefix sum)

Figure 4: Stream compaction in shared memory

where 0's correspond to elements being eliminated. Finally, the compacted sequence is written out to global memory. The current writing position is controlled by a global variable which gets updated (atomically) each time a block outputs its results to global memory, see Figure 4. Note that, such a memory access pattern does not cause a severe performance degradation on GT200 hardware. This is because on the devices of compute capability 1.2 (SM12) and higher memory coalescence is also achieved for misaligned access, see [1].

4.5.3 Interpolation kernel

The realization is based on the algorithm from Section 4.3. One thread block is responsible for interpolating one polynomial for some prime modulus m_j. Similarly to the resultant kernel, the inner loop is unrolled and vectorized. Again, vector updates are performed using SUB_MUL_MOD operation.

We have chosen to unroll the inner loop to have high arithmetic intensity and to decrease the number of shared memory accesses per iteration because running one SUB_MUL_MOD operation per thread results in far too low thread's workload. On the other hand, unrolling increases register pressure. Therefore, in order to keep the computations fast we unroll the loop by the factor of 2 for low resultant degrees ($n \leq 256$) and by the factor of 4 for the remaining ones ($n > 256$). Our tests have shown that the kernel with 128 threads and unrolling factor of 2 runs faster than that of 64 threads and the factor of 4. Accordingly, one thread is responsible for updating 2 or 4 elements of generator G. The number of threads per block is adjusted to estimated degree n of the resultant. Hence, in the current implementation the maximal degree is limited to 2048 which corresponds to 512 threads.

The major difference is that the generator's size stays constant throughout the algorithm. That is why, in each iteration we process only n relevant entries of G in a "sliding window" fashion. We have found it advantageous to parameterize the kernel by the "data parity", that is, by n mod 2 or n mod 4 depending on the loop unrolling factor, in place of the data size n itself. This allowed us to substantially reduce branching inside the outer loop which is a big performance issue for GPU algorithms. Again, the collected factors of the denominator $\mathsf{l_{int}}$ are multiplied using the prefix sum.

4.5.4 CRA kernel

The remaining CRA kernel processes each resultant coefficient $res(f,g)_k$ independently, see Figure 1. It first divides the residues $res(f,g)_k \bmod m_j$ by respective denominators $\mathsf{l_{int}}$ computed during the interpolation (using Montgomery inverse), and runs the CRA to recover the Mixed-radix representation of the coefficient. The algorithm is rather straightforward realization of formulas from Section 2.5. It consists

Table 1: Timing the resultants of f and $g \in \mathbb{Z}[x, y]$. *1st column*: **instance number;** *2nd column*: $\mathsf{deg_y(f/g)}$: **polynomials' y-degree;** $\mathsf{deg_x(f/g)}$: **polynomials' x-degree;** bits : **coefficient bit-length;** sparse/dense: **varying density of polynomials;** *3rd column*: **resultant degree**

#	Configuration	degree	GPU	Maple	speed-up
1-2.	$\deg_y(f)$: **20**, $\deg_y(g)$: **16** $\deg_x(f)$: **7**, $\deg_x(g)$: **11**, bits : **32 / 300**	332 / 332	0.015 s / 0.14 s	1.8 s / 16.3 s	120x / 116x
3-4.	$\deg_y(f)$: **29**, $\deg_y(g)$: **20** $\deg_x(f)$: **32**, $\deg_x(g)$: **25**, bits : **64 / 250**	1361 / 1361	0.3 s / 0.89 s	36.6 s / 143.9 s	122x / 161x
5-6.	$\deg_y(f)$: **62**, $\deg_y(g)$: **40**, bits : **24** $\deg_x(f)$: **12**, $\deg_x(g)$: **10** (sparse/dense)	1088 / 1100	0.28 s / 0.33 s	25.8 s / 34.8 s	92x / 105x
7-8.	$\deg_y(f)$: **90**, $\deg_y(g)$: **80**, bits : **20** $\deg_x(f)$: **10**, $\deg_x(g)$: **10** (sparse/dense)	1502 / 1699	1.06 s / 1.4 s	76.7 s / 148.8 s	72x / 106x
9-10.	$\deg_y(f)$: **75**, $\deg_y(g)$: **60** $\deg_x(f)$: **15**, $\deg_x(g)$: **7**, bits : **32 / 100**	1425 / 1425	1.2 s / 3.1 s	100.6 s / 298.4 s	84x / 96x
11-12.	$\deg_y(f)$: **126**, $\deg_y(g)$: **80**, bits : **16** $\deg_x(f)$: **4**, $\deg_x(g)$: **7** (sparse/dense)	1150 / 1202	1.29s / 1.53 s	87.9 s / 121.3 s	68x / 79x

of one outer loop while the inner loop is vectorized. In each iteration one mixed-radix digit α_i is computed and the remaining ones $\alpha_{i+1} \dots \alpha_M$ get updated. The intermediate variables V_i are computed on-the-fly while c_i's are preloaded from the host because modular inverse is expensive.

It is worth noting that we simplify the computations by processing moduli in *increasing order*, i.e., $m_1 < m_2 < \dots < m_M$. Indeed, suppose x_i and x_j are residues modulo m_i and m_j respectively ($j < i$). Then, the expressions of the form $(x_i - x_j) \cdot c_i \bmod m_i$ can be evaluated *without* initial modular reduction of x_j because $x_j < m_i$. The modular multiplication is performed using MUL_MOD procedure from Listing 1.

The block size is adjusted to the number of moduli M. Again, for performance reasons we have unrolled the outer loop by the factor of 2 or 4 and parameterized the kernel by "data parity" ($M \bmod 2$ or $M \bmod 4$).

As a very last step, we reconstruct the multi-precision coefficients from their MR representation by evaluating the Horner form on the host machine.

5. PERFORMANCE EVALUATION AND CONCLUSIONS

We have run experiments on the *GeForce GTX 280* graphics card. The resultant algorithm from Maple 13 (32-bit version) has been tested on a *2.8Ghz Dual-Core AMD Opteron 2220SE* with 1MB L2 cache comprised in a four-processor cluster with total of 16Gb RAM under Linux platform. We have configured Maple to use *deterministic* algorithm by setting `EnvProbabilistic` to 0. This is because our approach uses Hadamard's bounds both for resultant's height and degree while the default Maple algorithm is probabilistic, in other words it uses as many moduli as necessary to produce a "stable" solution, see [17].

Performance comparison is summarized in Table 1. The GPU timing covers all stages of the algorithm including initial modular reduction and recovering multi-precision results on the host. For large integer arithmetic we have used GMP-4.3.1 library.[1] We have varied different parameters such as polynomial's x- and y-degree, the number of moduli, the coefficient bit-length and the density of polynomials (the number of non-zero entries). One can see that our algorithm achieves better speed-up for dense polynomials. This implicitly indicates that Maple uses the PRS algorithm in its core: the PRS generally performs more iterations (divisions) for dense polynomials. Whereas our algorithm is indifferent to polynomial density as it is based on linear algebra.

Also, observe that, our algorithm is faster polynomials of high x-degree. This is expected because, with the x-degree, the number of thread blocks increase (thereby, leading to better hardware utilization) while the size of Sylvester matrix remains the same. On the contrary, increasing the y-degree penalizes the performance as it causes the number of threads per block to increase. Similarly, for larger bit-lengths, the attained performance is typically higher (again because of increased degree of parallelism), with the exception that for low-degree polynomials the time for CPU modular routines becomes noticeably large as compared to the resultant computation itself.

The histogram in Figure 5 shows how the different stages of the algorithm contribute to the overall timing. Apparently, the time for resultant kernel is dominating: this is no surprise because its grid size is much larger than that of other kernels, see Figure 1). The second largest time is either initial modular reduction ('mod. reduce' in the figure) for polynomials with large coefficients, or interpolation for high-degree polynomials. Also, observe that, the time for GPU–host data transfer ('data transfer' in the figure) is negligibly small. This indicates that our algorithm is *not* memory-bound, and therefore has a big performance potential on future generation GPUs. The remaining two graphs in Figure 5 examine the running time as a function of coefficients' bit-length and polynomial degree. The bit-length only causes the number of moduli to increase resulting in a linear dependency. While polynomial's y-degree affects both moduli and evaluation points, therefore the performance degrades quadratically.

We have presented the algorithm to compute polynomial resultants on the GPU. The displacement structure approach has been proved to be well-suited for realization on massively-threaded architectures. Our results indicate a significant performance improvement over a host-based implementation. We have achieved the high arithmetic intensity of the algorithm by dynamically balancing the thread's workload and taking into account hardware-specific features such as multiply-add instruction support. Moreover, our algorithm has a great scalability potential due to the vast amount of thread blocks used by the resultant kernel.

[1]`http://gmplib.org`

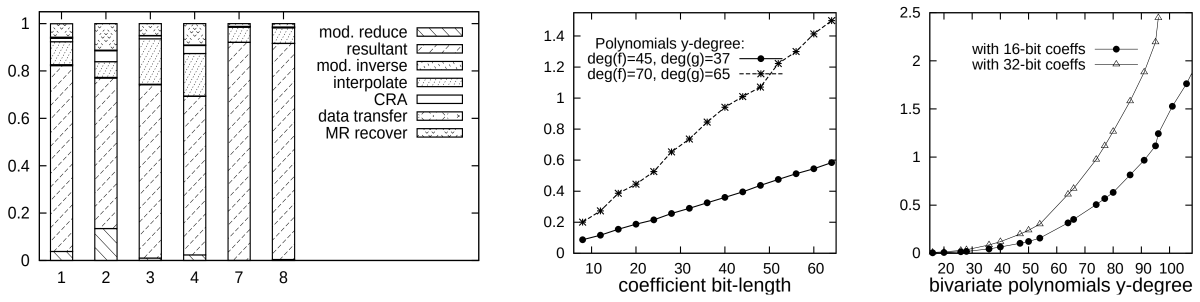


Figure 5: *Left:* **relative contribution of different stages to the overall time, x-axis: instance number in Table 1;** *Middle:* **the running time as a function of coefficient bit-length;** *Right:* **the running time as a function of polynomial y-degree. All timings are in seconds.**

As a future research directions, we would like to revisit implementation of our approach on the GPU in order to benefit from a block-level parallelism since the algorithm admits only a single-block parallelization, in that way limiting the size of input that can be handled. One possibility could be to adapt one of the recursive Schur-type algorithms for matrix factorization based on the block inversion formula, see for instance [18]. We would also like to extend our approach to other computationally-instensive symbolic algorithms that can be reformulated in matrix form: good candidates could be multivariate polynomial GCDs and subresultant sequences.

6. REFERENCES

[1] CUDA Compute Unified Device Architecture. Programming Guide. Version 2.3. NVIDIA Corp., 2009.

[2] PTX: Parallel Thread Execution. ISA Version 1.4. NVIDIA Corp., 2009.

[3] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on Graphics Cards. In *EUROCRYPT '09*, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] T. Bubeck, M. Hiller, W. Küchlin, and W. Rosenstiel. Distributed Symbolic Computation with DTS. In *IRREGULAR '95*, pages 231–248, London, UK, 1995. Springer-Verlag.

[5] S. Chandrasekaran and A. H. Sayed. A fast stable solver for nonsymmetric toeplitz and quasi-toeplitz systems of linear equations. *SIAM J. Matrix Anal. Appl.*, 19:107–139, 1998.

[6] G. E. Collins. The calculation of multivariate polynomial resultants. In *SYMSAC '71*, pages 212–222. ACM, 1971.

[7] G. de Dormale, P. Bulens, and J.-J. Quisquater. An improved Montgomery modular inversion targeted for efficient implementation on FPGA. In *FPT '04. IEEE International Conference on*, pages 441–444, 2004.

[8] P. Emeliyanenko. Efficient Multiplication of Polynomials on Graphics Hardware. In *APPT '09*, pages 134–149, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] P. Emeliyanenko. Modular Resultant Algorithm for Graphics Processors. In *ICA3PP '10*, pages 427–440, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] E. Frantzeskakis and K. Liu. A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing. *Signal Processing, IEEE Transactions on*, 42:2455–2469, Sep 1994.

[11] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for computer algebra*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1992.

[12] M. Harris, S. Sengupta, and J. D. Owens. CUDPP: CUDA Data Parallel Primitives Library. Version 1.1. http://gpgpu.org/developer/cudpp.

[13] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *AFRICACRYPT '09*, pages 350–367, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] C. Hecker. Let's get to the (floating) point. *Game Developer Magazine*, pages 19–24, 1996.

[15] H. Hong and H. W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine. In *CONPAR 94*, pages 325–336. Springer Verlag, 1994.

[16] T. Kailath and S. Ali. Displacement structure: theory and applications. *SIAM Review*, 37:297–386, 1995.

[17] M. Monagan. Probabilistic algorithms for computing resultants. In *ISSAC '05*, pages 245–252. ACM, 2005.

[18] J. H. Reif. Efficient parallel factorization and solution of structured and unstructured linear systems. *J. Comput. Syst. Sci.*, 71(1):86–143, 2005.

[19] E. Savas and C. Koc. The Montgomery modular inverse-revisited. *Computers, IEEE Transactions on*, 49(7):763–766, 2000.

[20] A. Schönhage and E. Vetter. A New Approach to Resultant Computations and Other Algorithms with Exact Division. In *ESA'94*, pages 448–459, London, UK, 1994. Springer-Verlag.

[21] W. Schreiner. Developing A Distributed System For Algebraic Geometry. In *EURO-CM-PAR'99*, pages 137–146. Civil-Comp Press, 1999.

[22] M. Yassine. Matrix Mixed-Radix Conversion For RNS Arithmetic Architectures. In *Proceedings of 34th Midwest Symposium on Circuits and Systems*, 1991.

Parallel operations of sparse polynomials on multicores - I. Multiplication and Poisson bracket

Mickaël Gastineau
IMCCE-CNRS UMR8028, Observatoire de Paris, UPMC
Astronomie et Systèmes Dynamiques
77 Avenue Denfert-Rochereau
75014 Paris, France
gastineau@imcce.fr

ABSTRACT

The multiplication of the sparse multivariate polynomials using the recursive representations is revisited to take advantage on the multicore processors. We take care of the memory management and load-balancing in order to obtain linear speedup. The widely used Poisson bracket during the studies of the dynamical systems had been parallelized on these computers. Benchmarks are presented, comparing our implementation to the other computer algebra systems.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algebraic Algorithms

General Terms

Design, Performance

Keywords

Parallel, Sparse, Polynomial, Multiplication, Poisson bracket

1. INTRODUCTION

As many applications, such as celestial mechanics, require to handle large sparse multivariate power series, many specialized or general computer algebra systems had been developed to handle these objects at the time when most of computers had only one processor. But despite multiple cores and multiple processors are now widely available, even in laptop computers, few existing computer algebra systems, such as SDMP [16], TRIP [8] and Piranha [3], take advantage of the presence of these processor-elements to reduce the time of the computation on the multivariate sparse polynomials. The SDMP library performs the multiplication of the sparse polynomials using a heap and divides the work on the multiple processors using a static scheduling. This library stores these objects in a distributed form and could only work with integer coefficients. The computer algebra system TRIP dedicated to celestial mechanics supports several representations for the multivariate sparse polynomials in the computer's memory, such as recursive forms or distributed forms optimized for celestial mechanics. TRIP could handle floating-point numbers (hardware double-precision, multiple precision), integer or rational numbers. The multiplication of polynomials using burst-tries was investigated on multiple processors [7] but it suffers from the merge step of the burst-tries that prevents a good scalability. In the first part, we present an efficient implementation of the multiplication of the multivariate sparse polynomials using the recursive form.

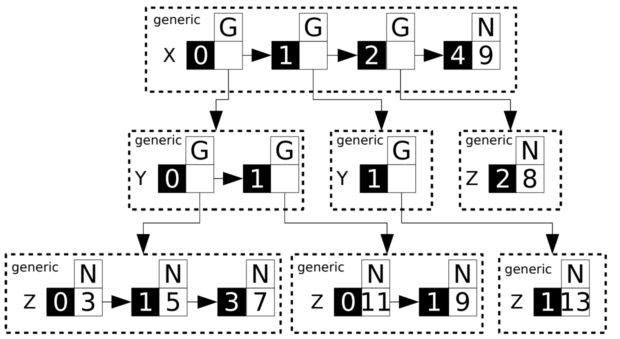
As the operator Poisson bracket on the multivariate sparse polynomials is widely used during the studies of the dynamical systems, Roldan demonstrated that this operator could benefit of the distributed architectures using MPI [18]. But a linear speedup was obtained only on few nodes for the computation on the homogeneous polynomials. In this paper, we present, for the shared memory architectures, a parallel implementation of the Poisson bracket on any sparse polynomials.

2. DATA AND PARALLEL COMPUTATIONS

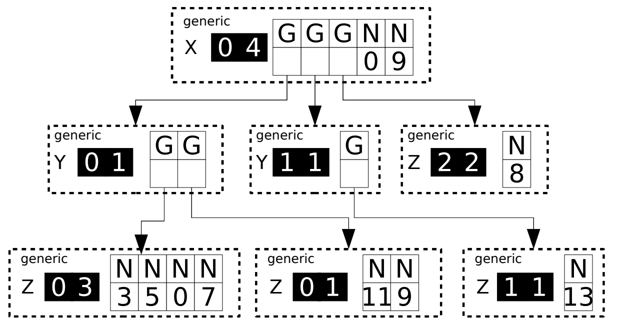
2.1 Polynomials representation

The multivariate polynomials could be represented in the memory using different data structures to keep them in a canonical form [22, 6]. Instead of using a distributed form, the polynomials are stored in a recursive container into the main memory of the computer. The multivariate polynomial in n variables is considered as a polynomial in one variable with coefficients in the polynomial ring in $n-1$ variables. These recursive data structure could be a recursive list or recursive dense vector, such as the containers implemented in the computer algebra system TRIP.

Each element of the recursive singly-linked list contains the exponent and the non-zero coefficients. These coefficients are in the polynomial ring in $n-1$ variables and are also recursive lists. As most problems need not large exponents, the exponents are encoded using hardware integers, e.g. signed 32-bit integers in TRIP. Figure 1(a) shows the representation of a multivariate polynomial as a recursive list which is used in TRIP. If a variable is not present in a term, this variable is missing in the representation (e.g., the variable y is not present in the term $8x^2z^2$ as shown in Fig. 1(a)). The complexity in search/insertion is in $O(\sum_{k=1}^{n}\deg(x_k))$. All polynomials could be represented in this structure.

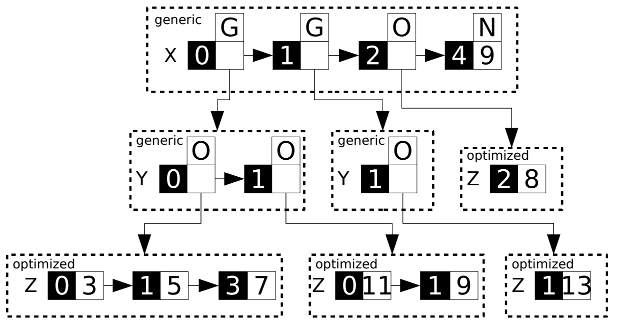


(a) recursive list (*sparse generic*)

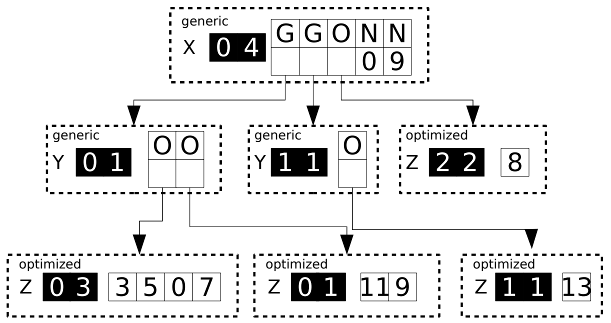


(b) recursive vector (*dense generic*)

Figure 1: representation of the polynomial $P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$ using generic container with the lexicographic order. The type information is G for generic container and N for numerical coefficient.



(a) recursive list (*sparse optimized*)



(b) recursive vector (*dense optimized*)

Figure 2: representation of the polynomial $P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$ using optimized container with the lexicographic order. The type information is G for generic container, O for optimized container and N for numerical coefficient.

The elements stored in the containers could be a polynomial or a numerical coefficient. For example in figures 1(a) and 1(b), the coefficient of x^0 is a polynomial and the coefficient of x^4 is a numerical value. To handle these different data types, the generic container requires additional information to determine the type of stored data. So each coefficient of the list is composed of two fields, type and value. This type field could be hidden if a polymorphic approach is used but, in this case, this type information is still present inside the value of the coefficient. During the computation, these generic containers require additional processing to check the data type of each element and select the appropriate algorithm. TRIP 1.0 encodes all coefficients in a generic container, even in the leaf nodes.

The recursive dense representation is similar to the previous one. The lists of tuple (degree, coefficient) is replaced by a vector of coefficients. All coefficients are stored in the array even the zero coefficients between the minimal and maximal degree. This minimal and maximal degree are kept in the header of the vector. Figure 1(b) shows the representation of a multivariate polynomial as a recursive vector which is used in TRIP. The complexity of the search and insertion algorithm is in $O(n)$ which is very efficient even for large polynomials. But this representation is not optimized for high degrees because the memory footprint of the vector becomes very large in some cases, such as $1 + x^{1000}$. In both representations, each container stores in its header the number of monomials with non-zero number coefficients in the full expanded form of the polynomial represented at that node (e.g., root containers of Fig. 1 and 2 store 8 as number of monomials).

Nevertheless, the leaf nodes contain always only numerical coefficients. To solve this bottleneck in TRIP 1.1, the leaf containers store only a single type of objects. So the type field is removed from the elements of the vector or list in the leaf nodes. This reduces the memory usage proportionately to the number of terms and reduces time consumption to check the data type. Figures 2(a) and 2(b) show the recursive list and vector representations with an optimized container for the leaf nodes. Afterwards, in the following examples, the recursive vector, respectively list, representation with a generic container for the leaf nodes is called *dense generic*, respectively *sparse generic*. The recursive vector, respectively list, representation with an optimized container for the leaf nodes is called *dense optimized*, respectively *sparse optimized*.

2.2 Memory management

As the polynomials stored into these data structures could create many small objects in the main memory, the memory management could become a bottleneck for the scalability on a computer with multiple cores [2]. PARSAC-2 [13] organizes memory in pages but its available pages are protected by a global lock. To handle these objects, we use two lock-free memory allocators, for fixed-size or any-size objects, based on [19] and [7]. The fixed-size objects, such as elements of the recursive list, do not require a header before the objects. Each thread has its own heap. They always allocate from their heap without locks. The "free" operation is more complex. If the memory was allocated by the same thread, this one released the memory without lock. If the memory was allocated by another thread, the address is pushed into a LIFO list of the distant heap using the lock-free techniques. The access from the cores to the main memory could be non uniform, such as on the Intel Xeon Nehalem processors. To hide the latency of the main memory accesses, the cores of the processors share the cache memory, which could have several megabytes. On these Non-Uniform Memory Architectures (NUMA), the allocators take care of allocating the memory on the same local node.

2.3 Parallel work

Nowadays, desktop computers have between 2 and 8 cores and server computers could have up to 24 cores or more. So the computation on the sparse polynomials could benefit from these multiple cores and could be split between these cores. A static split between them could not be performed because the recursive form could be irregular and unbalanced load occurs. Virtual tasks [20], based on S-threads [12] allow to parallelize algorithms, such as Karatsuba's method. The S-threads is based on the fork-join approach which have a significant overhead. In TRIP, a task stealing model, similar to the *work stealing* model [4], is thus used to balance the load and minimize the overhead. A pool of threads is created at the beginning of the execution of the session. Their number is equal to the number of available cores. At the beginning, these threads are in an idle state. Each parallelized task is divided into small tasks which are pushed into a LIFO queue owned by the thread. If a thread becomes idle, it looks to the queues of the other threads and steals a task if it is available. Our task stealing implementation is similar to the Intel Threading Building Blocks [17] to abstract the decomposition of loops in several tasks.

3. MULTIPLICATION

Some symmetries are present in celestial mechanics, such as d'Alembert relations in the planetary motion [14], and implies that sparse series are manipulated. The degree of these series are low during some computations, such as the computation of the Hamiltonian in the Restricted Three Body Problem [11]. Due to these sparse series and low degrees, the naive (term by term) algorithm of the multiplication is used instead of the fast methods, such as FFT or evaluation/interpolation.

3.1 Recursive dense

The product of 2 recursive dense multivariate polynomials A and B could be done in the same way as for the univariate case.

$$\begin{aligned} A(x_1, ..., x_n) &= \sum_i a_i(x_2, ..., x_n).x_1^i \\ B(x_1, ..., x_n) &= \sum_j b_j(x_2, ..., x_n).x_1^j \\ C = A \times B &= \sum_k c_k(x_2, ..., x_n).x_1^k \end{aligned}$$

Algorithm 1: FMA(A,B,C). Compute the fused multiplication-addition $C \leftarrow C + A \times B$. A, B and C are multivariate polynomials represented using a recursive sparse or dense structure.

```
Input: $A = \sum a_i x_a^i$
Input: $B = \sum b_j x_b^j$
Input: $C = \sum c_k x_c^k$
Output: $C = \sum c'_k {x'}_c^k$
 // compare order of variables
if $x_a < x_b$ then FMAcst (A,B,C)
else if $x_a > x_b$ then FMAcst (B,A,C)
else FMAsame (A,B,C)
```

Algorithm 2: FMAcst(A,B,C). Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive dense representations. Assume $x_a < x_b$ or A is a numerical value.

```
  Input: $A = \sum_{i=da_{min}}^{da_{max}} a_i x_a^i$ or A is a numerical value
  Input: $B = \sum_{j=db_{min}}^{db_{max}} b_j x_b^j$
  $n_b$ = number of monomials in the expanded form of $B$
  Input: $C = \sum_{k=dc_{min}}^{dc_{max}} c_k x_c^k$
  Output: $C = \sum_{k=dc'_{min}}^{dc'_{max}} c'_k x_{c'}^k$
  Data: $Thres$ thresold integer to perform loop in
        parallel
   // Adds a polynomial $\sum_{j=db_{min}}^{db_{max}} 0 \times x_b^j$ inside C
1 $D \leftarrow$ FindorInsertContainer $(C, x_b, db_{min}, db_{max})$
2 for $j \leftarrow db_{min}$ to $db_{max}$
   do in parallel if $(Thres < n_b)$
3 |  FMA $(a, b_j, d_j)$
4 end
5 parallel barrier
6 Put $C$ in canonical form if $D = 0$
```

where

$$c_k = \sum_{i+j=k} a_i b_j \tag{1}$$

If the naive algorithm for the univariate case is applied directly to the multivariate case, many data structures will be briefly created in the main memory. Indeed, if the computation of Eq. 1 is performed in two steps : $d \leftarrow a_i \times b_j$ and $c_k \leftarrow c_k + d$, then each computation $a_i b_j$ generates a recursive polynomial d in $x_2, ..., x_n$ and, just after, its content is merged with the current content of c_k. To avoid these unnecessarily data structures, we use a *Fused-Multiply-Add* algorithm for the multiplication of two polynomials.

The main algorithm 1 (FMA) checks the order of the most factorized variable of A and B and selects the appropriate algorithm. If A and B depend on the same main variable, then the algorithm 3 (FMAsame *dense*) is used. In the other cases, the algorithm 2 (FMAcst *dense*) is executed. The first step of this two procedures prepares the addition through the function FindorInsertContainer. This function finds or inserts a container which receives a polynomial depending on x_b by taking care of the order of the variables. If x_b is not present in the recursive structure C, this function inserts a vector depending on the variable x_b. On the other hand, if x_b is present, the corresponding container is resized if neces-

Algorithm 3: FMAsame(A,B,C). *dense*. Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive dense representations. Assume $x_a = x_b$.

```
    Input: $A = \sum_{i=da_{min}}^{da_{max}} a_i x_a^i$
    $n_a$ = number of monomials in the expanded form of $A$
    Input: $B = \sum_{j=db_{min}}^{db_{max}} b_j x_b^j$
    $n_b$ = number of monomials in the expanded form of $B$
    Input/Output: $C$ polynomial
    Data: $Thres$ thresold integer to perform loop in
          parallel
    // Adds a polynomial $\sum_{j=dab_{min}}^{dab_{max}} 0 \times x_b^j$ inside C
1   $dab_{min} \leftarrow da_{min} + db_{min}$
2   $dab_{max} \leftarrow da_{max} + db_{max}$
3   $D \leftarrow$ FindorInsertContainer $(C, x_b, dab_{min}, dab_{max})$
    // Computation
4   for $k \leftarrow dab_{min}$ to $dab_{max}$
    do in parallel if $(Thres < n_a \times n_b)$
5       for $j \in [da_{min}, da_{max}]$ and $k - j \in [db_{min}, db_{max}]$
        do
6           FMA $(a_j, b_{k-j}, d_k)$
7       end
8   end
9   parallel barrier
10  Put $C$ in canonical form if $D = 0$
```

Algorithm 4: FMAsamelarge(A,B,C). *sparse*. Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive sparse representations. Assume $x_a = x_b$ and $s_a \times s_b$ is large.

```
    Input: $A = \sum_i a_i x_a^i$ , $s_a$ = number of elements($A$),
    $n_a$ = number of monomials in the expanded form of $A$
    Input: $B = \sum_j b_j x_b^j$ , $s_b$ = number of elements($B$),
    $n_b$ = number of monomials in the expanded form of $B$
    Input/Output: $C$ polynomial
    Data: $Thres$ thresold integer to perform loop in
          parallel
    // Adds a polynomial $\sum_j 0 \times x_b^j$ inside C
1   $D \leftarrow$ FindorInsertContainer $(C, x_b)$
    // Computation
2   foreach element $\{\delta_a, a_{\delta_a}\}$ in $A$ do
3       foreach element $\{\delta_b, b_{\delta_b}\}$ in $B$ do
4           $\delta_d \leftarrow \delta_a + \delta_b$
5           $d_{\delta_d} \leftarrow$ Find/insert an element of degree $\delta_d$ in $D$
6           do in parallel if $(Thres < n_a \times n_b)$
7               FMA $(a_{\delta_a}, b_{\delta_b}, d_{\delta_d})$
8           end
9       end
10      parallel barrier
11  end
12  Put $C$ in canonical form if $D = 0$
```

sary. For example, if we need to insert a polynomial $\sum_{j=9}^{12} x^j$ inside P (Figure 2(b)), the root container will be resized to the dimension (0,12) and the function returns a reference to this container. Second example, we need to insert $\sum_{j=1}^{5} y^j$ at the location x^4 inside P, the processing moves the numerical value 9 from location x^4 to the location 0 of a new container for y with a dimension (0,5), references this new container at the location of x^4 in the root container and returns the reference to new container. The worst case for this type of algorithm occurs when they are many cancelations, such as in $(1 + x + y)(1 - x - y)$.

The second step of both procedures performs the computation of the d_k term by term. As the d_k could be computed independently [21], the outer loops of FMAsame (line 4) and FMAcst (line 2) could be easily parallelized. Only a synchronization barrier is required after the loop between the threads which process the body loop. This parallelization is done using the task-stealing model. Each different value of the counter loop k corresponds to a task. If the computation of the d_k is shared at each recursive step, the granularity is too fine and the cost of the stealing dominates largely the coefficients' arithmetic and the memory management. To avoid this problem, the coefficients are computed in parallel only if A and B have enough terms. This threshold is based on the product of their number of terms, which is the number of monomials in the full expanded form of the polynomial represented at that node. That is the reason why the number of monomials inside the children vectors is stored at each level of the recursive data structure. If this threshold is too small, performance degradation happens as shown in Fig. 3.

3.2 Recursive sparse

A similar *Fused-Multiply-Add* algorithm is used to reduce the memory management of the list. The recursive sparse representation uses the same algorithms 1 (FMA) and 2 (FMAcst) as the dense case. As the singly-linked lists are used to store the coefficients, the computation inside the procedure FMAsame cannot be done in the same way as for the dense case, except if the number of elements in the list of

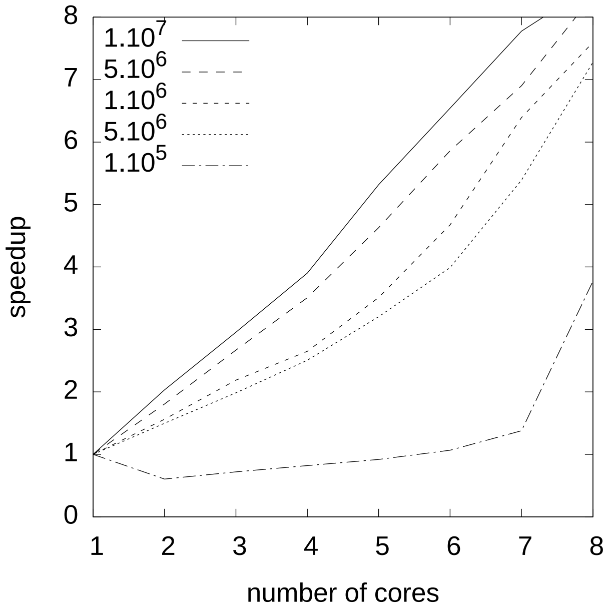


Figure 3: Speedup to compute and expand $f \times g$ with $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$ using the recursive dense representation for several threshold to stop the work stealing.

Algorithm 5: FMAsamesmall(A,B,C). *sparse.* Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive sparse representations. Assume $x_a = x_b$ and $s_a \times s_b$ is small.

```
   Input: $A = \sum_i a_i x_a^i$ , $s_a$ = number of elements($A$),
   $n_a$ = number of monomials in the expanded form of $A$
   Input: $B = \sum_j b_j x_b^j$ , $s_b$ = number of elements($B$),
   $n_b$ = number of monomials in the expanded form of $B$
   Input/Output: $C$ polynomial
   Data: $Thres$ thresold integer to perform loop in
         parallel
    // Adds a polynomial $\sum_j 0 \times x_b^j$ inside C
 1 $D \leftarrow$ FindorInsertContainer $(C, x_b)$
 2 Copy references of root elements of A in a vector $V_a$
 3 Copy references of root elements of B in a vector $V_b$
 4 Draw up the list of computed degree from $V_A$ and $V_b$
   inside a vector E (maximal size $s_a \times s_b$).
    // Computation
 5 foreach element of degree $k$ in $E$ do
 6     $d_k \leftarrow$ Find/insert an element of degree $k$ in $D$
 7     do in parallel if $(Thres < n_a \times n_b)$
 8         forall the $i + j = k$ do
 9             FMA $(a_i, b_j, d_k)$
10         end
11     end
12 end
13 parallel barrier
14 Put $C$ in canonical form if $D = 0$
```

A and B are small. The algorithm FMAsame for the sparse case just selects the appropriate algorithm 5 (FMAsamesmall) or 4 (FMAsameslarge) depending on the number of terms in A and B.

If the number of elements in the list of A and B are large, the loop (line 5) of FMAsame (*dense*) cannot be done with singly-linked lists. Indeed, this requires many traversals of the polynomial B to get the coefficient b_{k-j}. Therefore, the coefficients d_k are computed using a double loop over the list of A and B in the algorithm 4 (FMAsamelarge). In order to find or insert d_k into the list D, the traversal from the beginning of D is not performed at each computation of $a_i b_j$. Instead, as the result of $a_i b_j$ is stored after $a_i b_{j-1}$, the last position in D is kept for the next search or insertion.

The outer loop (line 2) of the algorithm 4 could not be parallelized because d_k could be accessed at the same time by different iterations, e.g. $a_i b_j$ and $a_{i+1} b_{j-1}$ access to the same location d_{i+j}. However, as the $d_i + a_i b_0$, $d_{i+1} + a_j b_1$, $d_{i+2} + a_j b_2$, ... could be computed independently and writes to a different location in D, the inner FMA statement (line 7) is parallelized. A synchronization barrier is added before the next iteration $(i+1)$ of the outer loop. As for the dense case, the splitting of the work in several parallel tasks is stopped if A and B have not enough terms (number of monomials in the distributed representation). A similar value, as the dense thresold, has been found for the sparse representation.

If the number of elements in the list of A and B are small, which is the case in most of the series used in the perturbation theories, the usage of barrier could be reduced. The following optimization is done in the algorithm 5 (FMAsamesmall). Using the stack frame to avoid memory allocation, the references of the elements of list A and B are thus copied into small vectors. It switches to a similar algorithm as the dense case to compute independently the coefficient d_k. The threshold limit for the size of the list, that is the product of the number of elements of A and B, is fixed to 2000 in order to use not too much stack memory.

Intel Xeon computer	
Processor	2 Intel Xeon X5570 quad-core
Total number of cores	8
Total number of threads	16 (hyper-threading)
L3 Cache Size	8 Mbytes by processor
Memory	32 Gbytes
Operating System	Linux kernel 2.6 - glibc 2.5
Compiler	Intel C++ 10.1 64 bits
Library	GMP 4.2.4

Intel Itanium2 computer	
Processor	4 Intel Itanium2 9040 dual-core
Total number of cores	8
L3 Cache Size	18 Mbytes by processor
Memory	16 Gbytes
Operating System	Linux kernel 2.6 - glibc 2.3
Compiler	Intel C++ 10.1 64 bits
Library	GMP 4.2.4

Table 1: Description of the computers used in the benchmarks

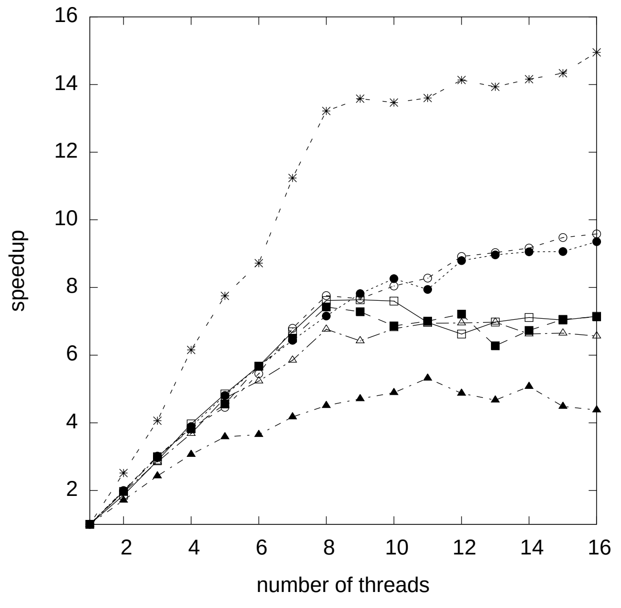
	example 1		*example 2*	
representation	time	memory	time	memory
Maple 13	1943.70	473	2310.23	10152
Singular 3.1.1	720.18	68.55	398.86	3935
SDMP	58.35	40.20	13.27	1291
TRIP 1.0				
vector	71.09	41.75	35.95	2041
list	72.88	52.55	22.68	2321
TRIP 1.1				
generic dense	55.02	41.25	22.47	2023
generic sparse	63.20	52.05	19.64	2304
optimized dense	22.54	31.13	19.63	1477
optimized sparse	29.53	41.92	16.54	1850

Table 2: Sequential execution timing expressed in seconds and memory consumption expressed in Mbytes. The numerical coefficients are integers numbers (GMP or hardware integers). The computations are performed on the Intel Xeon computer.

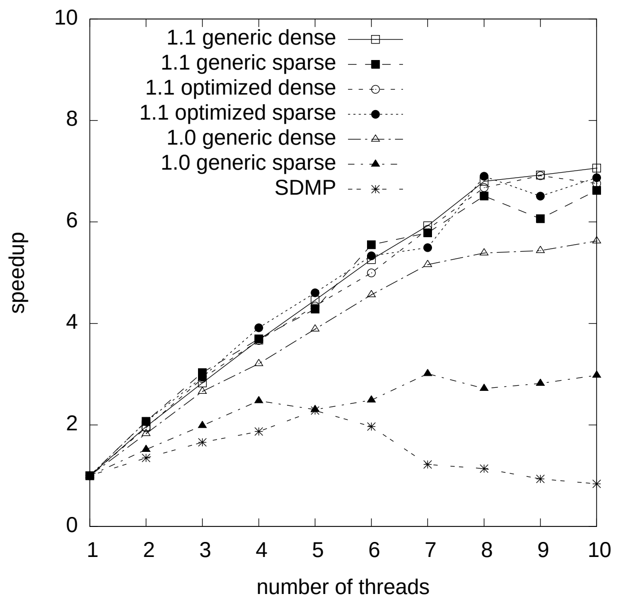
3.3 Benchmarks

The following benchmarks have been selected to test our implementation of the multiplication. These benchmarks are due to Fateman in [6] and Monagan and Pearce in [16].

- Example 1 : $f \times g$ with $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$. f ang g have 46376 terms. The result contains 635376 terms. This example is very dense.
- Example 2 : $f \times g$ with $f = (1+x+y+2z^2+3t^3+5u^5)^{16}$ and $g = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{16}$. f and g has 20349 terms. The result contains 28398035 terms. This example is very sparse. As shown in [16], a linear speedup is quite difficult to obtain on this example.

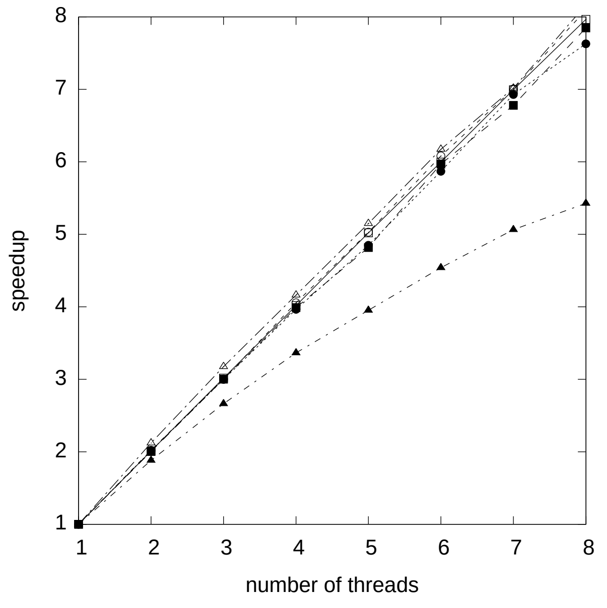


(a) example 1 (dense)

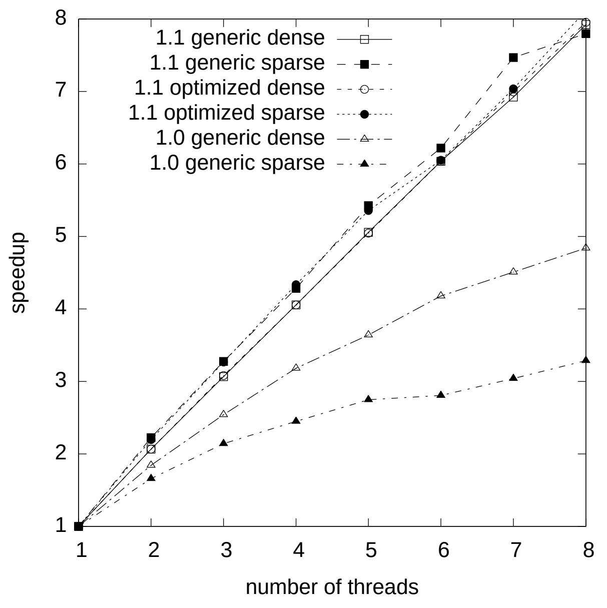


(b) example 2 (sparse)

Figure 4: Speedup to compute the examples on the Intel Xeon computer.



(a) example 1 (dense)



(b) example 2 (sparse)

Figure 5: Speedup to compute the examples on the Intel Itanium2 computer.

Table 1 describes the computer and software used to perform the benchmarks. In the sequential case, our algorithms are implemented in TRIP 1.1 and are compared to the computer algebra systems (Maple and Singular [10]) and to the existing software (SDMP library and TRIP 1.0) optimized for the sparse polynomials on shared memory computers. Table 2 shows the sequential execution timing and memory usage on the previous examples. The SDMP library stores the polynomials in a distributed form with packed exponents. The SDMP package performs the multiplication using a binary heap to sort the terms. TRIP 1.0 uses almost the same algorithm as the version 1.1 but splits the work only on the most factorized variable and does not take in account the number of terms in the children containers. This limitation could produce unbalanced-load if the data structure is very irregular. TRIP 1.0 uses the GNU MP Bignum Library (GMP)[9] to handle the integer or rational numbers. However, SDMP handles integers using hardware registers when these integers are small and comes back to the GNU MP library only when they grow [15]. TRIP 1.1

implements the same improvement for the integers smaller than $2^{63}-1$ on 64-bit computers. This optimization for small integers reduces the computation timing by a factor up to 2 on *example 1* but it has less impact on *example 2*. The specialized container for the leaf nodes of the recursive form strongly improves the computation timings. The SDMP library has better timings and uses less memory on *example 2* because it uses hardware integers up to 192 bits for the accumulation during the intermediate computations.

As shown in Fig. 4(a), the SDMP library has a super linear speedup due to its threads work on their binary heaps which fits in the cache memory. The optimized container in TRIP has a linear speedup up to 8 threads, except for the list container of the version 1.0. The presence of the hyper-threading improves the speedup up to 17% for SDMP and for the optimized containers in TRIP if we increase the number of threads to 16. The figure 5(a) shows the same computation on the computer Itanium with 8 cores without SDMP as this library is not available on the Itanium2 computer.

The SDMP library has only a speedup of 2 with 5 threads on the Xeon processor when it computes *example 2*, as shown in Fig. 4(b). With more threads, the speedup of SDMP decreases. TRIP 1.0 has the same difficulty to scale on this computer and on the Itanium computer, confirming the results founded in [16]. With the improvement of the job dispatching, the vector and list container of TRIP 1.1 have a speedup of about 6.7 with 8 threads. But it does not benefit of the hyper-threading with more threads, as this example requires more memory management. Indeed, this sparse example implies many memory operations with the recursive representations. Figure 5(b) shows that TRIP 1.1 has a linear speedup up to 8 cores on the Itanium2 and takes advantage of the larger cache.

The memory allocator could have a large impact on the execution time and on the memory foot print. Indeed, the speedup of *example 2*, which requires many memory allocations with the recursive form, drops to only 5.44 and the memory footprint is about bigger by half on 8 cores if the operating system allocator is used, as shown in Table 3. Even if the vector representation is used, similar impacts on the execution timings occur due to the resizing step (reallocation) of the vectors.

allocator	threads	time		memory
TRIP	8	8.5	(8.11x)	1851
	4	15.9	(4.33x)	1851
	1	68.7	(1x)	1841
Operating System	8	15.3	(5.44x)	3448
	4	22.1	(3.77x)	3536
	1	83.3	(1x)	2724

Table 3: Execution timing and memory usage on the *example 2* for different memory allocators on the Itanium2 computer using the *optimized sparse* representation. Timings are expressed in seconds and memory consumption expressed in Mbytes.

4. POISSON BRACKET

The equations describing a dynamical system are often expressed using the Hamiltonian formulation [1]. The Hamiltonian form $H(p, q)$, where p are the conjugate momenta of q,

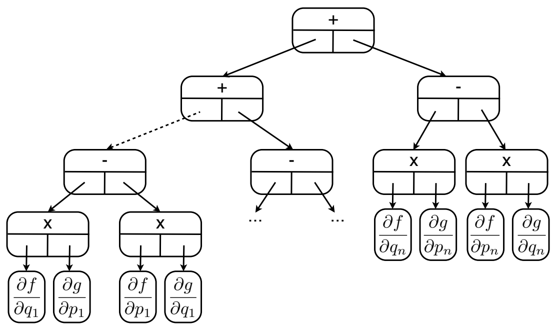

+
+
-
-
-
x
x
x
x
...
...
$\frac{\partial f}{\partial q_1}$
$\frac{\partial g}{\partial p_1}$
$\frac{\partial f}{\partial p_1}$
$\frac{\partial g}{\partial q_1}$
$\frac{\partial f}{\partial q_n}$
$\frac{\partial g}{\partial p_n}$
$\frac{\partial f}{\partial p_n}$
$\frac{\partial g}{\partial q_n}$

Figure 6: Tasks involved in the parallelization of the Poisson bracket.

is a function that verifies the following differential equations

$$\dot{p} = -\frac{\partial H}{\partial q}$$
$$\dot{q} = \frac{\partial H}{\partial p}$$

The *Poisson bracket* of the two functions $f(p, q)$ and $g(p, q)$ is defined as

$$\{f, g\} = \sum_k \left(\frac{\partial f}{\partial q_k}\frac{\partial g}{\partial p_k} - \frac{\partial f}{\partial p_k}\frac{\partial g}{\partial q_k} \right) \quad (2)$$

This Poisson bracket is an important operator to compute the canonical transformations [5]. Indeed, this operator is intensively used for the computation of the normal forms and for the Lie series.

The summation in the Poisson bracket could be viewed as a summation reduction operation of the computing task $\left(\frac{\partial f}{\partial q_k}\frac{\partial g}{\partial p_k} - \frac{\partial f}{\partial p_k}\frac{\partial g}{\partial q_k} \right)$. The four derivations of this task could be computed independantly followed by the two products. Even if the addition of the two series has a low complexity against the multiplication complexity, the addition becomes a bottleneck in the scalability. So the addition of the polynomials is parallelized. Figure 6 shows the tasks involved in the parallelization of the Poisson bracket.

To test the scalability of the implementation of these algorithms, we select the two following examples. The first example, called *example 3*, is the computation of the poisson bracket of two almost dense polynomials in 6 variables, $f(p_{1..3}, q_{1..3})$ and $g(p_{1..3}, q_{1..3})$ where

$$f = (1 + p_1 + q_1 + p_2 + q_2 + p_3 + q_3)^{12}$$
$$g = (1 + p_1^2 + q_1^2 + p_2^2 + q_2^2 + p_3^2 + q_3^2)^{12}$$

The second example, called *example 4*, is the computation of the Poisson bracket of two almost sparse polynomials in 6 variables, $H_{14}(p_{1..3}, q_{1..3})$ and $G_{14}(p_{1..3}, q_{1..3})$ where H_{14} and G_{14} are the homogeneous polynomials of the degree 14. Instead of taking all coefficients in the monomial sets of the degree 14 (11628 monomials) for these homogeneous polynomials, we set some terms to 0 in order to have a sparse poly-

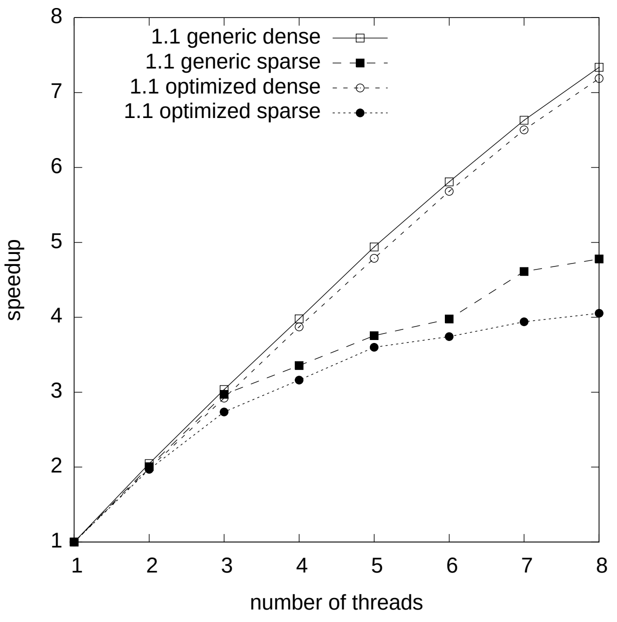


Figure 7: Speedup to compute the Poisson bracket of the example 3 on the Intel Itanium2 computer.

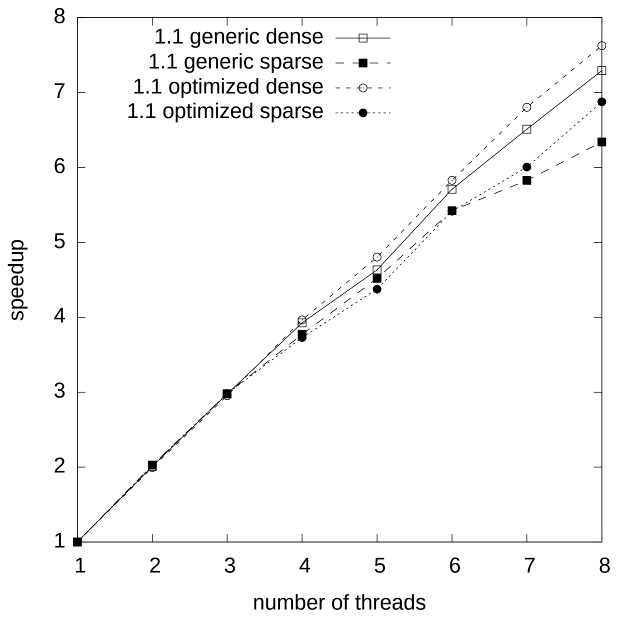


Figure 8: Speedup to compute the Poisson bracket of the example 4 on the Intel Itanium2 computer.

nomial[1]. So, H_{14} has 7722 terms and G_{14} has 5832 terms. The Poisson bracket of H_{14} and G_{14} has 142050 terms. Figures 7 and 8 show the speedup of the Poisson bracket on the Intel Itanium2 computer. The speedups of the recursive vector representations are almost linear on the two examples. On the *example 4*, the recursive list representations have the same behavior but their speedups are only about 5 with 8 threads on the *example 3*. This smaller speedup is due to many cancellations that occur during the addition step which requires more memory management for the elements of the lists. Similar behaviors have been obtained on the Intel Xeon computer.

5. CONCLUSION

The parallelization of the Poisson bracket benefits from the multiple threads but the recursive list representations do not have a linear speedup if many cancellations occur during the addition step. Recursive representations could exploit efficiently the multicore processors and obtain linear speedups for the multiplication of sparse polynomials if a dynamic scheduling, such as work stealing, is used to perform the load-balancing between the cores, even on the NUMA computers.

[1]The coefficients of the monomials $q_1^{d_1}p_1^{\bar{d}_1}q_2^{d_2}p_2^{\bar{d}_2}q_3^{d_3}p_3^{\bar{d}_3}$ are set to 0 if such that

$$\left(\sum_{j=1}^{3} d_j - \bar{d}_j\right) \bmod 3 = 0 \qquad \text{for } H_{14}$$

$$\left(\sum_{j=1}^{3} d_j - \bar{d}_j\right) \bmod 4 = 0 \qquad \text{for } G_{14}$$

6. REFERENCES

[1] V. I. Arnol'd. *Mathematical methods of classical mechanics*, volume 60 of *Graduate Texts in Mathematics*. Translated from the Russian by K. Vogtmann and A. Weinstein, Springer-Verlag, New York, NY, USA, second edition edition, 1989.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.

[3] F. Biscani. *Design and implementation of a modern algebraic manipulator for Celestial Mechanics*. PhD thesis, Centro Interdipartimentale Studi e Attivita Spaziali,Universita degli Studi di Padova, Padova, May 2008.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[5] J. R. Cary. Lie transform perturbation theory for hamiltonian systems. *Physics Reports*, 79(2):129–159, 12 1981.

[6] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37(1):4–15, 2003.

[7] M. Gastineau and J. Laskar. Development of trip: Fast sparse multivariate polynomial multiplication using burst tries. *Computational Science –ICCS 2006*, pages 446–453, 2006.

[8] M. Gastineau and J. Laskar. TRIP 1.0. TRIP Reference manual, IMCCE, Paris Observatory, 2009. `http://www.imcce.fr/trip/`.

[9] T. Granlund. GNU multiple precision arithmetic library 4.2.4, September 2008. `http://swox.com/gmp/`.

[10] G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3-1-0 — A computer algebra system for

polynomial computations, 2009. http://www.singular.uni-kl.de.

[11] A. Jorba. A methodology for the numerical computation of normal forms, centre manifolds and first integrals of hamiltonian systems. *Experiment. Math.*, 8(2):155–195, 1999.

[12] W. Küchlin. The s-threads environment for parallel symbolic computation. *Computer Algebra and Parallelism*, pages 1–18, 1992.

[13] W. Kuechlin. Parsac-2: A parallel sac-2 based on threads. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 341–353, 1991.

[14] J. Laskar. Accurate methods in general planetary theory. *Astronomy Astrophysics*, 144:133–146, Mar. 1985.

[15] M. Monagan and R. Pearce. Sparse polynomial pseudo division using a heap. Submitted to 'Milestones in Computer Algebra', J. Symb. Comput., September 2008.

[16] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[17] J. Reinders. *Intel threading building blocks.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[18] P. Roldán. *Analytical and numerical tools for the study of normally hyperbolic invariant manifolds in Hamiltonian systems and their associated dynamics.* PhD thesis, Departament de Matemàtica Aplicada I, ETSEIB-UPC., Avda. Diagonal 647, 08028 Barcelona, Spain, November 2007.

[19] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 84–94, New York, NY, USA, 2006. ACM.

[20] W. Schreiner. Virtual tasks for the paclib kernel. *Parallel Processing: CONPAR 94 —VAPP VI*, pages 533–544, 1994.

[21] P. S. Wang. Parallel polynomial operations on smps: an overview. *J. Symb. Comput.*, 21(4-6):397–410, 1996.

[22] F. Winkler. *Polynomial Algorithms in Computer Algebra.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

Parallel Computation of the Minimal Elements of a Poset

Charles E. Leiserson
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
cel@mit.edu

Liyun Li
University of Western Ontario
London ON, Canada N6A 5B7
lli287@csd.uwo.ca

Marc Moreno Maza
University of Western Ontario
London ON, Canada N6A 5B7
moreno@csd.uwo.ca

Yuzhen Xie
University of Western Ontario
London ON, Canada N6A 5B7
yxie@csd.uwo.ca

ABSTRACT

Computing the minimal elements of a partially ordered finite set (poset) is a fundamental problem in combinatorics with numerous applications such as polynomial expression optimization, transversal hypergraph generation and redundant component removal, to name a few. We propose a divide-and-conquer algorithm which is not only cache-oblivious but also can be parallelized free of determinacy races. We have implemented it in Cilk++ targeting multicores. For our test problems of sufficiently large input size our code demonstrates a linear speedup on 32 cores.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Parallel and vector implementations; G.2.2 [**Graph Theory**]: Hypergraphs

General Terms

Algorithms, Theory

Keywords

Partial ordering, minimal elements, multithreaded parallelism, Cilk, polynomial evaluation, transversal hypergraph

1. INTRODUCTION

Partially ordered sets arise in many topics of mathematical sciences. Typically, they are one of the underlying algebraic structures of a more complex entity. For instance, a finite collection of algebraic sets $V = \{V_1, \ldots, V_e\}$ (subsets of some affine space $\mathbb{K}^n$ where $\mathbb{K}$ is an algebraically closed field) naturally forms a partially ordered set (poset, for short) for the set-theoretical inclusion. Removing from V any V_i which is contained in some V_j for $i \neq j$ is an important practical question which simply translates to computing the maximal elements of the poset $(V, \subseteq)$. This simple problem is in fact challenging since testing the inclusion $V_i \subseteq V_j$ may require costly algebraic computations. Therefore, one may want to avoid unnecessary inclusion tests by using an efficient algorithm for computing the maximal elements of the poset $(V, \subseteq)$. However, this problem has received little attention in the literature [6] since the questions attached to algebraic sets (like decomposing polynomial systems) are of much more complex nature.

Another important application of the calculation of the minimal elements of a finite poset is the computation of the transversal of a hypergraph [2, 12], which itself has numerous applications, like artificial intelligence [8], computational biology [13], data mining [11], mobile communication systems [23], etc. For a given hypergraph $\mathcal{H}$, with vertex set V, the transversal hypergraph $\mathsf{Tr}(\mathcal{H})$ consists of all minimal transversals of $\mathcal{H}$: a transversal $\mathcal{T}$ is a subset of V having nonempty intersection with every hyperedge of $\mathcal{H}$, and is minimal if no proper subset of $\mathcal{T}$ is a transversal. Articles discussing the computation of transversal hypergraphs, as those discussing the removal of the redundant components of an algebraic set generally take for granted the availability of an efficient routine for computing the maximal (or minimal) elements of a finite poset.

Today's parallel hardware architectures (multi-cores, graphics processing units, etc.) and computer memory hierarchies (from processor registers to hard disks via successive cache memories) enforce revisiting many fundamental algorithms which were often designed with *algebraic complexity* as the main complexity measure and with *sequential running time* as the main performance counter. In the case of the computation of the maximal (or minimal) elements of a poset this is, in fact, almost a first visit. Up to our knowledge, there is no published papers dedicated to a general algorithm solving this question. The procedure analyzed in [18] is specialized to posets that are Cartesian products of totally ordered sets.

In this article, we propose an algorithm for computing the minimal elements of an arbitrary finite poset. Our motivation is to obtain an efficient implementation in terms of parallelism and data locality. This divide-and-conquer algorithm, presented in Section 2, follows the cache-oblivious philosophy introduced in [9]. Referring to the *multithreaded fork-join parallelism* model of Cilk [10], our algorithm has work $O(n^2)$ and span (or critical path length) $O(n)$, counting the number of comparisons, on an input poset of n elements. A straightforward algorithmic solution with a span of $O(\log(n))$ can be achieved in principle. This algorithm does not, however, take advantage of sparsity in the output, where the discovery that an element is nonminimal allows it to be removed from future comparisons with other elements. Our algorithm eliminates nonminimal elements immediately so that no work is wasted by comparing them with other elements. Moreover, our algorithm does not suffer from determinacy races and can be implemented in Cilk with `sync` as the only synchronization primitive. Experimental results show that our code can reach linear speedup on 32 cores for n large enough.

In several applications, the poset is so large that it is desirable to compute its minimal (or maximal) elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. We illustrate this strategy with two applications: polynomial expression optimization in Section 4 and transversal hypergraph generation in Section 5. In each case, we generate the poset in a divide-and-conquer manner and at the same time we compute its minimal elements. Since, for these two applications, the number of minimal elements is in general much smaller than the poset cardinality, this strategy turns out to be very effective and allows computations that could not be conducted otherwise.

This article is dedicated to Claude Berge (1926 - 2002) who introduced the third author to the combinatorics of sets.

2. THE ALGORITHM

We start by reviewing the notion of a partially ordered set. Let $\mathcal{X}$ be a set and $\preceq$ be a partial order on $\mathcal{X}$, that is, a binary relation on $\mathcal{X}$ which is reflexive, antisymmetric, and transitive. The pair $(\mathcal{X}, \preceq)$ is called a *partially ordered set*, or poset for short. If A is a subset of $\mathcal{X}$, then $(A, \preceq)$ is the poset induced by $(\mathcal{X}, \preceq)$ on A. When clear from context, we will often write A instead of $(A, \preceq)$. Here are a few examples of posets:

1. $(\mathbb{Z}, |)$ where $|$ is the divisibility relation in the ring $\mathbb{Z}$ of integer numbers,

2. $(2^S, \subseteq)$ where $\subseteq$ is the inclusion relation in the ranked lattice of all subsets of a given finite set S,

3. $(\mathcal{C}, \subseteq)$ where $\subseteq$ is the inclusion relation for the set $\mathcal{C}$ of all algebraic curves in the affine space of dimension 2 over the field of complex numbers.

An element $x \in \mathcal{X}$ is *minimal* for $(\mathcal{X}, \preceq)$ if for all $y \in \mathcal{X}$ we have: $y \preceq x \Rightarrow y = x$. The set of the elements $x \in \mathcal{X}$ which are minimal for $(\mathcal{X}, \preceq)$ is denoted by $\mathsf{Min}(\mathcal{X}, \preceq)$, or simply $\mathsf{Min}(\mathcal{X})$. From now on we assume that $\mathcal{X}$ is finite.

Algorithms 1 and 2 compute $\mathsf{Min}(\mathcal{X})$ respectively in a sequential and parallel fashion. Before describing these algorithms in more details let us first specify the programming model and data-structures. We adopt the multi-threaded programming model of Cilk [10]. In our pseudo-code, the keywords **spawn** and **sync** have the same semantics as the **cilk_spawn** and **cilk_sync** in the `Cilk++` programming language [15]. We assume that the subsets of $\mathcal{X}$ are implemented by a data-structure which supports the following operations for any subsets A, B of $\mathcal{X}$:

Split: if $|A| \geq 2$ then $\mathsf{Split}(A)$ returns a partition A^-, A^+ of A such that $|A^-|$ and $|A^+|$ differ at most by 1.

Union: $\mathsf{Union}(A, B)$ accepts two disjoint sets A, B and returns C where $C = A \cup B$;

In addition, we assume that each subset A of $\mathcal{X}$, with $k = |A|$, is encoded in a *C/C++ fashion* by an array `A` of size $\ell \geq k$. An element in A can be marked at trivial cost.

In Algorithm 1, this data-structure supports a straight-forward sequential implementation of the computation of $\mathsf{Min}(A)$, which follows from this trivial observation: an element $a_i \in A$ is minimal for $\preceq$ if for all $j \neq i$ the relation $a_j \preceq a_i$ does not hold. However, and unless the input data fits in cache, Algorithm 1 is not cache-efficient. We shall return to this point in Section 3 where cache complexity estimates are provided.

Algorithm 2 follows the cache-oblivious philosophy introduced in [9]. More precisely, and similarly to the matrix multiplication algorithm of [9], Algorithm 2 proceeds in a divide-and-conquer fashion such that when a subproblem fits into the cache, then all subsequent computations can be performed with no further cache misses. However, Algorithm 2, and other algorithms in this paper, use a threshold such that, when the size of the input is within this threshold, then a base case subroutine is called. In principle, this threshold can be set to the smallest meaningful value, say 1, and thus Algorithm 2 is cache-oblivious. In a software implementation, this threshold should be large enough so as to reduce parallelization overheads and recursive call overheads. Meanwhile, this threshold should be small enough in order to guarantee that, in the base case, cache misses are limited to cold misses. In the implementation of the matrix multiplication algorithm of [9], available in the `Cilk++` distribution, a threshold is used for the same purpose.

In Algorithm 2, when $|A| \leq \mathsf{MIN\_BASE}$, where $\mathsf{MIN\_BASE}$ is the threshold, Algorithm 1 is called. Otherwise, we partition A into a balanced pair of subsets A^-, A^+. By balanced pair, we mean that the cardinalities $|A^-|$ and $|A^+|$ differ at most by 1. The two recursive calls on A^- and A^+ in Lines 4 and 5 of Algorithm 2 will compare the elements in A^- and A^+ separately. Thus, they can be executed in parallel and free of data races. In Lines 4 and 5 we overwrite each input subset with the corresponding output one so that at Line 6 we have $A^- = \mathsf{Min}(A^-)$ and $A^+ = \mathsf{Min}(A^+)$. Line 6 is a synchronization point which ensures that the computations in Lines 4 and 5 complete before Line 7 is executed. At Line 7, cross-comparisons between A^- and A^+ are made, by means of the operation ParallelMinMerge of Algorithm 3.

We also apply a divide-and-conquer-with-threshold strategy for this latter operation, which takes as input a pair B, C of subsets of $\mathcal{X}$, such that $\mathsf{Min}(B) = B$ and $\mathsf{Min}(C) = C$ hold. Note that this

Algorithm 1: SerialMinPoset

```
   Input  : a poset $A$
   Output : $\mathsf{Min}(A)$
1  for $i$ from $0$ to $|A|-2$ do
2      if $a_i$ is unmarked then
3          for $j$ from $i+1$ to $|A|-1$ do
4              if $a_j$ is unmarked then
5                  if $a_j \preceq a_i$ then
6                      mark $a_i$ and break inner loop;
7                  if $a_i \preceq a_j$ then
8                      mark $a_j$;
9  $A \leftarrow \{$unmarked elements in $A\}$;
10 return $A$;
```

Algorithm 2: ParallelMinPoset

```
   Input  : a poset $A$
   Output : $\mathsf{Min}(A)$
1  if $|A| \leq$ MIN_BASE then
2      return SerialMinPoset($A$);
3  $(A^-, A^+) \leftarrow$ Split($A$);
4  $A^- \leftarrow$ spawn ParallelMinPoset($A^-$);
5  $A^+ \leftarrow$ spawn ParallelMinPoset($A^+$);
6  sync;
7  $(A^-, A^+) \leftarrow$ ParallelMinMerge($A^-, A^+$);
8  return Union($A^-, A^+$);
```

pair is not necessarily balanced. This leads to the following four cases in Algorithm 3, depending on the values of $|B|$ and $|C|$ w.r.t. the threshold MIN_MERGE_BASE.

Case 1: both $|B|$ and $|C|$ are no more than MIN_MERGE_BASE. We simply call the operation SerialMinMerge of Algorithm 4 which cross-compares the elements of B and C in order to remove the larger ones in each of B and C. The minimal elements from B and C are stored separately in an ordered pair (the same order as in the input pair) to remember the provenance of each result. In Cases 2, 3 and 4, this output specification helps clarifying the successive cross-comparisons when the input posets are divided into subsets.

Algorithm 3: ParallelMinMerge

```
Input  : $B, C$ such that $\mathsf{Min}(B) = B$ and
         $\mathsf{Min}(C) = C$ hold
Output : $(E, F)$ such that $E \cup F = \mathsf{Min}(B \cup C)$,
         $E \subseteq B$ and $F \subseteq C$ hold
1  if $|B| \le$ MIN_MERGE_BASE and
2     $|C| \le$ MIN_MERGE_BASE then
3      return SerialMinMerge$(B, C)$;
4  else if $|B| >$ MIN_MERGE_BASE and
5     $|C| >$ MIN_MERGE_BASE then
6      $(B^-, B^+) \leftarrow$ Split$(B)$;
7      $(C^-, C^+) \leftarrow$ Split$(C)$;
8      $(B^-, C^-) \leftarrow$ spawn
9              ParallelMinMerge$(B^-, C^-)$;
10     $(B^+, C^+) \leftarrow$ spawn
11             ParallelMinMerge$(B^+, C^+)$;
12     sync;
13     $(B^-, C^+) \leftarrow$ spawn
14             ParallelMinMerge$(B^-, C^+)$;
15     $(B^+, C^-) \leftarrow$ spawn
16             ParallelMinMerge$(B^+, C^-)$;
17     sync;
18     return (Union$(B^-, B^+)$, Union$(C^-, C^+)$);
19 else if $|B| >$ MIN_MERGE_BASE and
20    $|C| \le$ MIN_MERGE_BASE then
21     $(B^-, B^+) \leftarrow$ Split$(B)$;
22     $(B^-, C) \leftarrow$ ParallelMinMerge$(B^-, C)$;
23     $(B^+, C) \leftarrow$ ParallelMinMerge$(B^+, C)$;
24     return (Union$(B^-, B^+)$, $C$);
25 else
       // $|B| \le$ MIN_MERGE_BASE and
       // $|C| >$ MIN_MERGE_BASE
26     $(C^-, C^+) \leftarrow$ Split$(C)$;
27     $(B, C^-) \leftarrow$ ParallelMinMerge$(B, C^-)$;
28     $(B, C^+) \leftarrow$ ParallelMinMerge$(B, C^+)$;
29     return $(B$, Union$(C^-, C^+))$;
```

Case 2: both $|B|$ and $|C|$ are greater than MIN_MERGE_BASE. We split B and C into balanced pairs of subsets B^-, B^+ and C^-, C^+ respectively. Then, we recursively merge these 4 subsets, as described in Lines 8–14 in Algorithm 3. Merging B^-, C^- and merging B^+, C^+ can be executed in parallel without data races. These two computations complete half of the cross-comparisons between B and C. Then, we perform the other half of the cross-comparisons between B and C by merging B^-, C^+ and merging B^+, C^- in parallel. At the end, we return the union of the subsets from B and the union of the subsets from C.

Algorithm 4: SerialMinMerge

```
Input  : $B, C$ such that $\mathsf{Min}(B) = B$ and
         $\mathsf{Min}(C) = C$ hold
Output : $(E, F)$ such that $E \cup F = \mathsf{Min}(B \cup C)$,
         $E \subseteq B$ and $F \subseteq C$ hold
1  if $|B| = 0$ or $|C| = 0$ then
2      return $(B, C)$;
3  else
4      for i from $0$ to $|B| - 1$ do
5          for j from $0$ to $|C| - 1$ do
6              if $c_j$ is unmarked then
7                  if $c_j \preceq b_i$ then
8                      mark $b_i$ and break inner loop;
9                  if $b_i \preceq c_j$ then
10                     mark $c_j$;
11     $B \leftarrow \{$unmarked elements in $B\}$;
12     $C \leftarrow \{$unmarked elements in $C\}$;
13     return $(B, C)$;
```

Case 3, 4: either $|B|$ or $|C|$ is greater than MIN_MERGE_BASE, but not both. Here, we split the larger set into two subsets and make the appropriate cross-comparisons via two recursive calls, see Lines 15–25 in Algorithm 3.

3. COMPLEXITY ANALYSIS AND EXPERIMENTATION

We shall establish a worst case complexity for the work, the span and the cache complexity of Algorithm 2. More precisely, we assume that the input poset of this algorithm has $n \ge 1$ elements, which are pairwise incomparable for $\preceq$, that is, neither $x \preceq y$ nor $y \preceq x$ holds for all $x \ne y$. Our running time is estimated by counting the number of comparisons, that is, the number of times that the operation $\preceq$ is invoked. The costs of all other operations are neglected. The principle of Algorithm 2 is similar to that of a parallel merge-sort algorithm with a parallel merge subroutine, which might suggest that the analysis is standard. The use of thresholds requires, however, a bit of care.

We introduce some notations. For Algorithms 1 and 2 the size of the input is $|A|$ whereas for Algorithms 3 and 4 the size of the input is $|B| + |C|$. We denote by $W_1(n)$, $W_2(n)$, $W_3(n)$ and $W_4(n)$ the work of Algorithms 1, 2, 3 and 4, respectively, on an input of size n. Similarly, we denote by $S_1(n)$, $S_2(n)$, $S_3(n)$ and $S_4(n)$ the span of Algorithms 1, 2, 3 and 4, respectively, on an input of size n. Finally, we denote by N_2 and N_3 the thresholds MIN_BASE and MIN_MERGE_BASE, respectively.

Since Algorithm 4 is sequential, under our worst case assumption, we clearly have $W_4(n) = S_4(n) = \Theta(n^2)$. Similarly, we have $W_1(n) = S_1(n) = \Theta(n^2)$.

Observe that, under our worst case assumption, the cardinalities of the input sets B, C differ at most by 1, when each of Algorithms 3 and 4 is called. Hence, the work of Algorithm 3 satisfies:

$$W_3(n) = \begin{cases} W_4(n) & \text{if } n \leq N_3 \\ 4W_3(n/2) & \text{otherwise.} \end{cases}$$

This implies: $W_3(n) \leq 4^{\log_2(n/N_3)} N_3^2$ for all n. Thus we have $W_3(n) = O(n^2)$. On the other hand, our assumption implies that every element of B needs to be compared with every element of C. Therefore $W_3(n) = \Theta(n^2)$ holds. Now, the span satisfies:

$$S_3(n) = \begin{cases} S_4(n) & \text{if } n \leq N_3 \\ 2S_3(n/2) & \text{otherwise.} \end{cases}$$

This implies: $S_3(n) \leq 2^{\log_2(n/N_3)} N_3^2$ for all n. Thus we have $S_3(n) = O(nN_3)$. Moreover, $S_3(n) = \Theta(n)$ holds for $N_3 = 1$.

Next, the work of Algorithm 2 satisfies:

$$W_2(n) = \begin{cases} W_1(n) & \text{if } n \leq N_2 \\ 2W_2(n/2) + W_3(n) & \text{otherwise.} \end{cases}$$

This implies: $W_2(n) \leq 2^{\log_2(n/N_2)} N_2^2 + \Theta(n^2)$ for all n. Thus we have $W_2(n) = O(nN_2) + \Theta(n^2)$.

Finally, the span of Algorithm 2 satisfies:

$$S_2(n) = \begin{cases} S_1(n) & \text{if } n \leq N_2 \\ S_2(n/2) + S_3(n) & \text{otherwise.} \end{cases}$$

Thus we have $S_2(n) = O(N_2^2 + nN_3)$. Moreover, for $N_3 = N_2 = 1$, we have $S_2(n) = \Theta(n)$.

We proceed now with cache complexity analysis, using the ideal cache model of [9]. We consider a cache of Z words where each cache line has L words. For simplicity, we assume that the elements of a given poset are packed in an array, occupying consecutive slots, each of size 1 word. We focus on Algorithms 2 and 3, denoting by $Q_2(n)$ and $Q_3(n)$ the number of cache misses that they incur respectively on an input data of size n. We assume that the thresholds in Algorithms 2 and 3 are set to 1. Indeed, Algorithms 1 and 4 are not cache-efficient. Both may incur $\Theta(n^2/L)$ cache misses, for n large enough, whereas $Q_2(n) \in O(n/L + n^2/(ZL))$ and $Q_3(n) \in O(n^2/(ZL))$ hold, as we shall prove now. Observe first that there exist positive constants α_2 and α_3 such that we have:

$$Q_2(n) = \begin{cases} \Theta(n/L + 1) & \text{if } n \leq \alpha_2 Z \\ 2Q_2(n/2) + Q_3(n) + \Theta(1) & \text{otherwise,} \end{cases}$$

and:

$$Q_3(n) = \begin{cases} \Theta(n/L + 1) & \text{if } n \leq \alpha_3 Z \\ 4Q_3(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

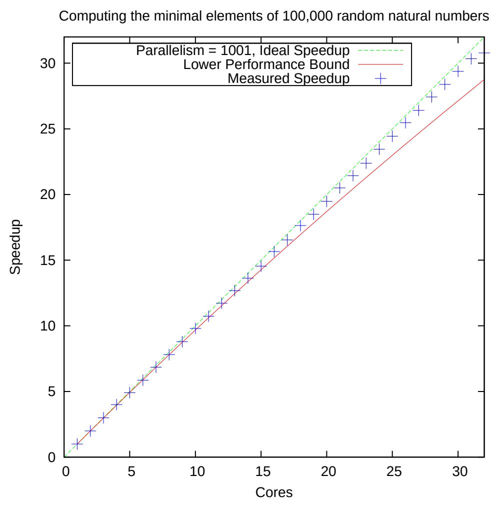
This implies: $Q_3(n) \leq 4^{\log_2(n/(\alpha_3 Z))} \Theta(Z/L)$ for all n, since $Z \in \Omega(L^2)$ holds. Thus we have $Q_3(n) \in O(n^2/(ZL))$. We deduce:

$$Q_2(n) \leq 2^k \Theta(Z/L) + \sum_{i=0}^{i=k-1} 2^i Q_3(n/2^i) + \Theta(2^k)$$

where $k = \log_2(n/(\alpha_2 Z))$. This leads to: $Q_2(n) \leq O(n/L + n^2/(ZL))$. Therefore, we have proved the following result.

PROPOSITION 1. *Assume that $\mathcal{X}$ has $n \geq 1$ elements, such that neither $x \preceq y$ nor $y \preceq x$ holds for all $x, y \in \mathcal{X}$. Set the thresholds in Algorithms 2 and 3 to 1. Then, the work, span and cache complexity of* Min($\mathcal{X}$)*, as computed by Algorithm 2, are $\Theta(n^2)$, $\Theta(n)$ and $O(n/L + n^2/(ZL))$, respectively.*

We leave for a forthcoming paper other types of analysis such as average case algebraic complexity. We turn now our attention to experimentation.



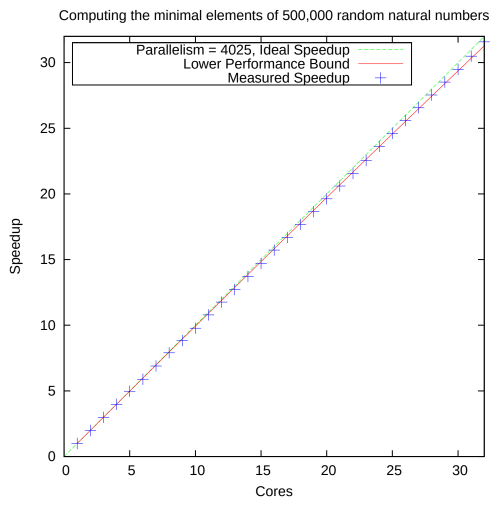


Figure 1: Scalability analysis for ParallelMinPoset **by Cilkview**

We have implemented the operation ParallelMinPoset of Algorithm 2 as a template function in Cilk++. It is designed to work for any poset providing a method Compare(a_i, a_j) that, for any two elements a_i and a_j, determines whether $a_j \preceq a_i$, or $a_i \preceq a_j$, or a_i and a_j are incomparable. Our code offers two data structures for encoding the subsets of the poset $\mathcal{X}$: one is based on arrays and the other uses the *bag structure* introduced by the first Author in [20].

For the benchmarks reported in this section, $\mathcal{X}$ is a finite set of natural numbers compared for the divisibility relation. For example, the set of the minimal elements of $\mathcal{X} = \{6, 2, 7, 3, 5, 8\}$ is $\{2, 7, 3, 5\}$. Clearly, we implement natural numbers using the type `int` of C/C++. Since checking integer divisibility is cheap, we expect that these benchmarks could illustrate the intrinsic parallel efficiency of our algorithm.

We have benchmarked our program on sets of random natural numbers, with sizes ranging from $50,000$ to $500,000$ on a 32-core machine. This machine has 8 Quad Core AMD Opteron 8354 @ 2.2 GHz connected by 8 sockets. Each core has 64 KB L1 data cache and 512 KB L2 cache. Every four cores share 2 MB of L3 cache. The total memory is 128.0 GB. We have compared the timings with MIN_BASE and MIN_MERGE_BASE being $8, 16, 32, 64$ and 128 for different sizes of input. As a result, we choose 64 for both MIN_BASE and MIN_MERGE_BASE to reach the best timing for all the test cases.

Figure 1 shows the results measured by the Cilkview [14] scalability analyzer for computing the minimal elements of $100,000$ and $500,000$ random natural numbers. The reference sequential algorithm for the speedup is Algorithm 2 running on 1 core; the running time of this latter code differs only by 0.5% or 1% from the C elision of Algorithm 2. On 1 core, the timing for computing the minimal elements of $100,000$ and $500,000$ random natural numbers is respectively 260 and 6454 seconds, which is slightly better (0.5%) than Algorithm 1. The number of minimal elements for the two sets of random natural numbers is respectively $99,919$ and $498,589$. These results demonstrate the abundant parallelism created by our divide-and-conquer algorithm and the very low parallel overhead of our program in Cilk++. We have also used Cilkview to check that our program is indeed free of data races.

4. POLYNOMIAL EXPRESSION OPTIMIZATION

We present an application where the poset can be so large that it is desirable to compute its minimal elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. As we shall see, this approach is very successful for this application.

We briefly describe this application which arises in the optimization of polynomial expressions. Let $f \in \mathbb{K}[X]$ be a multivariate polynomial with coefficients in a field $\mathbb{K}$ and with variables in $X = \{x_1, \ldots, x_n\}$. We assume that f is given as the sum of its terms, say $f = \sum_{m \in \mathsf{monoms}(f)} c_m\, m$, where $\mathsf{monoms}(f)$ denotes the set of the monomials of f and c_m is the coefficient of m in f.

A key procedure in this application computes a *partial syntactic factorization* of f, that is, three polynomials $g, h, r \in \mathbb{K}[X]$, such that f writes $gh + r$ and the following two properties hold: (1) every term in the product gh is the product of a term of g and a term of h, (2) the polynomials r and gh have no common monomials. It is easy to see that if both g and h are not constant and one has at least two terms, then evaluating f represented as $gh + r$ requires less additions/multiplications in $\mathbb{K}$ than evaluating f represented as $\sum_{m \in \mathsf{monoms}(f)} c_m\, m$, that is, as the sum of its terms. Consider for instance the polynomial $f = ax + ay + az + by + bz \in \mathbb{Q}[x, y, z, a, b]$. One possible partial syntactic factorization is $(g, h, r) = (a+b, y+z, ax)$ since we have $f = (a+b)(y+z)+ax$ and since the above two properties are clearly satisfied. Evaluating f after specializing x, y, z, a, b to numerical values will amount to 9 additions and multiplications in $\mathbb{Q}$ with $f = ax+ay+az+by+bz$ while 5 are sufficient with $f = (a + b)(y + z) + ax$.

One popular approach to reduce the size of a polynomial expression and facilitate its evaluation is to use Horner's rule. This high-school trick well-known for univariate polynomials is extended to multivariate polynomials via different schemes [4, 21, 22, 5]. However, it is difficult to compare these extensions and obtain an optimal scheme from any of them. Indeed, they all rely on selecting an appropriate ordering of the variables. Unfortunately, there are $n!$ possible orderings for n variables, which limits this approach to polynomials with moderate number of variables.

In [19], given a finite set $\mathcal{M}$ of monomials in $x_1, \ldots, x_n$, the authors propose an algorithm for computing a partial syntactic factorization (g, h, r) of f such that $\mathsf{monoms}(g) \subseteq \mathcal{M}$ holds. The complexity of this algorithm is polynomial in $|\mathcal{M}|, n, d, t$ where d and t are the total degree and number of terms of f, respectively. One possible choice for $\mathcal{M}$ would consist in taking all monomials dividing a term in f. The resulting *base monomial set* $\mathcal{M}$ would often be too large since the targeted n and d in practice are respectively in the ranges $4 \cdots 16$ and $2 \cdots 10$, which would lead $|\mathcal{M}|$ to be in the order of thousands or even millions. In [19], the set $\mathcal{M}$ is computed in the following way:

1. compute $\mathcal{G}$ the set of all non-constant $\gcd(m_1, m_2)$ where m_1, m_2 are any two monomials of f, with $m_1 \neq m_2$,
2. compute the minimal elements of $\mathcal{G}$ for the divisibility relation of monomials.

In practice, this strategy produces a more efficient evaluation representation comparing to the Horner's rule based polynomial expression optimization methods. However, there is an implementation challenge. Indeed, in practice, the number of terms of f is often in the thousands, which implies that $|\mathcal{G}|$ could be in the millions.

This has led to the design of a procedure presented through Algorithms 5, 6, 7 and 8, where $\mathcal{G}$ and $\mathsf{Min}(\mathcal{G})$ are computed concurrently in a way that the whole set $\mathcal{G}$ does not need to be stored. The proposed procedure is adapted from Algorithms 1, 2, 3 and 4. The top-level routine is Algorithm 5 which takes as input a set A of monomials. In practice one would first call this routine with $A = \mathsf{monoms}(f)$. Algorithm 5 integrates the computation of $\mathcal{G}$ and $\mathcal{M}$ (as defined above) into a "single pass" divide-and-conquer process. In Algorithms 5, 6, 7 and 8, we assume that monomials support the operations listed below, where m_1, m_2 are monomials:

- $\mathsf{Compare}(m_1, m_2)$ returns 1 if m_1 divides m_2 (that is, if m_2 is a multiple of m_1) and returns -1 if m_2 divides m_1. Otherwise, m_1 and m_2 are incomparable. This function implements the partial order used on the monomials.
- $\mathsf{Gcd}(m_1, m_2)$ computes the gcd of m_1 and m_2.

In addition, we have a data-structure for monomial sets which support the following operations, where A, B are monomial sets.

- $\mathsf{InnerPairsGcds}(A)$ computes $\mathsf{Gcd}(a_1, a_2)$ for all $a_1, a_2 \in A$ where $a_1 \neq a_2$ and returns the non-constant values only.
- $\mathsf{CrossPairsGcds}(A, B)$ computes $\mathsf{Gcd}(a, b)$ for all $a \in A$ and for all $b \in B$ and returns the non-constant values only.
- $\mathsf{SerialInnerBaseMonomials}(A)$ first calls $\mathsf{InnerPairsGcds}(A)$, and then passes the result to $\mathsf{SerialMinPoset}$ of Algorithm 1.
- $\mathsf{SerialCrossBaseMonomials}(A, B)$ applies $\mathsf{SerialMinPoset}$ to the result of $\mathsf{CrossPairsGcds}(A, B)$.

With the above basic operations, we can now describe our divide-and-conquer method for computing the base monomial set of A, that is, $\mathsf{Min}(\mathcal{G}_A)$, where $\mathcal{G}_A$ consists of all non-constant $\gcd(a_1, a_2)$ for $a_1, a_2 \in A$ and $a_1 \neq a_2$. The top-level function is $\mathsf{ParallelBaseMonomial}$ of Algorithm 5. If $|A|$ is within a threshold, namely $\mathsf{MIN\_BASE}$, the operation $\mathsf{SerialInnerBaseMonomials}(A)$ is called. Otherwise, we partition A as $A^- \cup A^+$ and observe that

$$\mathsf{Min}(\mathcal{G}_A) = \mathsf{Min}(\mathsf{Min}(\mathcal{G}_{A^-}) \cup \mathsf{Min}(\mathcal{G}_{A^+}) \cup \mathsf{Min}(\mathcal{G}_{A^-, A^+}))$$

holds where $\mathcal{G}_{A^-, A^+}$ consists of all non-constant $\gcd(x, y)$ for $(x, y) \in A^- \times A^+$. Following the above formula, we create two computational branches: (1) one for $\mathsf{Min}(\mathsf{Min}(\mathcal{G}_{A^-}) \cup \mathsf{Min}(\mathcal{G}_{A^+}))$ which is computed by the operation $\mathsf{SelfBaseMonomials}$ of Algorithm 6; (2) one for $\mathsf{Min}(\mathcal{G}_{A^-, A^+})$ which is computed by the operation $\mathsf{CrossBaseMonomials}$ of Algorithm 7. Algorithms 6 and 7 proceed in a divide-and-conquer manner:

- Algorithm 6 makes two recursive calls in parallel, then merges their results with Algorithm 3.

Algorithm 5: ParallelBaseMonomials

```
Input  : a monomial set $A$
Output : $\mathsf{Min}(\mathcal{G}_A)$ where $\mathcal{G}_A$ consists of all
         non-constant $\gcd(a_1, a_2)$ for $a_1, a_2 \in A$ and
         $a_1 \neq a_2$
1  if $|A| \leq$ MIN_BASE then
2      return SerialInnerBaseMonomials($A$);
3  else
4      $(A^-, A^+) \leftarrow$ Split($A$);
5      $B \leftarrow$ spawn SelfBaseMonomials($A^-, A^+$);
6      $C \leftarrow$ spawn CrossBaseMonomials($A^-, A^+$);
7      sync;
8      $(D_1, D_2) \leftarrow$ ParallelMinMerge($B, C$);
9      return Union($D_1, D_2$);
```

Algorithm 6: SelfBaseMonomials

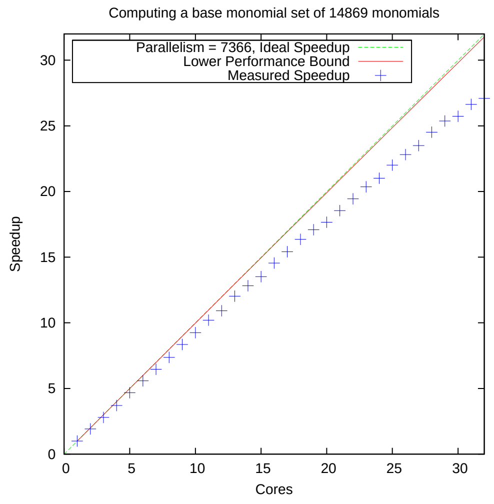
```
Input  : two disjoint monomial sets $B, C$
Output : $\mathsf{Min}(\mathcal{G}_B \cup \mathcal{G}_C)$ where $\mathcal{G}_B$ (resp. $\mathcal{G}_C$) consists
         of all non-constant $\gcd(x, y)$ for $x, y \in B$
         (resp. $C$) with $x \neq y$
1  $E \leftarrow$ spawn ParallelBaseMonomials($B$);
2  $F \leftarrow$ spawn ParallelBaseMonomials($C$);
3  sync;
4  $(D_1, D_2) \leftarrow$ ParallelMinMerge($E, F$);
5  return Union($D_1, D_2$);
```

Algorithm 7: CrossBaseMonomials

```
Input  : two disjoint monomial sets $B, C$
Output : $\mathsf{Min}(\mathcal{G}_{B,C})$ where $\mathcal{G}_{B,C}$ consists of all
         non-constant $\gcd(b, c)$ for $(b, c) \in B \times C$
1  if $|B| \leq$ MIN_MERGE_BASE then
2      return SerialCrossBaseMonomials($B, C$);
3  else
4      $(B^-, B^+) \leftarrow$ Split($B$);
5      $(C^-, C^+) \leftarrow$ Split($C$);
6      $E \leftarrow$ spawn
7          HalfCrossBaseMonomials($B^-, C^-, B^+, C^+$);
8      $F \leftarrow$ spawn
9          HalfCrossBaseMonomials($B^-, C^+, B^+, C^-$);
10     sync;
11     $(D_1, D_2) \leftarrow$ ParallelMinMerge($E, F$);
12     return Union($D_1, D_2$);
```

Algorithm 8: HalfCrossBaseMonomials

```
Input  : four monomial sets $A, B, C, D$ pairwise
         disjoint
Output : $\mathsf{Min}(\mathcal{G}_{A,B} \cup \mathcal{G}_{C,D})$ where $\mathcal{G}_{A,B}$ (resp.
         $\mathcal{G}_{C,D}$) consists of all non-constant $\gcd(x, y)$
         for $(x, y) \in A \times B$ (resp. $C \times D$)
1  $E \leftarrow$ spawn CrossBaseMonomials($A, B$);
2  $F \leftarrow$ spawn CrossBaseMonomials($C, D$);
3  sync;
4  $(G_1, G_2) \leftarrow$ ParallelMinMerge($E, F$);
5  return Union($G_1, G_2$);
```



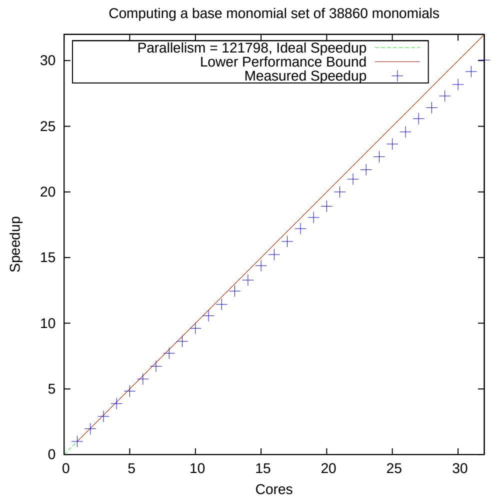


Figure 2: Scalability analysis for ParallelBaseMonomials **by Cilkview**

- Algorithm 7 uses a threshold. In the base case, computations are performed serially. Otherwise, both input monomial sets are split evenly then processed via two concurrent calls to Algorithm 8, whose results are merged with Algorithm 3.

- Algorithm 8 simply performs two concurrent calls to Algorithm 7 whose results are merged with Algorithm 3.

We have implemented these algorithms in Cilk++. A monomial is represented by an exponent vector. Each entry of an exponent vector is an unsigned `int`. A set A of input monomials is represented by an array of $|A|\,n$ unsigned `int`s where n is the number of variables. Accessing the elements in A is simply by indexing.

When the divide-and-conquer process reaches the base cases, at Lines 1–2 in Algorithm 5 and lines 1–2 in Algorithm 7, we compute either InnerPairsGcds(A) or CrossPairsGcds(B, C), followed by the computation of the minimal elements of these gcds. Here, we allocate dynamically the space to hold the gcds. Each execution of InnerPairsGcds(A) allocates memory for $|A|(|A|-1)/2$ gcds. Each execution of CrossPairsGcds(B, C) allocates space for $|B||C|$ gcds. The size of these allocated memory spaces in the base cases is rather small, which, ideally, should fit in cache. Right after computing the gcds, we compute the minimal elements of these gcds in place. In other words, we remove those gcds which are not minimal for the divisibility relation. In the Union operations, for example the $\mathsf{Union}(D_1, D_2)$ in Line 9 in Algorithm 5, we reallocate the larger one between D_1 and D_2 to accommodate $|D_1| + |D_2|$ monomials and free the space of the smaller one. This memory management strategy combined with the divide-and-conquer technique permits us to handle large sets of monomials, which could not be handled otherwise. This is confirmed by the benchmarks of our implementation.

Figure 2 gives the scalability analysis results by Cilkview for computing the base monomial sets of two large monomial sets. The first one has 14869 monomials with 28 variables; its number of minimal elements here 14. Both thresholds MIN_BASE and MIN_MERGE_BASE are set to 64. Its timing on 1 core is about 3.5 times less than the serial loop method, which is the function SerialInnerBaseMonomials. Using 32 cores we gain a speedup factor of 27 with respect to the timing on 1 core. Another monomial set has 38860 monomials with 30 variables. There are 15 minimal elements. The serial loop method for this case aborted due to memory allocation failure. However, our parallel execution reaches a speedup of 30 on 32 cores. We also notice that the ideal parallelism and the lower performance bound estimated by Cilkview for both benchmarks are very high but our measured speedup curve is lower than the lower performance bound. We attribute this performance degradation to the cost of our dynamic memory allocation.

5. TRANSVERSAL HYPERGRAPH GENERATION

Hypergraphs generalize graphs in the following way. A *hypergraph* $\mathcal{H}$ is a pair $(V, \mathcal{E})$ where V is a finite set and $\mathcal{E}$ is a set of subsets of V, called the *edges* (or *hyperedges*) of $\mathcal{H}$. The elements of V are called the *vertices* of $\mathcal{H}$. The number of vertices and edges of $\mathcal{H}$ are denoted here by $n(\mathcal{H})$ and $|\mathcal{H}|$ respectively; they are called the *order* and the size of $\mathcal{H}$. We denote by $\mathsf{Min}(\mathcal{H})$ the hypergraph whose vertex set is V and whose hyperedges are the minimal elements of the poset $(\mathcal{E}, \subseteq)$. The hypergraph $\mathcal{H}$ is said *simple* if none of its hyperedges is contained in another, that is, whenever $\mathsf{Min}(\mathcal{H}) = \mathcal{H}$ holds.

We denote by $\mathsf{Tr}(\mathcal{H})$ the hypergraph whose vertex set is V and whose hyperedges are the minimal elements of the poset $(\mathcal{T}, \subseteq)$ where $\mathcal{T}$ consists of all subsets A of V such that $A \cap E \neq \emptyset$ holds for all $E \in \mathcal{E}$. We call $\mathsf{Tr}(\mathcal{H})$ the *transversal* of $\mathcal{H}$. Let $\mathcal{H}' = (V, \mathcal{E}')$ and $\mathcal{H}'' = (V, \mathcal{E}'')$ be two hypergraphs. We denote by $\mathcal{H}' \cup \mathcal{H}''$ the hypergraph whose vertex set is V and whose hyperedge set is $\mathcal{E} \cup \mathcal{E}'$. Finally, we denote by $\mathcal{H}' \vee \mathcal{H}''$ the hypergraph whose vertex set is V and whose hyperedges are the $E' \cup E''$ for all $(E', E'') \in \mathcal{E}' \times \mathcal{E}''$. The following proposition [2] is the basis of most algorithms for computing the transversal of a hypergraph.

PROPOSITION 2. *For two hypergraphs $\mathcal{H}' = (V, \mathcal{E}')$ and $\mathcal{H}'' = (V, \mathcal{E}'')$ we have*

$$\mathsf{Tr}(\mathcal{H}' \cup \mathcal{H}'') = \mathsf{Min}(\mathsf{Tr}(\mathcal{H}') \vee \mathsf{Tr}(\mathcal{H}'')).$$

All popular algorithms for computing transversal hypergraphs, see [12, 16, 1, 7, 17], make use of the formula in Proposition 2 in an incremental manner. That is, writing $\mathcal{E} = E_1, \ldots, E_m$ and $\mathcal{H}_i = (V, \{E_1, \ldots, E_i\})$ for $i = 1 \cdots m$, these algorithms compute $\mathsf{Tr}(\mathcal{H}_{i+1})$ from $\mathsf{Tr}(\mathcal{H}_i)$ as follows

$$\mathsf{Tr}(\mathcal{H}_{i+1}) = \mathsf{Min}(\mathsf{Tr}(\mathcal{H}_i) \vee (V, \{\{v\} \mid v \in E_{i+1}\}))$$

The differences between these algorithms consist of various techniques to minimize the construction of unnecessary intermediate hyperedges. While we believe that these techniques are all important, we propose to apply Berge's formula *à la lettre*, that is, to

Algorithm 9: ParallelTransversal

```
Input   : A hypergraph $\mathcal{H}$
Output  : $\mathsf{Tr}(\mathcal{H})$
1 if $|\mathcal{H}| \leq$ TR_BASE then
2     return SerialTransversal($\mathcal{H}$);
3 $(\mathcal{H}^-, \mathcal{H}^+) \leftarrow$ Split($\mathcal{H}$);
4 $\mathcal{H}^- \leftarrow$ spawn ParallelTransversal($\mathcal{H}^-$);
5 $\mathcal{H}^+ \leftarrow$ spawn ParallelTransversal($\mathcal{H}^+$);
6 sync;
7 return ParallelHypMerge($\mathcal{H}^-, \mathcal{H}^+$);
```

Algorithm 10: ParallelHypMerge

```
Input   : $\mathcal{H}, \mathcal{K}$ such that $\mathsf{Tr}(\mathcal{H}) = \mathcal{H}$ and $\mathsf{Tr}(\mathcal{K}) = \mathcal{K}$.
Output  : $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$
1  if $|\mathcal{H}| \leq$ MERGE_HYP_BASE and
2     $|\mathcal{K}| \leq$ MERGE_HYP_BASE then
3      return SerialHypMerge($\mathcal{H}, \mathcal{K}$);
4  else if $|\mathcal{H}| >$ MERGE_HYP_BASE and
5          $|\mathcal{K}| >$ MERGE_HYP_BASE then
6      $(\mathcal{H}^-, \mathcal{H}^+) \leftarrow$ Split($\mathcal{H}$);
7      $(\mathcal{K}^-, \mathcal{K}^+) \leftarrow$ Split($\mathcal{K}$);
8      $\mathcal{L} \leftarrow$ spawn
9          HalfParallelHypMerge($\mathcal{H}^-, \mathcal{K}^-, \mathcal{H}^+, \mathcal{K}^+$);
10     $\mathcal{M} \leftarrow$ spawn
11         HalfParallelHypMerge($\mathcal{H}^-, \mathcal{K}^+, \mathcal{H}^+, \mathcal{K}^-$);
12     return Union(ParallelMinMerge($\mathcal{L}, \mathcal{M}$));
13 else if $|\mathcal{H}| >$ MERGE_HYP_BASE and
14         $|\mathcal{K}| \leq$ MERGE_HYP_BASE then
15     $(\mathcal{H}^-, \mathcal{H}^+) \leftarrow$ Split($\mathcal{H}$);
16     $\mathcal{M}^- \leftarrow$ ParallelHypMerge($\mathcal{H}^-, \mathcal{K}$);
17     $\mathcal{M}^+ \leftarrow$ ParallelHypMerge($\mathcal{H}^+, \mathcal{K}$);
18     return Union(ParallelMinMerge($\mathcal{M}^-, \mathcal{M}^+$));
19 else
       // $|\mathcal{H}| \leq$ MERGE_HYP_BASE and
       // $|\mathcal{K}| >$ MERGE_HYP_BASE
20     $(\mathcal{K}^-, \mathcal{K}^+) \leftarrow$ Split($\mathcal{K}$);
21     $\mathcal{M}^- \leftarrow$ ParallelHypMerge($\mathcal{K}^-, \mathcal{H}$);
22     $\mathcal{M}^+ \leftarrow$ ParallelHypMerge($\mathcal{K}^+, \mathcal{H}$);
23     return Union(ParallelMinMerge($\mathcal{M}^-, \mathcal{M}^+$));
```

Algorithm 11: HalfParallelHypMerge

```
Input   : four hypergraphs $\mathcal{H}, \mathcal{K}, \mathcal{L}, \mathcal{M}$
Output  : $\mathsf{Min}(\mathsf{Min}(\mathcal{H} \vee \mathcal{K}) \cup \mathsf{Min}(\mathcal{L} \vee \mathcal{M}))$
1 $\mathcal{N} \leftarrow$ spawn ParallelHypMerge($\mathcal{K}, \mathcal{H}$);
2 $\mathcal{P} \leftarrow$ spawn ParallelHypMerge($\mathcal{L}, \mathcal{M}$);
3 sync;
4 return Union(ParallelMinMerge($\mathcal{N}, \mathcal{P}$));
```

divide the input hypergraph $\mathcal{H}$ into hypergraphs $\mathcal{H}'$, $\mathcal{H}''$ of similar sizes and such that $\mathcal{H}' \cup \mathcal{H}'' = \mathcal{H}$. Our intention is to create opportunity for parallel execution. At the same time, we want to control the intermediate expression swell resulting from the computation of

$$\mathsf{Tr}(\mathcal{H}) \vee \mathsf{Tr}(\mathcal{H}').$$

To this end, we compute this expression in a divide-and-conquer manner and apply the Min operator to the intermediate results.

Algorithm 9 is our main procedure. Similarly to Algorithm 2, it proceeds in a divide-and-conquer manner with a threshold. For the base case, we call SerialTransversal($\mathcal{H}$), which can implement any serial algorithms for computing the transversal of hypergraph $\mathcal{H}$. When the input hypergraph is large enough, then this hypergraph is split into two so as to apply Proposition 2 with the two recursive calls performed concurrently. When these recursive calls return, their results are merged by means of Algorithm 10.

Given two hypergraphs $\mathcal{H}$ and $\mathcal{K}$, with the same vertex set, satisfying $\mathsf{Tr}(\mathcal{H}) = \mathcal{H}$ and $\mathsf{Tr}(\mathcal{K}) = \mathcal{K}$, the operation ParallelHypMerge of Algorithm 10 returns $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$. This operation is another instance of an application where the poset can be so large that it is desirable to compute its minimal elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. As for the application described in Section 4, one can indeed efficiently generate the elements of the poset and compute its minimal elements simultaneously.

The principle of Algorithm 10 is very similar to that of Algorithm 3. Thus, we should simply mention two points. First, Algorithm 10 uses a subroutine, namely HalfParallelHypMerge of Algorithm 11, for clarity. Secondly, the base case of Algorithm 10, calls SerialHypMerge($\mathcal{H}, \mathcal{K}$), which can implement any serial algorithms for computing $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$.

We have implemented our algorithms in Cilk++ and benchmarked our code with some well-known problems on the same 32-core machine reported in Section 3. An implementation detail which is worth to mention is data representation. We represent each hyperedge as a bit-vector. For a hypergraph with n vertices, each hyperedge is encoded by n bits. By means of this representation, the operations on the hyperedges such as inclusion test and union can be reduced to bit operations. Thus, a hypergraph with m edges is encoded by an array of $m\,n$ bits. Traversing the hyperedges is simply by moving pointers to the bit-vectors in this array.

Our test problems can be classified into three groups. The first one consists of three types of large examples reported in [16]. We summarize their features and compare the timing results in Table 1. A scalability analysis for the three large problems in data mining on a 32-core is illustrated in Figure 3. The second group considers an enumeration problem (Kuratowski hypergraph), as listed in Table 2 and Figure 4. The third group is Lovasz hypergraph [2], reported in Table 3. The sizes of the three base cases used here (TR_BASE, MERGE_HYP_BASE and MIN_MERGE_BASE) are respectively 32, 16 and 128. Our experimentation shows that the base case threshold is an important influential factor on performance. In this work, they are determined by our test runs. To predict the feasible values based on the type of a poset and the hierarchical memory of a machine would definitely help. We shall develop a tool for this purpose when we deploy our software.

In Table 1, we describe the parameters of each problem following the same notation as in [16]. The first three columns indicate respectively the number of vertices, n, the number of hyperedges, m, and the number of minimal transversals, t. The problems classified as *Threshold*, *Dual Matching* and *Data Mining* are large examples selected from [16]. We have used `thg`, a Linux executable program developed by Kavvadias and Stavropoulos in [16] for their algo-

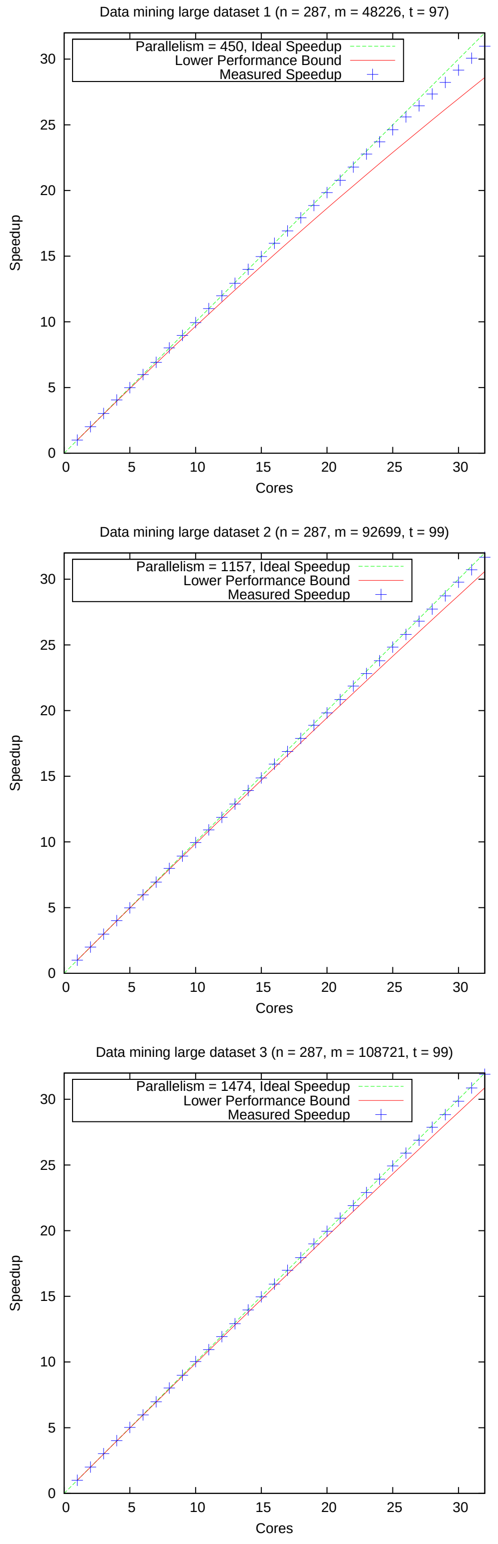


Figure 3: Scalability analysis on ParallelTransversal **for data mining problems by Cilkview**

rithm, named KS, to measure the time for solving these problems on our machine. We observed that the timing results of `thg` on our machine were very close to those reported in [16]. Thus, we show here the timing results (seconds) presented in [16] in the fourth column (KS) in our Table 1. From the comparisons in [16], the KS algorithm outperforms the algorithm of Fredman and Khachiyan as implemented by Boros et al. in [3] (BEGK) and the algorithm of Bailey et al. given in [1] (BMR) for the *Dual Matching* and *Threshold* graphs. However, for the three large problems from data mining, the KS algorithm is about 30 to 60 percent slower than the best ones between BEGK and BMR.

In the last three columns in Table 1, we report the timing (in seconds) of our program for solving these problems using 1 core and 32 cores, and the speedup factor on 32-core w.r.t on 1-core. On 1-core, our method is about 6 to 18 times faster for the selected *Dual Matching* problems and the large problems in data mining. Our program is particularly efficient for the *Threshold* graphs, for which it takes only about 0.01 seconds for each of them, while `thg` took about 11 to 82 seconds. In addition, our method shows significant speedup on multi-cores for the problems of large input size. As shown in Figure 3, for the three data mining problems, our code demonstrates linear speedup on 32 cores w.r.t the timing of the same algorithm on 1 core.

There are three sets of hypergraphs in [16] on which our method does not perform well, namely *Matching*, *Self-Dual Threshold* and *Self-Dual Fano-Plane* graphs. For these examples our code is about 2 to 50 times slower than the KS algorithm presented in [16]. Although the timing of such examples is quite small (from 0.01 to 178 s), they demonstrate the efficient techniques used in [16]. Incooperating such techniques into our algorithm is our future work.

Instance parameters			KS	ParallelTransversal		Speedup Ratio	
n	m	t	(s)	1-core (s)	32-core (s)	KS/1-core	KS/32-core
Threshold problems							
140	4900	71	11	0.01	-	1000	-
160	6400	81	23	0.01	-	2000	-
180	8100	91	44	0.01	-	4000	-
200	10000	101	82	0.02	-	4000	-
Dual Matching problems							
34	131072	17	57	9	0.57	6	100
36	262144	18	197	23	1.77	9	111
38	524288	19	655	56	3.53	12	186
40	1048576	20	2167	131	7.13	17	304
Data Mining problems							
287	48226	97	1648	92	3	18	549
287	92699	99	6672	651	21	10	318
287	108721	99	9331	1146	36	8	259

Table 1: Examples from [16]

The first family of classical hypergraphs that we have tested is related to an enumeration problem, namely the Kuratowski K_n^r hypergraphs. Table 2 gives two representative ones. This type of hypergraphs are defined by two parameters n and r. Given n distinct vertices, such a hypergraph contains all the hyperedges that have exactly r vertices. Our program achieves linear speedup on this class of hypergraphs with sufficiently large size, as reported in Table 2 and Figure 4 for K_{40}^5 and K_{30}^7. We have also used the `thg` program provided by the Authors of [16] to solve these problems. The timing for solving K_{30}^5 by the `thg` program is about 6500 seconds, which is about 70 times slower than our ParallelTransversal on 1-core. For the case of K_{40}^5 and K_{30}^7, the `thg` program did not produce a result after running for more than 15 hours.

Another classical hypergraph is the Lovasz hypergraph, which is defined by a positive integer r. Consider r finite disjoint sets $X_1, \ldots, X_r$ such that X_j has exactly j elements, for $j = 1 \cdots r$. The Lovasz hypergraph of rank r, denoted by L_r, has all its hyper-

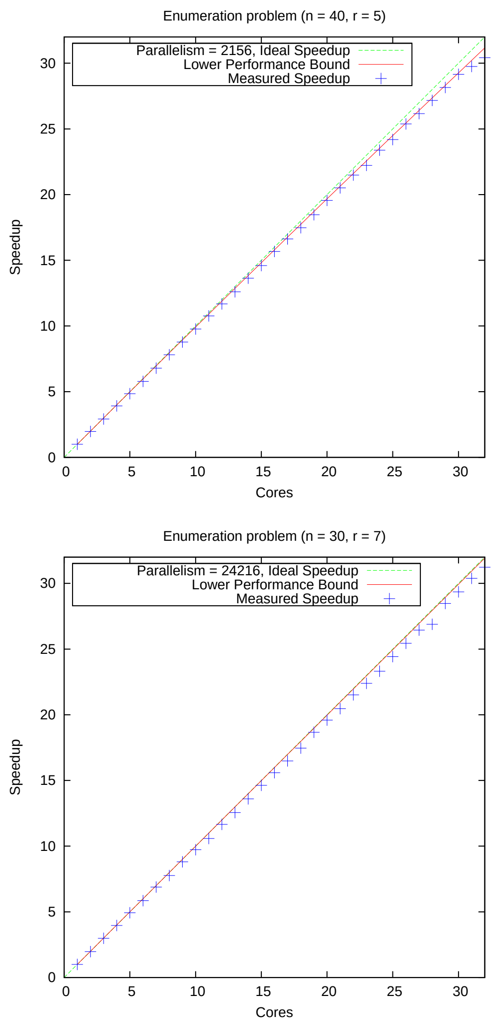


Figure 4: Scalability analysis on ParallelTransversal for K_{40}^5 and K_{30}^7 by Cilkview

edges of the form

$$X_j \cup \{x_{j+1}, \ldots, x_r\},$$

where $x_{j+1}, \ldots, x_r$ belong respectively to $X_{j+1}, \ldots, X_r$, for $j = 1 \ldots r$. We have tested our implementation with the Lovasz hypergraphs up to rank 10. For the rank 9 problem, we obtained 25 speedup on 32-core. For the one of rank 10, due to time limit, we only obtained the timing on 32-core and 16-core, which shows a linear speedup from 16 cores to 32 cores. The `thg` program solves the problem of rank 8 in 8000 seconds. For the problems of rank 9 and 10, the `thg` program did not complete within 15 hours.

6. CONCLUDING REMARKS

In this paper, we have proposed a parallel algorithm for computing the minimal elements of a finite poset. Its implementation in Cilk++ on multi-cores is capable of processing large posets that a serial implementation could not process. Moreover, for sufficiently large input data set, our code reaches linear speedup on 32 cores.

We have integrated our algorithm into two applications. One is polynomial expression optimization and the other one is the computation of transversal hypergraphs. In both cases, we control intermediate expression swell by generating the poset and computing

Instance parameters				KS	ParallelTransversal				
n	r	m	t	(s)	1-core	16-core		32-core	
					(s)	(s)	Speedup	(s)	Speedup
30	5	142506	27405	6500	88	6	14.7	3.5	25.0
40	5	658008	91390	>15 hr	915	58	15.8	30	30.5
30	7	2035800	593775	>15 hr	72465	4648	15.6	2320	31.2

Table 2: Tests for the Kuratowski hypergraphs

Instance parameters				KS	ParallelTransversal				
n	r	m	t	(s)	1-core	16-core		32-core	
					(s)	(s)	Speedup	(s)	Speedup
36	8	69281	69281	8000	119	13	8.9	10	11.5
45	9	623530	623530	>15 hr	8765	609	14.2	347	25.3
55	10	6235301	6235301	>15 hr	-	60509	-	30596	

Table 3: Tests for the Lovasz hypergraphs

its minimal elements concurrently. Our Cilk++ code for computing transversal hypergraphs is competitive with the implementation reported by Kavvadias and Stavropoulos in [16]. Moreover, our code outperforms the one of our colleagues on three sets of large input problems, in particular the problems from data mining. However, our code is slower than theirs on other data sets. In fact, our code is a preliminary implementation, which simply applies Berge's formula in a divide-and-conquer manner. We still need to enhance our implementation with the various techniques which have been developed for controlling expression swell in transversal hypergraph computations [12, 16, 1, 7, 17].

We are extending the work presented in this paper in different directions. First, we would like to obtain a deeper complexity analysis of our algorithm for computing the minimal elements of a finite poset. Secondly, we are adapting this algorithm to the computation of GCD-free bases and the removal of redundant components.

Acknowledgements.

Great thanks to our colleagues Dimitris J. Kavvadias and Elias C. Stavropoulos for providing us with their program (implementing the KS algorithm) and their test suite. Sincere thanks to the reviewers for their constructive comments. We are grateful to Matteo Frigo for fruitful discussions on Cilk++. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) the Mathematics of Information Technology and Complex Systems. (MITACS) of Canada and the National Science Foundation (NSF) under Grants CNS-0615215 and CCF-0621511. In addition, our benchmarks were made possible by the dedicated resource program of the Shared Hierarchical Academic Research Computing Network (SHARCNET) of Canada.

7. REFERENCES

[1] J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *ICDM '03: Proceedings of the 3rd IEEE International Conference on Data Mining*, page 485, 2003. IEEE Computer Society.

[2] C. Berge. *Hypergraphes : combinatoire des ensembles finis*. Gauthier-Villars, 1987.

[3] E. Boros, K. hachiyan, K. Elbassioni, and V. Gurvich. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals. In *Proc. of the 11th European Symposium on Algorithms (ESA)*, volume 2432, pages 556–567. LNCS, Springer, 2003.

[4] J. Carnicer and M. Gasca. Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, 54(189):231–243, 1990.

[5] M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate Horner schemes. *SIGSAM Bull.*, 38(1):8–15, 2004.

[6] C. Chen, F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. Efficient computations of irredundant triangular decompositions with the `regularchains` library. In *Proc. of the International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 268–271. Springer, 2007.

[7] G. Dong and J. Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowledge and Information Systems*, 8(2):178–202, 2005.

[8] T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and ai. In *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 549–564, 2002. Springer-Verlag.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, USA, 1999.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN*, 1998.

[11] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003.

[12] M. Hagen. Lower bounds for three algorithms for transversal hypergraph generation. *Discrete Appl. Math.*, 157(7):1460–1469, 2009.

[13] U. Haus, S. Klamt, and T. Stephen. Computing knock out strategies in metabolic networks. *ArXiv e-prints*, 2008.

[14] Y. He, C. E. Leiserson, and W. M. Leiserson. The cilkview scalability analyzer. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, New York, USA, 2010. ACM.

[15] Intel Corp. Cilk++. http://www.cilk.com/.

[16] D. J. Kavvadias and E. C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

[17] L. Khachiyan, E. Boros, K. M. Elbassioni, and V. Gurvich. A new algorithm for the hypergraph transversal problem. In *COCOON*, pages 767–776, 2005.

[18] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[19] C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Efficient evaluation of large polynomials. In *Proc.* International Congress of Mathematical Software - ICMS 2010. Springer, 2010. To appear.

[20] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 303–314, New York, USA, 2010. ACM.

[21] J. M. Pena. On the multivariate Horner scheme. *SIAM J. Numer. Anal.*, 37(4):1186–1197, 2000.

[22] J. M. Pena and T. Sauer. On the multivariate Horner scheme ii: running error analysis. *Computing*, 65(4):313–322, 2000.

[23] S. Sarkar and K. N. Sivarajan. Hypergraph models for cellular mobile communication systems. *IEEE Transactions on Vehicular Technology*, 47(2):460–471, 1998.

Parallel Disk-Based Computation for Large, Monolithic Binary Decision Diagrams

Daniel Kunkle* Vlad Slavici Gene Cooperman*
Northeastern University
360 Huntington Ave.
Boston, Massachusetts 02115
{kunkle,vslav,gene}@ccs.neu.edu

ABSTRACT

Binary Decision Diagrams (BDDs) are widely used in formal verification. They are also widely known for consuming large amounts of memory. For larger problems, a BDD computation will often start thrashing due to lack of memory within minutes. This work uses the parallel disks of a cluster or a SAN (storage area network) as an extension of RAM, in order to efficiently compute with BDDs that are orders of magnitude larger than what is available on a typical computer. The use of parallel disks overcomes the bandwidth problem of single disk methods, since the bandwidth of 50 disks is similar to the bandwidth of a *single* RAM subsystem. In order to overcome the latency issues of disk, the Roomy library is used for the sake of its latency-tolerant data structures. A breadth-first algorithm is implemented. A further advantage of the algorithm is that RAM usage can be very modest, since its largest use is as buffers for open files. The success of the method is demonstrated by solving the 16-queens problem, and by solving a more unusual problem — counting the number of tie games in a three-dimensional 4×4×4 tic-tac-toe board.

Categories and Subject Descriptors: I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms — *Algebraic algorithms*, *Analysis of algorithms*; E.1 [**Data Structures**]: *Distributed data structures*

General Terms: Algorithms, Experimentation, Performance

Keywords: parallel, disk-based, binary decision diagram, BDD, breadth-first algorithm

1. INTRODUCTION

There are three widespread symbolic techniques for formal verification currently in use: binary decision diagrams, SAT solvers, and explicit state model checking. Binary decision diagrams (BDDs) have a particular attraction in being able to compactly represent a Boolean function on k Boolean variables. Because BDDs are a representation of the general solution, they are an example of *symbolic model checking* — in contrast with SAT-solving and explicit state model checking, which find only particular solutions.

BDDs have found widespread use in industry — especially for circuit verification. Unfortunately, as with many formal verification methods, it tends to quickly run out of space. It is common for a BDD package to consume the full memory of RAM in a matter of minutes to hours.

To counter this problem of storage, a novel approach is demonstrated that uses parallel disks. While we are aware of approaches using the local disks of one computer and of approaches using the RAM of multiple computers or CPUs, we are not aware of work simultaneously using the parallel disks of many computers. The parallel disks may be the local disks of a cluster, or the disks of a SAN in a cluster.

The parallelism is not for the sake of speeding up the computation. It is only to support the added storage of parallel disks. Because aggregate disk bandwidth is the bottleneck of this algorithm, more disks produce a faster computation. Nevertheless, the goal of the algorithm is to extend the available storage — *not* to provide a parallel speedup beyond the traditional sequential computation in RAM.

The new package is demonstrated by building large QBDDs (quasi-reduced BDDs) for two combinatorial problems. The first is the well-known N-queens problem, and is used for comparability with other research. The second combinatorial problem may be new in the literature. The number of tie boards are counted on a 3-dimensional tic-tac-toe board of dimensions $4 \times 4 \times 4$.

The question of comparability of this new work with previous work is a subtle one. Previously, researchers have used the disks of a single node in an effort to extend the available storage of a sequential BDD package [2, 21, 24, 25]. Researchers have also developed parallel BDD algorithms with the goal of making BDD computations run faster [11, 15, 19, 22, 28, 29]. All of the work cited was performed in the 1990s, with little progress since then. One exception is that in 2001, Minato and Ishihara [16] demonstrated a streaming BDD mechanism that incorporates pipelined parallelism. That work stores its large result on a single disk at the end of the pipeline, but it does not use parallel disks.

Given that there has been little progress in disk-based and in parallel BDDs in the last ten years, there is a question of how to produce a fair scalability comparison that takes into account the advances in hardware over that time. While the last ten years have seen only a limited growth in CPU speed,

*This work was partially supported by the National Science Foundation under Grant CNS-0916133.

they have seen a much greater growth in the size of RAM.

We use the N-queens problem to validate the scalability of our new algorithm. We do this in two ways. Recall that the N-queens problem asks for the number of solutions in placing N non-attacking queens on an $N \times N$ chess board.

In the first validation, we compare with a traditional sequential implementation on a computer with large RAM. The BuDDy package [8] was able to solve the N-queens problem for 14 queens or less, when given 8 GB of RAM. When given 56 GB of RAM, BuDDy was still not able to go beyond 14 queens. In comparison, the new program was able to solve the 16-queens problem. One measure of the size of the computation is that the size of the canonical BDD produced for 16 queens was 30.5 times larger than that for 14 queens. The number of solutions with 16 queens was 40.5 times the number of solutions with 14 queens.

A second validation of the scalability is a comparison with the more recent work of Minato and Ishihara [16]. That work used disk in 2001 to store a BDD solution to the N-queens problem. Like BuDDy, that work was also only able to solve the 14-queens problem. Nevertheless, it represented the state of the art for using BDDs to solve the N-queens problem at that time. It reported solving the 14-queens problem in 73,000 seconds with one PC and in 2,500 seconds with 100 PCs. The work of this paper solves the 16-queens problem using 64 computers in 100,000 seconds. The ROBDD for 16 queens is 30 times larger than that for 14 queens. The bandwidth of disk access has not grown by that factor of 30 over those 9 years.

To our knowledge, the BDD package presented here is the first demonstration of the practicality of a breadth-first *parallel disk-based computation*. Broadly, parallel-disk based computation uses disks as the main working memory of a computation, instead of RAM. The goal is to solve space-limited problems without significantly increasing hardware costs or radically altering existing algorithms and data structures. BDD computations make an excellent test of parallel-disk based computation because larger examples can exhaust several gigabytes of memory in just minutes (see experimental results in Section 5).

There are two fundamental challenges in using disk-based storage as main memory:

> **Bandwidth**: roughly, the bandwidth of a single disk is 50 times less than that of a single RAM subsystem (100 MB/s versus 5 GB/s). Our solution is to use many disks in parallel, achieving an aggregate bandwidth comparable to RAM.
>
> **Latency**: worse than bandwidth, the latency of disk is many orders of magnitude worse than RAM. Our solution is to avoid latency penalties by using streaming data access, instead of costly random access.

Further, as an optimization, we employ a hybrid approach: using serial, RAM-based methods for relatively small BDDs; transitioning to parallel disk-based methods for large BDDs.

Briefly, the main points of novelty of our approach are:

- This is the first package to efficiently use both parallel CPUs and parallel disks. More disks make the computation run faster. The package also allows computations for monolithic BDDs larger than ever before (the alternative approach is partitioning the very large BDD into smaller BDDs). There are advantages to using one monolithic BDD rather than many sub-BDDs: the duplicate work is reduced, thus reducing computation time in some cases; sometimes, partitioning a BDD requires domain-specific knowledge, while our approach does not. There are, however, good partitioning methods for which no domain-specific knowledge is necessary. In any case, our approach can be combined with BDD partitioning, to solve very very large problems.

- By using Roomy [12], the high-level algorithms are separated from the low-level memory management, garbage collection and load balancing. Most existing BDD packages have at least some of these three components integrated in the package, while our BDD package deals only with the "application" level, thus making it very flexible.

- As opposed to most other packages for manipulating very large BDDs, we do not require that a level in a BDD fit entirely in the aggregate RAM of a cluster. The RAM is used primarily to hold buffers for open files. While the number of open files can potentially grow in some circumstances, a technique analogous to the use of two levels in external sorting can be employed to again reduce the need for RAM, at a modest slowdown in performance (less than a factor of two).

Related work is described in Section 1.1. An overview of binary decision diagrams is presented in Section 2. A brief overview of the Roomy library is given in Section 3. Section 4 presents the parallel algorithms that are the foundation of this work. Section 5 presents experimental results. This is followed by a discussion of future work in Section 6.

1.1 Related Work

Prior work relevant to the proposed BDD package comes from two lines of research: sequential breadth-first BDD algorithms and parallel RAM-based BDD algorithms. The two approaches often overlap, resulting is parallel RAM-based breadth-first algorithms. However, there are no prior successful attempts at creating a parallel disk-based package in the literature.

Algorithms and representations for large BDDs stored in secondary memory have been the subject of research since the early 1990s. Ochi et al. [21] describe efficient algorithms for BDDs too large to fit in main memory on the commodity architectures available at that time. The algorithms are based on breadth-first search, thus avoiding random pointer chasing on disk. Shared Quasi-Reduced BDDs (SQBDDs) are used to minimize lookups at random locations on disk. These algorithms are inherently sequential, requiring the use of a local "operation-result-table". Random lookups to this local table avoid the creation of duplicate nodes.

Ashar and Cheong [2] build upon the ideas of [21] and improve the performance of BDD operations based on breadth-first search by removing the need for SQBDDs and using the most compact representation, the Reduced Ordered BDD (ROBDD). Duplicate avoidance is performed by lookups to local queues, which make the algorithm hard to parallelize.

A High Performance BDD Package which exploits the memory hierarchy is presented in [24]. The package builds upon the findings in [2], making the algorithms more general and more efficient. Even though [24] introduces concepts as "superscalarity" and "pipelining" for BDD algorithms, the presented package is still an inherently sequential one, relying on associative lookups to request queues.

Ranjan et al. [22] propose a BDD package based on breadth-first algorithms for a network of workstations, making possible the use of distributed memory for large BDD computations. However, their algorithms are sequential – the computation is carried out one workstation at a time, thus only taking advantage of the large amount of space that a network of workstations offers, without parallelizing the processing of data. Their approach is infeasible for efficient parallelization.

Stornetta and Brewer [28] propose a parallel BDD package for distributed fast-memory (RAM) based on depth-first algorithms, while providing a mechanism for efficient load balancing. However, their algorithms incur a large communication overhead. Any approach based on depth-first search would be infeasible for a parallel disk-based package, because the access pattern has a high degree of randomness. Milvang-Jensen and Hu [15] provide a BDD package building upon ideas in [22]. As opposed to [22], the package proposed in [15] is parallel – the workstations in a network all actively process data at the same time. However, since each workstation deals only with a sequence of consecutive levels, the workload can become very unbalanced, which would be unacceptable for a parallel disk-based BDD algorithm.

In 1997, Yang and O'Hallaron [29] propose a method for parallel construction of BDDs for shared memory multiprocessors and distributed shared memory (DSM) systems based on their technique named *partial breadth-first expansion*, having as starting point the depth-first/breadth-first hybrid approach in [1]. Their algorithms rely on very frequent synchronization of global data structures and on exclusive access to critical sections of the code which make this approach infeasible for any distributed-memory system. Also in 1997, Bianchi et al. [3] propose a MIMD approach to parallel BDDs. Their approach exhibits very good load balancing, but the communication becomes a bottleneck, making it infeasible to be adapted for parallel disks, because it does not delay and batch requests.

Other methods for manipulating very large BDDs are partitioning the BDD by means of partitioning the Boolean space using window functions [18] or by determining good splitting variables [6].

The newest result mentioned so far in this section is from 2001. Although there is substantial BDD-related work after 2001, little of it is concerned with providing fundamentally different ways of parallelizing BDD algorithms. Most use existing ideas and improve upon them or optimize the implementations. Most BDD-related research in the past decade was oriented towards better and faster algorithms for reachability analysis [9, 10, 23], compacting the representation of BDDs [27] or variable ordering [7, 17, 20].

There are many other methods for improving the efficiency of BDD processing. One such method, described by Mao and Wu [14], applies Wu's method to symbolic model checking.

2. BACKGROUND: BINARY DECISION DIAGRAMS

The conceptual idea behind Binary Decision Diagrams is quite simple. One wishes to represent an arbitrary Boolean function from k Boolean variables to a Boolean result. For example, "and", "or", and "xor" are three functions, each of which is a Boolean function on two Boolean variables. In the following description, the reader may wish to refer to Figure 1 for an example.

An *ordered binary decision diagram* (OBDD) fixes an ordering of the k Boolean variables, and then represents a binary decision tree with respect to that ordering. The tree has $k+1$ levels, with each of the first k levels corresponding to the next Boolean variable in the ordering. For each node at level i, the Boolean values of $x_1, \ldots, x_{i-1}$ are known, and the child along the left branch of a tree represents those same Boolean values, along with x_i being set to 0 or false. Similarly, the descendant of a node at level i along the right branch of a tree corresponds to setting x_i to 1 or true. For a node at level $k+1$, the Boolean values of $x_1, \ldots, x_k$ at that node are all determined by the unique path from the root of the tree to the given node. Next, a node at level $k+1$ is set either to false or to true, according to the result of the Boolean function being represented.

For example, the "and" function on two variables will have three levels: one root node at the first level, two nodes at the second level, and four leaf nodes at the third level. Three of the leaf nodes are set to false and one is set to true.

A *reduced ordered binary decision diagram* (ROBDD) is a lattice representing the same information as an OBDD, but with maximal sharing of nodes. More precisely, one can progressively convert an OBDD into an ROBDD as follows. First, identify all "true" nodes of the OBDD with a single "true" node of the ROBDD, and similarly for the "false" nodes. Thus, there are exactly two nodes at level $k+1$ in the ROBDD. One says that the nodes at level $k+1$ have been *reduced*.

Next, one iterates moving bottom up. Assume that all nodes at levels $i+1, i+2, \ldots, k$ of the OBDD have been reduced. Two nodes at level i are *reduced*, or identified with each other, if they have the same left child node and the same right child node. After all possible reductions at level i are completed, one continues to level $i-1$, and so on.

The structure described so far is in fact a *quasi-reduced BDD* (QBDD), which has no redundant nodes, and a child of a node can be either a node at the next level or a terminal. A fully reduced BDD (ROBDD) has one additional optimization: nodes that have identical left and right children (sometimes called *forwarding nodes*) are removed. So, children can be from any lower level.

Often, one refers to an ROBDD as simply a BDD for short. The advantage of a BDD is a compact representation. For example, the parity Boolean function is the Boolean function that returns true if an even number of the k input variables are true, and it returns false otherwise. A BDD representing the parity function will have exactly two distinct nodes for each level below level 1. Intuitively, one of the two nodes at level i can be thought of as indicating that the variables $x_1, \ldots, x_{i-1}$ have even parity, and the other node represents odd parity. Following the branch that sets x_i to true will lead from a node at level i to a node of the opposite parity at level $i+1$. Following the branch that sets x_i to false will lead to a node of the same parity at level $i+1$.

Finally, to take the logical "and" of two BDDs, one constructs a new BDD in top-down fashion using the Shannon expansion. If one wishes to form the "and" of two nodes at the same level (one from the first BDD and one from the second BDD), the result will be a new node whose left child will be the "and" of the two original left children, and the right child will be the “and” of the two original right children. Thus, to construct the “and” of two nodes requires that one recursively construct the “and” of the left children along with the “and” of the right children. The recursion stops when reaching the leaf nodes. The combinatorial explosion is reduced (but not necessarily eliminated) by using an implementation related to dynamic programming [4].

Logical “or” and “xor” of BDDs are implemented similarly to logical “and”. The generic algorithm for combining two BDDs is often called `apply` in the literature. Logical “not” of a BDD consists of exchanging the values of the two leaf nodes, “true” and “false”.

The QBDD is preferred over the ROBDD due to its more natural specification of node levels. In a QBDD, the *level* of a node is the number of edges that have to be traversed on any path from the root to the node. Note that the level definition is consistent for any non-terminal node – any path from the root to a node in a QBDD has the same length. We use the convention that the root of a BDD is at level 0. Level i corresponds to variable index i.

Figure 1 shows a comparison between OBDD, QBDD and ROBDD representations of the same Boolean formula.

3. A BRIEF OVERVIEW OF ROOMY

This section gives a brief overview of the parts of Roomy that are pertinent to the BDD algorithms in Section 4. Complete documentation and source code for Roomy can be found online [12]. Also published in the same proceedings as this paper is a tutorial on Roomy [13], which describes the Roomy programming model and gives example programs.

We chose to use Roomy for our implementation because it removes the need for the programmer to deal with various correctness and performance issues that arise when writing parallel disk-based applications.

3.1 Goals and Design of Roomy

Roomy is implemented as an open-source library for C/C++ that provides programmers with a small number of simple data structures (arrays, unordered lists, and hash tables) and associated operations. Roomy data structures are transparently distributed across many disks, and the operations on these data structures are transparently parallelized across the many compute nodes of a cluster. All aspects of the parallelism and remote I/O are hidden within the Roomy library.

The primary goals of Roomy are:

1. to provide the most general programming model possible that still biases the application programmer toward high performance for the underlying parallel disk-based computation.

2. the use of full parallelism; providing not only the use of parallel disks (e.g., as in RAID), but also parallel processing.

3. to allow for a wide range of architectures, for example: a single shared-memory machine with one or more disks; a cluster of machines with locally attached disks; or a compute cluster with storage area network (SAN).

The overall design of Roomy has four layers: foundation; programming interface; algorithm library; and applications. Figure 2 shows the relationship between each of these layers, along with examples of the components contained within each layer.

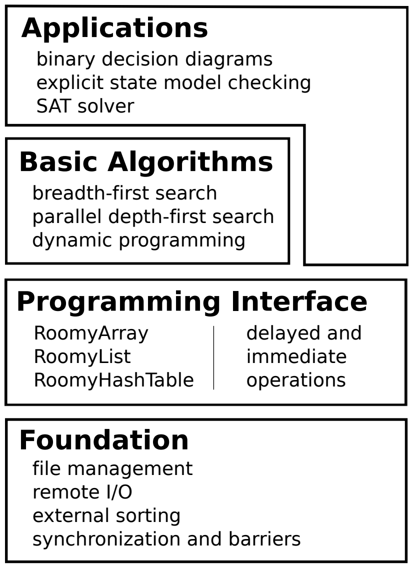


Figure 2: The layered design of Roomy.

3.2 Roomy-hashtable

Because the Roomy-hashtable is the primary data structure used in the parallel disk-based BDD algorithms in Section 4, it is used to illustrate the various operations provided by the Roomy data structures. Operations for the Roomy-array and Roomy-list are similar in nature.

A *Roomy-hashtable* is a variable sized container which maintains a mapping between user-defined *keys* and *values*. Those elements can be built in data types (such as integers), or user defined structures of arbitrary size. A Roomy-hashtable can be of any size, up to the limits of available disk space, and will grow as needed as elements are inserted.

Each Roomy-hashtable is stored as a number of RAM-sized *subtables*, distributed across the disks of a cluster (or the disks of a SAN). Delayed operations are buffered to disk and co-located with the subtable they reference. Once many delayed operations are collected, each subtable can be processed independently to complete the delayed operations, avoiding costly random access patterns.

Below are the operations provided by a Roomy-hashtable, categorized by whether they are delayed or immediate.

Roomy-hashtable delayed operations..

`insert`: insert a given key/value pair, replacing any existing pair with the same key

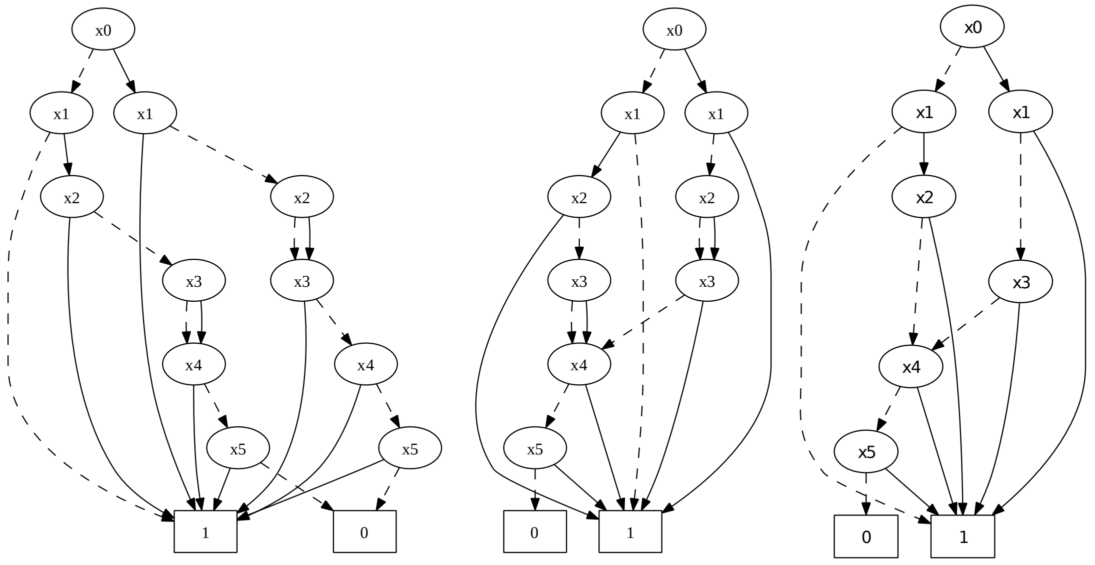
`access`: apply a user defined function to the key/value pair with the given key, if it exists

`remove`: remove the key/value pair with the given key, if it exists

Roomy-hashtable immediate operations..

`size`: return the number of key/value pairs stored in the hashtable

`map`: applies a user defined function to each key/value pair in the hashtable



(a) OBDD representation (b) QBDD representation (c) ROBDD representation

Figure 1: OBDD, QBDD and ROBDD representations for Boolean formula $(x_0 \vee \neg x_1 \vee x_2 \vee x_4 \vee x_5) \wedge (\neg x_0 \vee x_3 \vee x_1 \vee x_4 \vee x_5)$. Solid lines represent the high (true) branch, dashed lines the low (false) branch. The OBDD is the least compact representation. Combining identical nodes (for variables x_4 and x_5 in this case) will create a QBDD. The QBDD can be further reduced to an ROBDD by removing a node if it has identical high and low pointers (for variables x_2 and x_3 in this case).

`reduce`: applies a user defined function to each key/value pair in the hashtable and returns a value (e.g. the sum of all of the values)

`predicateCount`: return the number of key/value pairs in the hashtable that satisfy the given predicate

`sync`: perform all outstanding delayed `insert`, `access`, and `remove` operations

4. PARALLEL DISK-BASED ALGORITHMS FOR BDDS

The proposed Roomy-based parallel disk-based BDD package uses the SQBDD (shared QBDD) representation described in [21]. This package does not support BDDs that are shared among multiple Boolean expressions (although this feature can easily be implemented in the package), thus making the representation a quasi-reduced BDD (QBDD). Although [2] describes cases in which the ROBDD representation can be a few times more compact than the corresponding (S)QBDD representation, this is not the case for the problems considered here. The focus of the experimental results (see Section 5) is on the solutions to combinatorial problems (the N-queens problem and 3-D tic-tac-toe) in which the inherent regularities and symmetries of the problem lead to QBDDs that are only at most 15 % larger than the corresponding ROBDD (the percentage was observed experimentally). Since most operations in Roomy are delayed and processed in batches, the locality advantage of a the QBDD representation is important (a node in a QBDD can only have as children the nodes at the immediate next level). This is especially true in cases where the QBDD representation is not much larger than the ROBDD representation, as is the case here.

Since the QBDDs for the considered problems are close in size to their corresponding ROBDDs, it means that one can solve problems one or two orders of magnitude larger using disk-based methods than by using a RAM-based alternative.

The rest of this section presents a brief implementation description of our package, followed by a high-level description of the main operations: `apply`, `any-SAT` and `SAT-count`.

4.1 Implementation Description

Each BDD is represented in a quasi-reduced form. This means that, when a node at level i has a child at level $j > i + 1$, there are $j - i - 1$ padding nodes inserted in the BDD, one at each level between $i + 1$ and $j - 1$, thus creating an explicit path from the node at level i to its child at level j. These padding nodes would be considered redundant nodes in an ROBDD implementation, and thus, would not exist. However, a QBDD needs such redundant nodes to maintain the invariant that each node at a certain level only has children at the immediate next level.

Our implementation of a QBDD maintains a data structure that contains:

- d, the depth of the BDD.
- *Nodes*, an array of d Roomy-hashtables. $Nodes[i]$ is a Roomy-hashtable that contains all the nodes at level i of the BDD in key-value form. The key is the BDD-unique node id and the value is a pair of ids *(low(id), high(id))*, which reference the children of the node. Each of the children is either at level $i + 1$ or is a terminal node.

This representation allows the *delayed* lookup of a node in the BDD by providing only its unique id. Since each BDD level has its own Roomy-hashtable, when processing a certain level i we only need to inspect $Nodes[i]$ (*the current nodes*), $Nodes[i + 1]$ (*the child nodes*) and, in some cases $Nodes[i - 1]$ (*the parents*). Inspecting the entire BDD is not necessary. This is important for very large BDDs, in which even the nodes at a single level can overflow the aggregate RAM of a cluster.

4.2 The APPLY Algorithm

The foundation that most BDD algorithms are built upon is the `apply` algorithm. `apply` applies a Boolean operation (like `or`, `and`, `xor`, `implication`, a.s.o) to two BDDs. If BDDs A and B represent Boolean expressions f_A and f_B respectively, then $AB = op(A, B)$ is the BDD which represents the Boolean expression $op(f_A, f_B)$. The traditional RAM-based `apply` algorithm performs this operation by employing a depth-first algorithm, as explained in [5]. A memoization table in which already computed results are stored increases the performance of the algorithm from exponential to quadratic in the number of input nodes. The pseudo-code is presented in Algorithm 1.

Algorithm 1 RAM-based depth-first `apply`

```
Input: BDD $A$ with root-node $n_1$, BDD $B$ with root-node $n_2$, Boolean operator $op$
Output: $n$, the root node of a BDD representing $op(A, B)$
  Init $M$ (memoization table for new nodes)
  if $M(n_1, n_2)$ was already set then
    return $M(n_1, n_2)$
  if $n_1 \in \{0, 1\}$ AND $n_2 \in \{0, 1\}$ then
    $n = op(n_1, n_2)$
  else if $var(n_1) = var(n_2)$ then
    $n \leftarrow new\,Node(var(n_1), apply(low(n_1), low(n_2), op),$
                $apply(high(n_1), high(n_2), op))$
  else if $var(n_1) < var(n_2)$ then
    $n \leftarrow new\,Node(var(n_1), apply(low(n_1), n_2, op),$
                $apply(high(n_1), n_2, op))$
  else
    $n \leftarrow new\,Node(var(n_2), apply(n_1, low(n_2), op),$
                $apply(n_1, high(n_2), op))$
    set $M(n_1, n_2) \leftarrow n$
  return $n$
```

To create a version of `apply` applicable for data stored in secondary memory, [21] converts the requirements on the implicit process stack in Algorithm 1 into an explicit requirement queue. This is the main idea behind converting a DFS algorithm into a BFS one. Our parallel disk-based package uses the same framework to create an efficient parallel disk-based `apply`. The structure used is a *parallel queue*, implemented on top of a Roomy-hashtable. A parallel queue relaxes the condition that all requirements must be processed one by one to a condition that all requirements at a certain level in the BFS must be processed before any requirement at the next level. In this way, parallel processing of data at each BFS level is enabled.

An important feature that Roomy provides for any application is load balancing. For the BDD package, load balancing means that the BDD nodes are distributed evenly across the disks of a cluster. This is achieved by assigning each BDD node to a random disk. Hence, a BDD node is not necessarily stored near its children, and accessing the children of a node in an immediate manner would lead to long delays due to network and disk latency. Using delayed batched operations, as Roomy does (see Section 3), is the only acceptable solution if the very large BDDs are being treated as monolithic. Other approaches decompose the large BDD into smaller BDDs and then solve each of them separately on various compute nodes in the cluster [6]. These two solutions are not mutually exclusive. A decomposition approach can be adapted to our BDD package as well. For the rest of the paper, only the case of very large monolithic BDDs which are not decomposed into smaller BDDs is considered.

The parallel disk-based `apply` algorithm consists of two phases: an `expand` phase and a `reduce` phase, described in Algorithms 2 and 3, respectively. The `expand` phase creates a valid but larger than necessary QBDD. It can be reduced to a smaller size, because it contains non-unique nodes. Non-unique nodes are nodes at the same level i which represent identical sub-BDDs. However, the fact that they are duplicates cannot be determined in the `expand` phase. `expand` is a top-down breadth-first scan. The purpose of the `reduce` phase is to detect the duplicates at each level and keep only one representative of each duplication class. `reduce` is a bottom-up scan of the BDD returned by `expand`.

Algorithm 2 Parallel disk-based `expand`

```
Input: QBDDs $A$, $B$ and Boolean operator $op$
Output: QBDD $AB = op(A, B)$
 1: Initialize $Q$ with the entry representing the root of $AB$:
    $(root\text{–}id_A, root\text{-}id_B) \rightarrow (new\,Id(), N/A)$
 2: $level \leftarrow 0$, $Q' \leftarrow \emptyset$
 3: while $level \leq max(depth(A), depth(B))$ AND $Q \neq \emptyset$ do
 4:   Remove entries with duplicate keys from $Q$ and update their parent nodes (extracted from the entry's value) to point to the id found in the value of the representative of the duplicate class.
 5:   for each entry $(id_A, id_B) \rightarrow (id_{AB}, parent\text{–}id_{AB})$ in $Q$ do
 6:     Perform delayed lookup of the child nodes of
        $id_A$ : $low(id_A)$ and $high(id_A)$ and
        of $id_B$ : $low(id_B)$ and $high(id_B)$
 7:     $id' \leftarrow new\,Id()$
        $id'' \leftarrow new\,Id()$
        Create two entries in $Q'$:
        $(low(id_A), low(id_B)) \rightarrow (id', id_{AB})$
        and, respectively,
        $(high(id_A), high(id_B)) \rightarrow (id'', id_{AB})$
 8:     Insert node $id_{AB} \rightarrow (id', id'')$ in $Nodes[level]$
 9:   $level \leftarrow level + 1$
10:   $Q \leftarrow Q'$, $Q' \leftarrow \emptyset$
```

Notation.

The notation $k \rightarrow v$ is used to represent a key-value entry in a Roomy-hashtable.

All *id*s are Roomy-unique integers that can be passed back to Roomy as keys in a Roomy-hashtable.

A *forwarding node* is a temporary node created during the `reduce` phase. Such a node acts as a pointer to a terminal node, meaning that all parents pointing to the node should actually point to the terminal node. When it is no longer needed, it is removed.

If *node* is a forwarding node in the reduce phase, the notation $fwd(id_{node})$ is used to represent the id of the permanent node that *node* points to.

Implementation aspects.

The `expand` algorithm uses per-level distributed disk-based requirement queues Q and Q', which are implemented as

Roomy-hashtables. All entries in Q are of the form $(id_A, id_B) \rightarrow (id_{AB}, parent\text{–}id_{AB})$.

The nodes at a certain level in AB are stored in $Nodes[i]$. In AB, $id_{AB} = op(id_A, id_B)$. id_{AB} is a low or high child of $parent\text{–}id_{AB}$, a node at the previous level in AB.

`expand` creates a partially-reduced OBDD, which `reduce` converts into a QBDD.

Note that our `expand` algorithm differs from the `expand` presented in [21] in the following way: in our case all data structures are distributed and stored on the parallel disks of a cluster; there is no need for any QBDD level to fit even in the distributed RAM at any time; all the operations that involve reading or writing remote data are delayed and batched; in line 7, unique ids for the left and right child of a node are created in advance (in the next iteration it will be found that some are duplicates), while in [21] a per-level memoization table named "operation-result-table" is used. When duplicates are found, the parent nodes of the duplicates are updated to point to the child representative of the duplicate class. A per-level memoization table would not work for parallel `expand` algorithms because all reads are delayed, and so one cannot immediately check whether a certain node has already been computed.

Algorithm 3 Parallel disk-based `reduce`

```
Input: a QBDD $AB$, returned by expand
Output: a QBDD $AB'$, resulted from $AB$ by eliminating
    non-unique nodes
 1: $level \leftarrow depth(AB)$
 2: while $level \geq 1$ do
 3:   for each node entry $(id) \rightarrow (low(id), high(id))$ in
      $Nodes[level]$ do
 4:     if $low(id)$ is a forwarding node then
 5:       $low(id) \leftarrow fwd(low(id))$
 6:     if $high(id)$ is a forwarding node then
 7:       $high(id) \leftarrow fwd(high(id))$
 8:     if $low(id) = high(id) = T$ (terminal node 0 or 1)
        then
 9:       Make this a forwarding node to $T$.
10:   Remove non-unique nodes (duplicates) at the current
      level. Update the parents of the duplicate nodes to
      point to the representative of the duplicate class. This
      is duplicate detection.
11:   $level \leftarrow level + 1$
```

Satisfying Assignments.

The `any-SAT` algorithm simply follows any path in the QBDD that ends in the terminal 1. When a high child is followed, the level's variable is set to 1. When a low child is followed, the level's variable is set to 0. When terminal 1 is reached, the assigned values of all the variables are listed.

The `SAT-count` algorithm is implemented with a top-down breadth-first scan of the QBDD. Each node in the QBDD has a counter associated with it. Initially, the root of the QBDD has its counter set to 1 and all other counters are set to 0. At each level in the breadth-first scan, the counter of each node is added to each of its children's counters. When a child of a node at level i is the terminal node 1, update a global counter by adding the node's local counter multiplied by 2^m, where $m = depth(QBDD) - i - 1$, to it. All updates are delayed and batched, to maintain the parallelism of the scan. After the scan finishes and the updates are processed, the global counter contains the number of satisfying assignments of the QBDD.

5. EXPERIMENTAL RESULTS

Hardware and Software Configurations.

The Roomy-based parallel disk BDD package was experimentally compared with BuDDy [8], a popular open-source BDD package written in C/C++. BuDDy was used with two different computer architectures.

- **Server:** with one dual-core 2.6 GHz AMD Opteron processor, 8 GB of RAM, running HP Linux XC 3.1, and compiling code with Intel ICC 11.0.
- **Large Shared Memory:** with four quad-core 1.8 GHz AMD Opteron processors, 128 GB of RAM, running Ubuntu SMP Linux 2.6.31-16-server, and compiling code with GCC 4.4.1.

While BuDDy could make use of all of the RAM on the machine with 8 GB, it was only able to use approximately 56 GB out of 128 GB in the other case, because the array used to store the nodes of the BDDs was limited to using a 32-bit index.

For Roomy, the architecture was a cluster of the 8 GB machines mentioned above, using a Quadrics QsNet II interconnect. Each machine had a 40 GB partition on the local disk for storage. This provides an upper bound for the total storage used at any time. For smaller problem instances, 8 machines of the cluster were used, with larger examples using 32 or 64 machines.

Test Problems.

Four cases were tested: BuDDy using a maximum of 1, 8, or 56 GB of RAM; and Roomy. These four cases were tested on two combinatorial problems:

- **N-queens:** counting the number of unique solutions to the N-queens problem.
- **3-dimensional tic-tac-toe:** counting the number of possible tying configurations when placing N X's and $64 - N$ O's in a 4×4×4 3D tic-tac-toe board.

In both cases, the experiments were run with increasing values of N. For N-queens, this increases the problem difficulty by increasing the dimension of the board. For 3D tic-tac-toe, this increases the difficulty by increasing the number of possible positions to consider (with maximum difficulty at 32 out of 64 positions for each X and O).

The rest of this section describes the details of each problem, how the solutions are represented as Boolean formulas, and gives experimental results.

5.1 Counting N-Queens Solutions

Problem Definition.

The N-queens problem considered here is: how many different ways can N queens be placed on an N×N chess board such that no two queens are attacking each other? The size of the state space is $N!$, corresponding to an algorithm that successively places one queen in each row, and for row k chooses one of the remaining $N - k$ columns not containing a queen.

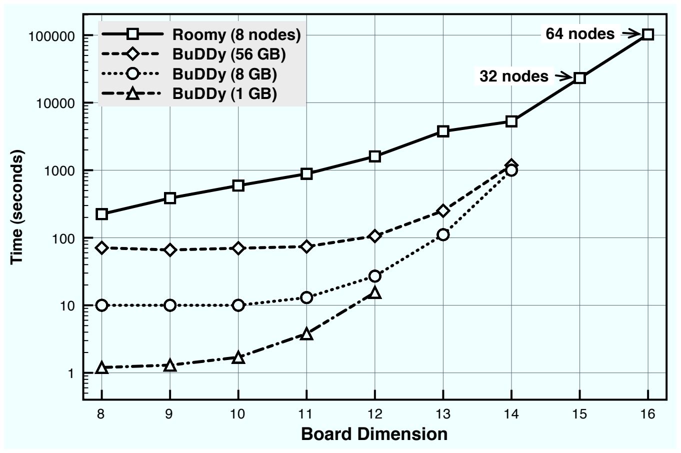


Figure 3: Running times for the N-queens problem.

The current record for the largest N solved is 26, achieved using massively-parallel, FPGA-based methods [26]. The more general BDD approach described here is not meant to directly compete with this result, but to serve as a method for comparing traditional RAM-based BDD packages and our parallel-disk based approach. The N-queens problem is often used as an illustration in introductions to the subject, and a solution to the problem comes with BuDDy as an example application.

The following defines a Boolean formula that represents all of the solutions to a given instance of the N-queens problem.

First, define N^2 variables of the form $x_{i,j}$, where $x_{i,j}$ is true iff a queen is placed on the square at row i column j. Then, define $S_{i,j}$, which is true iff there is a queen on square i, j and not on any square attacked from that position.

$$S_{i,j} = x_{i,j} \wedge \neg x_{i_1,j_1} \wedge \neg x_{i_2,j_2} \wedge \ldots$$

The constraint that row i must have exactly one queen is

$$R_i = S_{i,1} \vee S_{i,2} \vee \ldots \vee S_{i,N}$$

And finally, the board has one queen in each row.

$$B = R_1 \wedge R_2 \wedge \ldots \wedge R_N$$

The number of solutions is computed by counting the number of satisfying assignments of B.

Experimental Results.

Figure 3 shows the running times for the N-queens problem using BuDDy with a maximum of 1, 8, or 56 GB of RAM, and using our BDD package based on Roomy.

BuDDy with 1 GB is able to solve up to $N = 12$, which finished in 15 seconds. BuDDy with 8 GB is able to solve up to $N = 14$, which finished in 16.7 minutes. For higher N, BuDDy runs out of RAM. This demonstrates how quickly BDD computations can exceed available memory, and the need for methods providing additional space.

Even when given up to 56 GB, BuDDy can not complete $N = 15$. For smaller cases, the version using more memory is slower because the time for memory management dominates the relatively small computation time.

The Roomy-based package is able to solve up to $N = 16$, which finished in 28.4 hours. The figure shows the running times for $N \leq 14$ using 8 compute nodes, $N = 15$ using 32 nodes, and $N = 16$ using 64 nodes. The results demonstrate that parallel-disk based methods can extend the use of BDDs to problem spaces several orders of magnitude larger than methods using RAM alone.

Table 1: Sizes of largest and final BDDs, and number of solutions, for the N-queens problem (* indicates that BuDDy exceeded 56 GB of RAM without completing).

N	Largest Buddy BDD	Largest Roomy BDD	Ratio
8	10705	11549	1.08
9	44110	50383	1.14
10	212596	234650	1.10
11	1027599	1105006	1.08
12	4938578	5250309	1.06
13	26724679	29370954	1.10
14	153283605	165030036	1.08
15	*	917859646	–
16	*	3380874259	–

N	Final Buddy BDD	Final Roomy BDD	Ratio	# of Solutions
8	2451	2451	1.00	92
9	9557	9557	1.00	352
10	25945	25945	1.00	724
11	94822	94822	1.00	2680
12	435170	435170	1.00	14200
13	2044394	2044394	1.00	73712
14	9572418	9572418	1.00	365596
15	*	51889029	–	2279184
16	*	292364273	–	14772512

For the cases using BuDDy, the versions using more RAM had a time penalty due to the one-time cost of initializing the larger data structures. For Roomy, the time penalties for the smaller cases are primarily due to synchronization of the parallel processes, which is amortized for the longer running examples.

For the computations using Roomy, BuDDy was first used to compute each of the N row constraints, R_i. Then, Roomy was used to combine these into the final BDD B. This was done to avoid performing many operations on very small BDDs with Roomy, which would cause many small, inefficient disk operations.

Table 1 shows the sizes of the largest BDD, the final BDD, and the number of solutions for $8 \leq N \leq 16$. In this case, the largest BDD produced by Roomy was approximately 22 times larger than the largest BDD produced by BuDDy (for instances that were solved). Table 1 also shows that, for this problem, the additional nodes required by the use of quasi-reduced BDDs, instead of the traditional fully reduced BDDs, increase BDD size by at most 14 percent. We consider this cost acceptable given the increase in locality QBDDs provide the computation.

5.2 Counting Ties in 4×4×4 3D Tic-Tac-Toe

Problem Definition.

This problem deals with a generalization of the traditional game of tic-tac-toe: two players, X and O, take turns marking spaces in a 3×3 grid, attempting to place three marks in a given row, column, or diagonal. In this case, we consider a 3-dimensional 4×4×4 grid.

The question considered here is: how many possible tie games are there when X marks N spaces, with O filling the remaining $64 - N$? As N is increased from 1 to 32, both

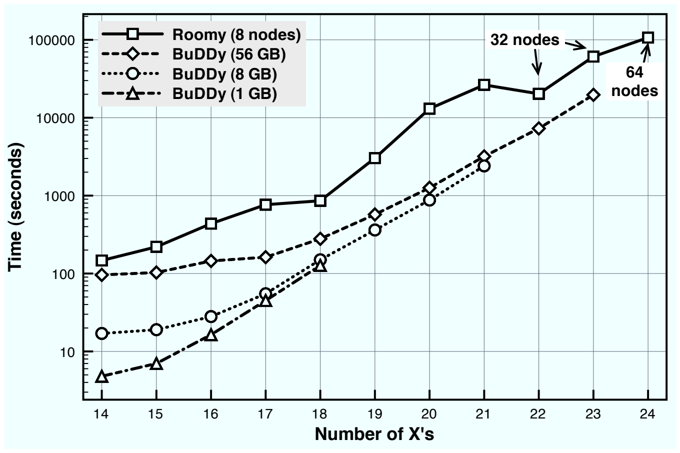


Figure 4: Running times for the 3D tic-tac-toe problem.

the number of possible arrangements and the difficulty of the problem increase. To the best of our knowledge, this problem has not been solved before.

The Boolean formula representing the solution uses 64 variables of the form $x_{i,j,k}$, which are true iff the corresponding space is marked by X. First, a BDD representing placements that have exactly N out of 64 spaces marked with X is constructed. Then, BDD constraints for each possible sequence of four spaces in a row are successively added. The constraints are of the form: $\neg$(all four spaces are X) $\wedge$ (at least one space has an X).

The number of possible ties is computed by counting the number of satisfying assignments of the final BDD.

Experimental Results.

Figure 4 shows the running times for the 3D tic-tac-toe problem using BuDDy with a maximum of 1, 8, or 56 GB of RAM, and using the Roomy-based package. As in the N-queens problem, BuDDy was used to compute several smaller BDDs, which were then combined into the final BDD using Roomy.

BuDDy with 1 GB of RAM is able to solve up to $N = 18$, finishing in 127 seconds. BuDDy with 8 GB of RAM is able to solve up to $N = 21$, finishing in 40 minutes. Unlike the N-queens problem, increasing the available RAM from 8 to 56 GB increases the number of cases that can be solved, up to $N = 23$, which finished in 5.5 hours. Roomy was able to solve up to $N = 24$, finishing in under 30 hours using 64 nodes.

Table 2 shows the sizes of the largest BDD, the final BDD, and the number of solutions for $14 \leq N \leq 24$. The smallest number of X's that need to be placed to force a tie is 20, yielding 304 tying arrangements. The number of possible ties then increases rapidly, up to over 5 billion for $N = 24$.

For the 3D tic-tac-toe problem, all of the QBDDs used by Roomy are exactly the same size as the ROBDDs used by BuDDy. So, like the N-queens problem, using the possibly less compact representation is not an issue here.

Table 2: Sizes of largest and final BDDs, and number of tie games, for the 3D tic-tac-toe problem (* indicates that BuDDy exceeded 56 GB of RAM without completing).

X's	Largest Buddy BDD	Largest Roomy BDD	Ratio
14	389251	389251	1.00
15	671000	671000	1.00
16	1350821	1350821	1.00
17	4378636	4378636	1.00
18	11619376	11619376	1.00
19	24742614	24742614	1.00
20	42985943	42985943	1.00
21	113026291	113026291	1.00
22	383658471	383658471	1.00
23	988619402	988619402	1.00
24	*	2003691139	–

X's	Final Buddy BDD	Final Roomy BDD	Ratio	# of Ties
14	1	1	1.00	0
15	1	1	1.00	0
16	1	1	1.00	0
17	1	1	1.00	0
18	1	1	1.00	0
19	1	1	1.00	0
20	8179	8179	1.00	304
21	433682	433682	1.00	136288
22	6560562	6560562	1.00	9734400
23	60063441	60063441	1.00	296106640
24	*	373236946	–	5000129244

6. CONCLUSIONS AND FUTURE WORK

This work provides a parallel disk-based BDD package whose effectiveness is demonstrated by solving large combinatorial problems. Those problems are typical for a broad class of mathematical problems whose large state space contains a significant degree of randomness.

Future work will tackle the verification of large industrial circuits. A parallel disk-based ROBDD representation will be provided, together with modified versions of the BFS algorithms that now use QBDDs. It is expected that this future implementation will perform better for industrial circuits, while we believe that QBDDs will still be more suitable to combinatorial problems. These expectations coincide with the motivation of using ROBDDs in [2] and [24].

Extensive research shows that, for many industrial problems, dynamic variable reordering can yield significant space savings, which get translated into time savings. Providing a parallel disk-based method for dynamic variable reordering is also part of the future work.

7. REFERENCES

[1] Y. an Chen, B. Yang, and R. E. Bryant. Breadth-first with depth-first BDD construction: A hybrid approach. Technical report, 1997.

[2] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided design*, pages 622–627, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] F. Bianchi, F. Corno, M. Rebaudengo, M. S. Reorda, and R. Ansaloni. Boolean function manipulation on a parallel system using BDDs. In *HPCN Europe '97:*

Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, pages 916–928, London, UK, 1997. Springer-Verlag.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, New York, NY, USA, 1990. ACM.

[5] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[6] G. Cabodi, P. Camurati, and S. Quer. Improving the efficiency of BDD-based operators by means of partitioning. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(5):545–556, 1999.

[7] M. Carbin. Learning effective BDD variable orders for BDD-based program analysis, 2006.

[8] H. Cohen. BuDDy: A binary decision diagram library, 2004. `http://sourceforge.net/projects/buddy/`.

[9] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *In CHARME*, pages 129–145. Springer, 2005.

[10] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. A scalable parallel algorithm for reachability analysis of very large circuits. *Form. Methods Syst. Des.*, 21(3):317–338, 2002.

[11] S. Kimura and E. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. of IEEE International Conference onComputer Design: VLSI in Computers and Processors, 1990 (ICCD '90)*, pages 220 –223, sep 1990.

[12] D. Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. `http://roomy.sourceforge.net/`.

[13] D. Kunkle. Roomy: A system for space limited computations. In *Proc. of Parallel Symbolic Computation (PASCO '10)*. ACM Press, 2010.

[14] W. Mao and J. Wu. Application of wu's method to symbolic model checking. In *ISSAC '05: Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, pages 237–244, New York, NY, USA, 2005. ACM.

[15] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *FMCAD '98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 501–507, London, UK, 1998. Springer-Verlag.

[16] S. Minato and S. Ishihara. Streaming BDD manipulation for large-scale combinatorial problems. In *DATE '01: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 702–707, Piscataway, NJ, USA, 2001. IEEE Press.

[17] H. Moeinzadeh, M. Mohammadi, H. Pazhoumand-dar, A. Mehrbakhsh, N. Kheibar, and N. Mozayani. Evolutionary-reduced ordered binary decision diagram. In *AMS '09: Proceedings of the 2009 Third Asia International Conference on Modelling & Simulation*, pages 142–145, Washington, DC, USA, 2009. IEEE Computer Society.

[18] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs — a compact, canonical and efficiently manipulable representation for Boolean functions. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, pages 547–554, Washington, DC, USA, 1996. IEEE Computer Society.

[19] Z. Nevo and M. Farkash. Distributed dynamic BDD reordering. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 223–228, New York, NY, USA, 2006. ACM.

[20] Z. Nevo and M. Farkash. Distributed dynamic BDD reordering. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 223–228, New York, NY, USA, 2006. ACM.

[21] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided design*, pages 48–55, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[22] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstation. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 358–364, Washington, DC, USA, 1996. IEEE Computer Society.

[23] D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill, and E. A. Emerson. Multi-threaded reachability. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 467–470, New York, NY, USA, 2005. ACM.

[24] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 635–640, New York, NY, USA, 1996. ACM.

[25] A. Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: challenges and solutions. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 344, Washington, DC, USA, 1997. IEEE Computer Society.

[26] R. G. Spallek, T. B. Preußer, and B. Nägel. Queens@TUD, 2009. `http://queens.inf.tu-dresden.de/`.

[27] S. Stergios and J. Jawahar. Novel applications of a compact binary decision diagram library to important industrial problems. *Fujitsu scientific and technical journal*, 46(1):111–119, 2010.

[28] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *DAC '96: Proceedings of the 33rd Annual Design Automation Conference*, pages 641–644, New York, NY, USA, 1996. ACM.

[29] B. Yang and D. R. O'Hallaron. Parallel breadth-first BDD construction. In *PPOPP '97: Proceedings of the sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 145–156, New York, NY, USA, 1997. ACM.

Parallel arithmetic encryption for high-bandwidth communications on multicore/GPGPU platforms*

Ludovic Jacquin
INRIA
Planète team, France
ludovic.jacquin@inria.fr

Vincent Roca
INRIA
Planète team, France
vincent.roca@inria.fr

Jean-Louis Roch [*] [†] [‡]
Laboratoire d'Informatique de Grenoble (LIG), France
jean-louis.roch@imag.fr

Mohamed Al Ali
Laboratoire d'Informatique de Grenoble (LIG), France
mohamed.alali@imag.fr

ABSTRACT

In this work we study the feasibility of high-bandwidth, secure communications on generic machines equipped with the latest CPUs and General-Purpose Graphical Processing Units (GPGPU). We first analyze the suitability of current Nehalem CPU architectures. We show in particular that high performance CPUs are not sufficient by themselves to reach our performance objectives, and that encryption is the main bottleneck. Therefore we also consider the use of GPGPU, and more particularly we measure the bandwidth of the AES ciphering on CUDA. These tests lead us to the conclusion that finding an appropriate solution is extremely difficult.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: General; C.4 [**Performances of Systems**]

Keywords

High-performance communications, encryption, parallelism, GPU/GPGPU

1. INTRODUCTION

During the past few years, communications have experienced tremendous throughput increases since 10 Gb/s Network Interfaces Controllers (NIC) are now common on high performance machines. This situation authorizes the deployment of large scale clusters for distributed computing. In this context, security is more and more often a requirement and communications between sites must be encrypted to avoid critical information leaks. This is the case for instance with medical applications that require the processing power of computational grids but manipulate highly confidential data from patients for instance [17]. This is also the case for inter-site secure communications, where the aggregated traffic can reach high throughput during database synchronization or remote backup procedures for instance. The Moore law has long been sufficient to keep a reasonable equilibrium between the available processing power and the physical link throughput. However, this is no longer the case with transmission throughput in the order of 10 Gb/s when ciphering is required. Indeed, at 10 Gb/s, the available processing time to encrypt or decrypt data is in the order of a few CPU cycles only.

This work addresses the feasibility of achieving high-bandwidth secured communications using generic off-the-shelf components, such as multicore CPUs and General-Purpose Graphical Processing Units (GPGPU). Based on our detailed experimental analysis, we conclude that high-bandwidth secure networking does require high-speed arithmetic facilities.

This article is structured as follows. Section 2 gives an overview of the state of the art in the domain, both from the hardware and software points of view. Section 3 describes the benchmark architecture on which the chosen proof-of-concept tests are performed. In Section 4, we inspect the algorithms and protocols used to secure communications, as well as their parallelization. Then the results of our experimentations are presented, on multicore architectures in Section 5, and on GP-GPU in section 6. Finally, Section 7 presents the conclusions from this study and the perspectives.

2. RELATED WORKS

To our knowledge, no reference that couples together high-bandwidth and parallel encryption exists nowadays. But there are some work that has been done in fields near our center of interest.

*This work is partially supported by French the French Ministère de l'Économie, de l'Industrie, et de l'Emploi and the Global competitive cluster Minalogic, project SHIVA no 09-2-93-0473.

[*]INRIA
[†]Grenoble Université
[‡]Institut Nation Polytechnique de Grenoble (INPG).

Specialized Hardware.

Solutions exist for encryption and high-bandwidth networking, but they are based on specialized hardware provided by companies such as Cisco [2] or Cavium [1]. The main disadvantages of these solutions are:

- scalability: for example a Cisco encryption board upper limit is 2,5 Gb/s, and four units are needed to reach 10 Gb/s.
- upgradability: once the hardware is installed, it is no more possible to change the chips.
- recyclability: in the future, when we will be deploying about 100+ Gb/s links, 10 Gb/s hardware will become obsolete and useless.
- price: such solutions cost 10 times more than an average server.

Therefore we chose not to consider this approach in our work.

High-bandwidth networking.

Previous works focused on routing and have explored the capabilities of generic servers to serve as high bandwidth routers. When the need for security is considered, the experiments carried out are rather limited. Yet their two main conclusions are that: (1) 10 Gb/s routing is possible, although it uses all the CPU resources [15], and (2) that when using IPsec and AES 128 bits encryption, they only achieve 1,5 to 4,5 Gb/s transmission speeds (depending on the packet size) [16].

Cipher parallelism.

On CPU, Roche et al [14] propose to use a block cipher in counter chaining mode, a mode that is well-suited for parallelization in addition to feature a high security level. The authors use a very effective method based on work-stealing and loop rescheduling for DES encryption. Using this method and when considering only in memory operations, they obtain a near-optimal speedup on multicore systems. However, when considering network I/O (our case), the speedup is significantly reduced, by up to 50%.

On GPUs, Andrea Di Biagio et al [13] present two implementations of the AES algorithm that are parallelized for GPUs. They propose a fine-grained approach for parallelizing the inner operations of the AES round, and a coarse-grained approach that focuses on the parallelism outside the round operations an data blocks. Thanks to these techniques, they obtain great speed-ups when ciphering large files. However this is not directly applicable to our case since we need to consider smaller sizes (we follow a per-packet approach).

3. BENCHMARK ARCHITECTURE

In this section, we describe the server we used for our experiments and its basic performance.

Processor architecture.

The Intel Nehalem architecture [4] introduces parallelism mechanisms at the hardware level. Those mechanisms concentrate on two aspects of the architecture: cache/memory accesses and I/O connections.

As far as memory is concerned, each CPU chip introduces a new level of cache (L3) that is shared by the four cores,

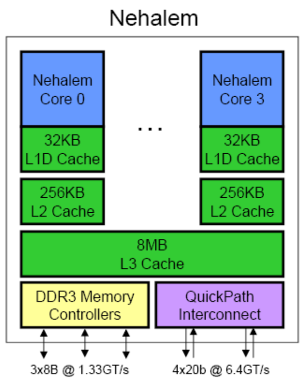


Figure 1: Nehalem cache architecture.

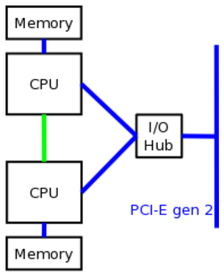


Figure 2: New Nehalem motherboard architecture.

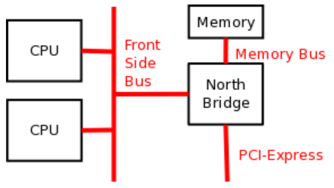


Figure 3: Old motherboard architecture.

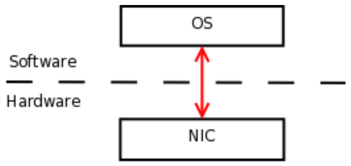


Figure 4: Old Network Interface Cards (NIC).

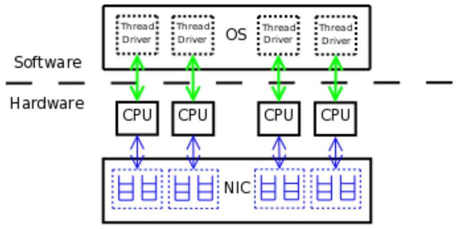


Figure 5: New Network Interface Cards (NIC), with multi-queue.

and two physical buses (RAM and QuickPath) (figure 1). To each CPU chip is associated a privileged memory bank (connected to the CPU chip RAM bus) that is accessed directly by the associated CPU, and that can still be accessed by remote CPU chips through a CPU-to-CPU interconnection (figure 2). This new memory hierarchy enhances parallelism since each CPU can now access its own memory bank independently, without creating any contention with respect to other CPUs. As far as I/O communications are concerned, Intel also removed the classic bottleneck by replacing the old shared-bus architecture (e.g. the Front Side Bus, FSB, of figure 3) by point-to-point connections between each CPU and a I/O hub (figure 2).

Network Interface Card (NIC).

In traditional NIC architectures, the NIC driver is the only entity capable of accessing the NIC (figure 4). On the opposite, recent NIC also include parallelism mechanisms at the hardware level, and more precisely new NIC define multiple reception and transmission queues (figure 5). Thanks to this change, the traditional bottleneck at the hardware/software edge is removed. Even though it was primarily designed for virtualized server [5], the new NIC architecture is useful in our use-case because they can help spreading the network charge over different cores, over different OS threads.

First performance evaluation.

Using a Nehalem based server, with multi-queue NIC and running GNU[9]/Linux[7], we carried out several bandwidth tests, using a TCP/IPv4 bulk transfer, with/without IPsec, in order to have a first idea of the performance of an "out of the box" solution. More precisely, we use a bi-Xeon 5530 server (for a total of 8 cores), equipped with a 10 Gb/s NIC based on the Intel 82598EB chipset. The average bandwidth at application level (i.e. without counting the Ethernet, IP and TCP header overheads) is summarized in table 1. The evaluation tool used was Iperf[6].

Without IPsec/encryption, we can saturate the physical link, with average transmission rates between 8 and 9.2 Gb/s depending on the server load. However, when IPsec (ESP / tunneling mode, see section 4, using AES in counter mode) is

Table 1: First performance evaluation.

Without IPsec	9,2 Gb/s (best)
IPsec in tunnel mode (AES-CTR)	0,8 Gb/s
AES-CTR cipher (mono-thread)	0,8 Gb/s

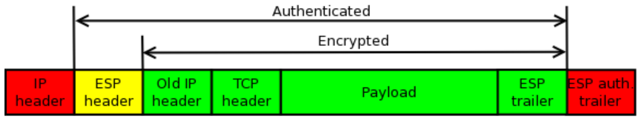


Figure 6: IPsec packet format.

used encrypt communications, performances drop by more than a factor 10. One can notice that this is roughly the same throughput as the AES-CTR cipher of the libcrypto library [8] in mono-thread mode.

The conclusions is that using the latest hardware (Nehalem processors, a multi-queue NIC) and kernel-space IPSec support is not sufficient to achieve high throughput: an "out of the box" GNU/Linux solution fails to achieve 10 Gb/s with a bulk encrypted traffic. More fundamentally, during the past 10 years, we observed a fundamental shift in CPU development, from frequency to parallelism, and this shift is also impacting motherboards and NIC. Therefore, the seek for high performance communications requires that developers take this situation into account and develop highly parallelized applications, which is the only possibility to take benefit of current and future servers, instead of counting on the raw performance growth of CPU cores.

4. ENCRYPTED COMMUNICATIONS AND PARALLELISM

IPsec.

A common solution for network-level encrypted communications is IPsec [10]. IPsec is mostly used with ESP (Encapsulated Security Payload) in tunnel mode, and provides confidentiality, source authentication and integrity verification services. When this mode is used in a security gateway, packets coming from the local network are encapsulated in a new IP packet as shown in figure 6. The ESP header allows the remote host to identify and use to same cipher and key. One of the ciphers supported by IPsec is AES which is considered as one of the best symmetric key algorithm of the moment. Therefore we concentrate our effort on this cipher in counter mode (section 4). Finally, IKE (Internet Key Exchange) is another protocol related to IPsec which allows two hosts to safely exchange the session keys. However since IKE does not impact our tests, we do not consider it.

Counter mode.

From the five ciphering modes of operations (Electronic Code-Book, Cipher Block Chaining, Cipher-FeedBack, Output FeedBack and Counter), we choose to use the Counter mode since it allows an easy parallelism[3]. More precisely (figure 7), using a set $\{nonce, counter\_0, f\}$ (where f is a function that produces a sequence guaranteed not to repeat for a long time and that enables to easily obtain $counter\_i = f(counter\_0, i)$) and a secret key K (constant during the communication), we can generate a pseudo random bit-stream

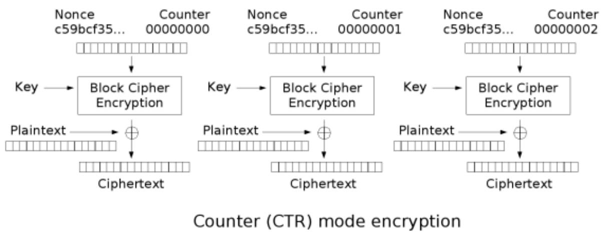


Figure 7: Cipher counter mode.

(or keystream). This keystream is then `XORed` to all the blocks M_i of the plaintext message M, during the encoding process, and the same keystream is `XORed` to all the blocks of the ciphertext message during the decoding process. Since the various blocks are encrypted/decrypted independently, the Counter mode allows an easy parallelization, by dispatching block processing over the computing units.

Parallelism and packet streams.

We are dealing with TCP flows, and therefore the TCP segments should be delivered in order to the receiving TCP engine. Indeed mis-ordered TCP segments are considered as the sign of a potential packet erasure over the network, which leads the receiver to generate immediate duplicated acknowledgements. Upon receiving three such duplicated acknowledgements, the TCP sender will immediately enter in congestion avoidance state and reduce the connection throughput accordingly. Therefore the parallel processing of incoming packets must preserve the ordering within a given TCP connection.

[12] introduces a work-stealing based algorithm to distribute tasks for a given stream over different processors in such a way that it guaranties the output ordering. The relative speedup is a factor 6 when using 8 processors, which is rather good compared to the factor 4 obtained with classic parallel algorithms.

5. EXPERIMENTING BANDWIDTH LIMITATIONS ON MULTICORE ENVIRONMENTS

Experimental settings.

We know (section 3) that even though the Nehalem architecture can handle 10Gb/s traffic without encryption, the standard IPsec implementation limits us to 0.8 Gb/s (which is also the monothread bandwidth of AES-CTR). We now want to assess our server *peak performance level* (see section 3 for server description), in the most ideal configuration, at the sender. To that purpose, one thread is dedicated to TCP/IP processing (no encryption) and several threads performs encryption over unrelated buffers (these buffers are not related to the TCP/IP packets, even if in a real situation, they would be IPsec packets). More precisely, the sending application sends 240 Gbit of data with the communication thread, and distributes the encryption of the corresponding number of data packets over the set of encryption threads. We then evaluate the upper encryption performance as 240 Gbit divided by the processing time of the slowest thread. As can be seen, these tests fail to catch the real behavior of

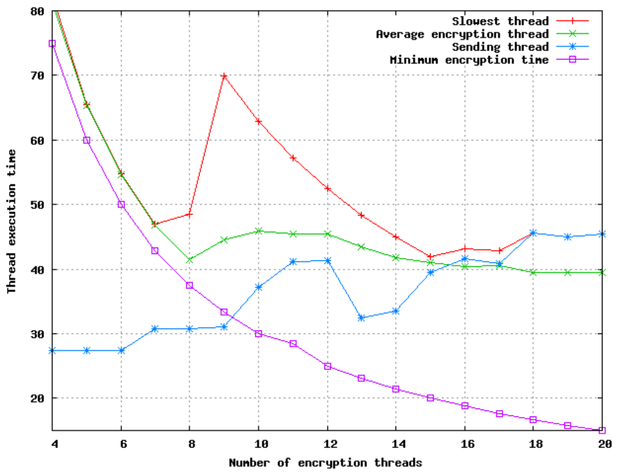


Figure 8: Communication and encryption thread execution times as a function of the number of encryption threads.

an IPsec protocol stack. However it is sufficient to provide an upper bound of the performance, and is useful to assess the feasibility of the problem: is a 10 Gb/s encryption feasible or not on this architecture?

Optimal number of threads.

First of all, we evaluate the number of threads that enable us to achieve the best performance. Figure 8 plots the measured time used by each thread as a function of the number of threads.

As one can predict, under 7 encryption threads, we are limited by the encryption thread processing time, which in turn is limited by the AES bandwidth. This test was performed on 240 Gbit of data, split in buffers of 3 kB. So for the 6 encryption threads (each having to encrypt 40 Gbit), one cannot expect better results than 50 seconds for each thread. We can see that threads are only a few seconds slower than the monothread bandwidth of the AES cipher itself. For the networking thread, we are under 30 seconds which corresponds to more than 8 Gb/s. In this part of the curve, there is a linear gain with the number of available threads.

Between 8 and 15 encryption threads, we are above the maximum number of threads that can truly run in parallel over the 8 cores. However, the CPU hyperthreading technique (HT) enables the parallel execution of these threads. By looking more carefully at the results, at first we see that the bandwidth decreases (figure 9) and HT seems to be a drawback. Then, between 13 and 15 encryption threads, the HT mechanism improves the achievable throughput.

To better understand the HT behaviour, we measure the thread execution time distribution for the 2 extremes (values are averaged over 10 tests): with 9 and 15 encryption threads (figure 10), to which we need to add the communication thread. More precisely, we performed 10 tests for each configuration, and we calculate the execution thread distribution. We see that the distribution is uniform with 15 encryption threads. On the opposite, with 9 encryption threads the distribution exhibits an important tail: we have six threads around 36 seconds, one thread between 45 and

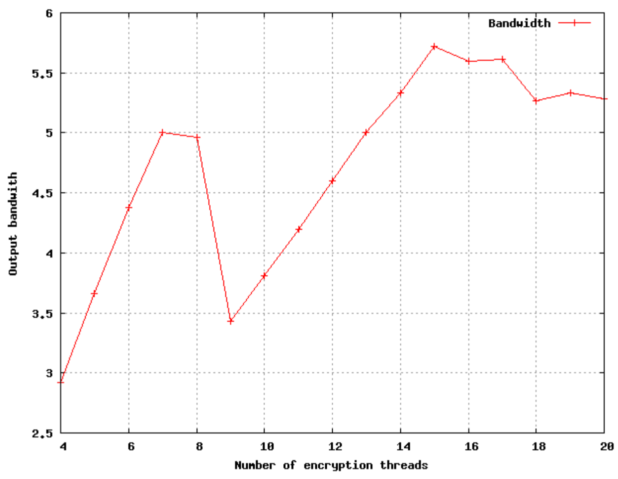


Figure 9: Bandwidth as a function of the number of encryption threads.

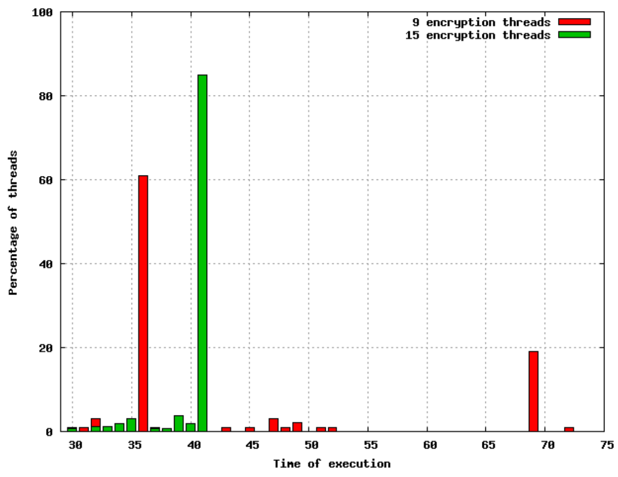


Figure 10: Communication and encryption thread execution time distribution.

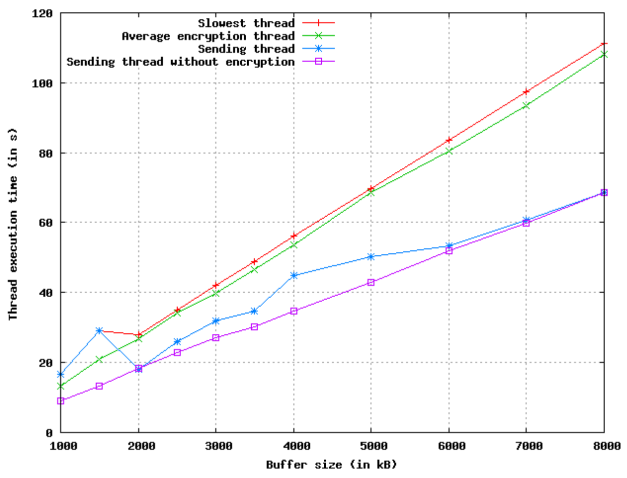


Figure 11: Sending time as a function of the data size.

50 seconds and two threads near 70 seconds. What happens here is that the first 8 threads use the 8 ”real” cores (one is in fact slower because of the impacts of other system processes), and the remaining threads are scheduled using the HT capability of the processors. This experiment shows that two threads are delayed and do not get access to a fair share of the available CPUs, which significantly impacts the estimated throughput that only takes into account the slowest thread (bottleneck). We do not experience such a phenomenon with 15 encryption threads, which shows that scheduling is done in a fair way.

Finally, with 16 or more encryption threads we do not experience any benefit in increasing the number of threads. We can conclude that optimum performance is achieved with 15 encryption threads, plus one communication thread, and that the maximum achievable throughput is around 5.75 Gb/s.

Data size impact.

Now that we identified the appropriate number of threads (15 for encryption, 1 for communications), we consider the influences of the data buffer size. Figure 11 shows that the execution time of all threads (encryption and communications) is a linear function of the data size. We also noticed that encryption is the bottleneck compared to communication.

Extrapolation to the receiver.

In a high-bandwidth context, a receiver will most likely use the polling mechanism instead of interruptions that generate high CPU consumption. This is possible because of the high-bandwidth context which implies the reception of millions of packets per second, which will be handled by a dedicated thread. Therefore the situation is rather similar to the sender side where a thread is dedicated to the sending operations. We can anticipate that the same bottleneck remains the same at the receiver side (this claim remains to be confirmed).

Partial conclusions.

Previous experiments have shown that even an up-to-date

8-core server (two Nehalem CPUs) cannot handle secure communications at the full 10 Gb/s Ethernet speed, and that encryption is the bottleneck. So we need to search another way of doing encryption without overloading the CPUs.

6. CYPHERING CAPABILITIES OF GPUS

GPU and CUDA.

The domain of Graphical Processing Units (GPU) recently made major progress, much faster than CPUs, and as a consequence, some GPUs now have more transistors than new quad-core CPUs. This is due to the fact that they are specifically designed for intensive and highly parallel computing, as is needed for image rendering. Since these GPUs have a highly parallel structure, many complex algorithms have been redesigned to take advantage of it. This approach also offloads computing intensive tasks on to the GPU, which saves a lot of processing time in the general purpose CPU.

The two major GPU manufacturers, NVidia and AMD, have both released development systems for GPU hardware, respectively CUDA and Stream. The Nvidia CUDA framework extends the C language to give access to the GPU, allowing C functions to run on the GPU stream processors in parallel. With CUDA, a programmer can use both regular C code and GPU code in the same file which simplifies the development process. CUDA abstracts the parallelism and gives the notion of "blocks" and "threads": several "threads" run within each "block". Each "block" is independent from other "blocks", and if "threads" from different "blocks" need to communicate they need to use the global memory of the GPU. Communication within a "block" is done using the block's memory. Finally, data needs to be moved from the host memory to the GPU memory, which is one of the main limitation of the approach because of the associated latency.

AES parallelization.

The AES algorithm is a standard algorithm, widely used in communication systems. With the CTR mode of operation, AES can be easily implemented in a parallel manner (section 4), and different parts of the cleartext message are processed on different GPU processing units. However, since each round in the algorithm depends on the previous round result, we cannot go any further and introduce a finer grain parallelism.

Experimental settings.

During the experiments, we use an NVIDIA GeForce GTX 295 GPU. We consider buffers of size in the range 1KB to 128KB, since our focus is on network packets whose size depends on the target physical layer, but is usually small. We then compare the results with those achieved on a CPU (an Intel bi-Xeon 5530 running at 2,4 GHz in this case).

Performance Evaluation.

We did several modifications on the implementation of [11] in order to get better results when encrypting files of small sizes. Indeed, using small files prevents to use the GPU in an optimal way for two reasons: first of all, the GPU cores are not all used in this case, and secondly, a lot of overhead is introduced when we wait for previous files to be processed. In order to solve this problem, we propose to use threads or what is known as "CUDA streams". In our experiments we define 1024 streams to cipher files of the same size. The idea is that operations are done asynchronously, so while computations are done on previous files, new files are transfered to the GPU. In addition, since the files are relatively small, the ciphering of each file is performed on one CUDA block in order to reduce the overhead of distributing the file across many blocks. These techniques allow to use all the stream processors of the GPU efficiently.

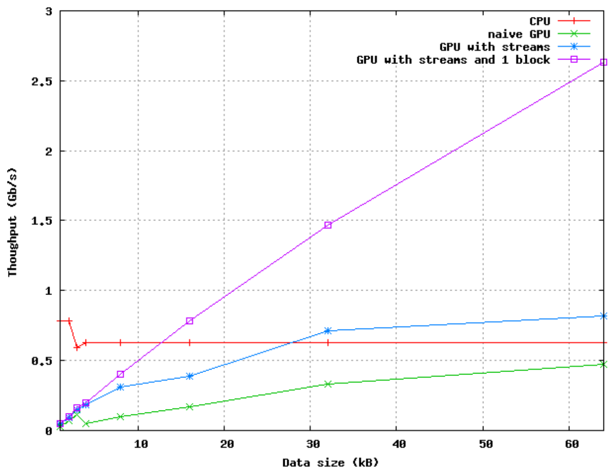


Figure 12: Cipher Throughput.

The results (figure 12) show that the CPU speed is almost constant, regardless of the file size. Concerning GPU, we compare three strategies: the first one is the naive implementation that was used for large file sizes; the second implementation uses CUDA streams to cipher asynchronously; and the last implementation limits the third implementation to use one block per file in order to reduce the overhead. The speed-up obtained from the third implementation over the naive one is 4 times for files of size 4KB. This increases up to 8 times in the case of files of size 128KB, reaching a throughput of 3,7 Gb/s. It should be noted that there is almost no significant speed-up in the case of files smaller than 4KB. Nonetheless, the throughput of very small files is still relatively low. To overcome this in terms of network packets, we propose the use of multiple GPUs (our server authorizes the use of up to four GPU cards) and the grouping of packets together to have files of higher size, almost 64KB or even 128KB. This may allow the reach of a throughput higher than 8Gb/s.

7. CONCLUSIONS AND PERSPECTIVES

This paper analyzes the raw capabilities of today's generic server for high performance secure communications. We show that relying on multi-core processors only is not sufficient to encrypt and send the data over the network at 10 Gb/s speed. We also show that the hyper-threading feature of processors should be used very carefully: correctly used, hyper-threading can help increase the sustainable bandwidth by distributing the processing load uniformly over the various cores, but certain configurations can also seriously decrease performances.

Since our experiments highlight the need for additional techniques, we also consider GPGPU for encryption operations. Although previous works in the domain have shown great results, it appears that for small data (i.e. one network packet) a GPGPU is slower than a CPU. However, we managed to optimize the AES implementation for the CUDA API and to achieve higher ciphering throughput than was previously feasible. We are continuing this effort in the hope to have similar results for data sizes in the order of a packet size.

Future research efforts will also address the use of multiple GPUs on a given server, as another possible way of improving performances. We are also considering the possibility to group several packets and to encrypt them together, as another way of improving the use of GPGPUs. However the practical impacts of doing so, both from a protocol point of view and inter-dependence point of view have to be seriously considered.

8. REFERENCES

[1] Cavium networks. `http://www.caviumnetworks.com/`.

[2] Cisco systems. `http://www.cisco.com/`.

[3] Counter mode. `http://http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Counter_.28CTR.29`.

[4] Intel nehalem architecture. `http://www.intel.com/technology/architecture-silicon/next-gen/`.

[5] Intel virtual machine device queue technology. `http://www.intel.com/network/connectivity/vtc_vmdq.htm`.

[6] Iperf. `http://perf.sourceforge.net/`.

[7] Linux. `http://kernel.org/`.

[8] Openssl's libcrypto manual page. `http://www.openssl.org/docs/crypto/crypto.html`.

[9] GNU. `http://www.gnu.org/`.

[10] IPsec. `http://tools.ietf.org/html/rfc4301`.

[11] C. Berk Guder. AES on CUDA. `http://github.com/cbguder/aes-on-cuda`, January 2009.

[12] J. Bernard, J.-L. Roch, and D. Traore. Processor-oblivious parallel stream computations. In *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, Toulouse, France, Feb 2007.

[13] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009.

[14] V. Danjean, R. Gillard, S. Guelton, J.-L. Roch, and T. Roche. Adaptive loops with kaapi on multicore and grid: Applications in symmetric cryptography. In A. publishing, editor, *Parallel Symbolic Computation'07 (PASCO'07)*, London, Ontario, Canada.

[15] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, 2009. ACM.

[16] N. Egi, A. Greenhalgh, M. Handley, G. Iannaccone, M. Manesh, L. Mathy, and S. Ratnasamy. Improved forwarding architecture and resource management for multi-core software routers. In *Network and Parallel Computing, 2009. NPC '09. Sixth IFIP International Conference on*, pages 117–124, Oct. 2009.

[17] P. Vicat-Blanc/Primet, V. Roca, J. Montagnat, J.-P. Gelas, O. Mornard, L. Giraud, G. Koslovski, and T. T. Huu. A scalable security model for enabling dynamic virtual private execution infrastructures on the internet. In *9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'09), Shanghai, China*, May 2009.

Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures

Brice Boyer   Jean-Guillaume Dumas[*]
Laboratoire J. Kuntzmann, UMR CNRS 5224,
Université de Grenoble
BP 53X, F38041 Grenoble, France.
{Brice.Boyer,Jean-Guillaume.Dumas}@imag.fr

Pascal Giorgi
Laboratoire LIRMM, UMR CNRS 5506,
Université Montpellier 2
F34095 Montpellier cedex 5, France.
Pascal.Giorgi@lirmm.fr

ABSTRACT

We propose different implementations of the sparse matrix–dense vector multiplication (SpMV) for finite fields and rings $\mathbf{Z}/m\,\mathbf{Z}$. We take advantage of graphic card processors (GPU) and multi-core architectures. Our aim is to improve the speed of SpMV in the LinBox library, and henceforth the speed of its black-box algorithms. Besides, we use this library and a new parallelisation of the sigma-basis algorithm in a parallel block Wiedemann rank implementation over finite fields.

Categories and Subject Descriptors

F.2.1 [**Numerical Algorithms and Problems**]: Computations in finite fields; G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems (direct and iterative methods); G.4 [**Mathematical Software**]: Parallel and vector implementations

Keywords

Sparse Matrix Vector multiplication, Finite field, Parallelism, Rank, Block Wiedemann

1. INTRODUCTION

Nowadays, personal computers and laptops are often equipped with multicore architectures, as well as with more and more powerful graphic cards. The latter can be easily programmable for a general purpose computing usage (Nvidia Cuda, Ati Stream, OpenCL). Graphic processors can offer nowadays superior performance on a same budget as their CPU counterparts. However, programmers can also efficiently use many-core CPUs for parallelization e.g. with the OpenMP standard.

[*]Part of this work was done while the second author was visiting the Claude Shannon Institute and the University College Dublin, Ireland, under a CNRS grant.

On the numerical side, several libraries automatically tune the sparse matrix kernels [19, 20, 16] and recently some kernels have been proposed for GPU's [17, 2, 1]. In this paper we want to adapt those techniques for exact computations so we first focus on $\mathbf{Z}/m\,\mathbf{Z}$ rings, with m smaller that a machine word.

The first idea consists in using the numerical methods in an exact way as done for dense matrix operations [7]. For sparse matrices, however, the extraction of sparse submatrices is different. Also, over small fields some more dedicated optimizations (such as a separate format for ones and minus ones) can be useful. Finally, we want to be able to use both multi-cores and GPU's at the same time and the best format for a given matrix depends on the underlying architecture.

Therefore, we propose an architecture with hybrid data formats, user-specified or heuristically discovered dynamically using ad-hoc sparse matrix completions. The idea is that a given matrix will have different parts in different formats adapted to its data or to the resources. Also we outline a "just-in-time" technique that allows to compile on the fly some parts of the matrix vector product directly with the values of the matrix.

We have efficiently implemented[1] "`S`parse `M`atrix-`V`ector multiplication" (SpMV) on finite rings, together with the transpose product and the iterative process to compute the power of a matrix times a vector.

We also make use of this library to improve the efficiency of the block Wiedemann algorithm's of the LinBox[2] library. Indeed, this kind of algorithm uses block "black box" [13] techniques: the core operation is a matrix-vector multiplication and the matrix is never modified. We use the new matrix-vector multiplication library, together with a new parallel version of the sigma-basis algorithm, used to compute minimal polynomials [11, 8].

In section 2 we present different approaches to the parallelization of the SpMV operation: the adaptation of numerical libraries (section 2.3) and new formats adapted to small finite rings (section 2.5) together with our new hybrid strategy and their iterative versions (section 2.6). Then in section 3 we propose a new parallelization of the block Wiedemann rank algorithm in LinBox, via the parallelization of the matrix-sequence generation (section 3.1) and the parallelization of the matrix minimal polynomial computation (section 3.2).

[1]`https://ljkforge.imag.fr/projects/ffspmvgpu/`
[2]`http://linalg.org`

2. SPARSE-VECTOR MATRIX MULTIPLICATION

We begin with introducing some notations. In this section, we will consider a matrix A; the element at row i, column j is $A[i, j]$. The number `nbnz` is the number of non zero elements in matrix A, it has `row` lines and `col` columns. If $\mathbf{x}$ and $\mathbf{y}$ are vectors, then we perform here the operation $\mathbf{y} \leftarrow A\,\mathbf{x} + \mathbf{y}$. The general operation $\mathbf{y} \leftarrow \alpha A\,\mathbf{x} + \beta\,\mathbf{y}$ can then be done by pre-multiplying $\mathbf{x}$ and $\mathbf{y}$ by α and β respectively. The operation $\mathbf{y} \leftarrow A^{\top}\mathbf{x} + \mathbf{y}$ is done by pre-transposing the matrix A. The `apply` operation in LinBox black box algorithms, or $\mathbf{y} \leftarrow A\,\mathbf{x}$, is performed by first setting the elements of $\mathbf{y}$ to zero. For further use in block methods, we also provide the operation $\mathbf{Y} \leftarrow \alpha A\,\mathbf{X} + \beta\,\mathbf{Y}$ where $\mathbf{X}$ and $\mathbf{Y}$ are sets of vectors (or multivectors).

2.1 Sparse Matrix Formats and Multiplication

Sparse matrices arise form various domains and their shapes can be very specific. Taking into consideration the structure of a sparse matrix can dramatically improve the performance of SpMV. However, there is no general storage format that is efficient for all kind of sparse matrices.

Among the most important storage formats is the `COO` (coordinate) format which stores triples. It consists of three vectors of size `nbnz`, named `data`, `colid` and `rowid`, such that `data[k]`= A `[rowid[k], colid[k]]`.

The `CSR` (compressed storage row) stores more efficiently the previous representation: the `rowid` field is replaced by a $(\texttt{row} + 1)$ long `start` vector such that if $\texttt{start}[i] \leqslant k < \texttt{start}[i+1]$, then $\texttt{data}[k] = A[i,\texttt{colid}[k]]$. In other words, `start` indicates where a line starts and ends in the other two ordered fields.

The `ELL` (ELLpack) format stores data in a denser way: it has `data` and `colid` fields such that $\texttt{data}[i, j_0] = A[i, \texttt{colid}[i, j_0]]$, where j_0 varies between 0 and the maximum number of non zero elements on a line of A. One notices that these fields can be stored in row-major or column-major order. A variant is the `ELL_R` format that adds a `row` long `rownb` vector that indicates how many non zero entries there are per line.

The `DIA` (DIAgonal) is used to store matrices with non zero elements grouped along diagonals. It stores these diagonals in an array along with offsets where they start. We refer to [1],[17] for more details on these formats.

This very schematic description of a few well-known formats shows that each of them has pros and cons. Our aim is to produce a more efficient implementation of the SpMV operation on finite fields than the one present in LinBox, first taking advantage of this variety of formats.

2.2 Finite field representation

We present now how the data is stored. We use common data types such as `float`, `int`,... Firstly, when doing modular linear algebra over $\mathbf{Z}/m\mathbf{Z}$, we try to minimize the number of costly `fmod` (reduction $\bmod m$) operation calls. Indeed, this `fmod` operation is the one difference with numerical implementations of SpMV and needs very special attention. For instance, we prefer, if possible, the right loop to the left one in the following figure 1:

```
for (i=0 ; i<n ; ++i){
    y += a[i] * b[i] ;
    y =  fmod(y,m);
}
```

```
for (i=0 ; i<n ; ++i){
    y += a[i] * b[i] ;
}
y =  fmod(y,m);
```

Figure 1: Delaying `fmod`

In this case, suppose that $y = 0$, $a[i]$ and $b[i]$ are reduced modulo m at first. Let M be the largest representable integer. On $\mathbf{Z}/m\mathbf{Z}$, we can represent the ring on $[\![0, m-1]\!]$. Then we can do at most $M/(m-1)^2$ accumulations before reducing. We can also represent the ring on $[\![-\lfloor\frac{m-1}{2}\rfloor, \lceil\frac{m-1}{2}\rceil]\!]$. This last representation enables us to perform twice more operations before a reduction, but this reduction is slightly more expensive. Another trade-off consists in choosing a `float` representation instead of `double` (on the architectures that support `double`). Indeed, operations can be much faster on `float` that on `double` but the `double` representation lets us do more operations before reduction. This is particularly true on some GPU's.

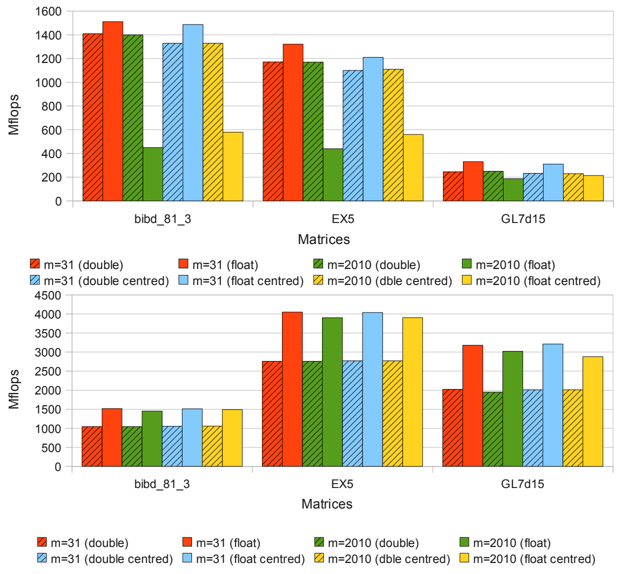


Figure 2: `float`–`double` trade-off for different sizes of m, on the CPU and GPU (`ELL_R` format)

In figure 2, we present variations in efficiency due to the storage data type and the size of m on one core of a 3.2GHz Intel Xeon CPU and a Nvidia GTX280 GPU. The timings correspond to the average of 50 SpMV operations, where $\mathbf{x}$ and $\mathbf{y}$ are randomly generated on the CPU; every data transfer between the CPU and GPU is taken into account. The measure unit corresponds to the number of million floating point operations per seconds (flops); a SpMV operation requires $2 * \texttt{nbnz}$ such operations. The data storage here is `ELL_R` and the matrices[3] are presented in table 1. This figure shows a significant slow down for for `double` operations on this GPU and a minor one on this CPU. On the border case (large prime, small data type), we can see a large performance gain on the CPU but not on the GPU. We suspect the massive parallelisation on the GPU hides the cost of the `fmod` operation.

[3]matrices available at `http://www-ljk.imag.fr/membres/Jean-Guillaume.Dumas/simc.html`

name	mat1916	bibd_81_3	EX5	GL7d15	mpolyout2
`row`	1916	3240	6545	460261	2410560
`col`	1916	85320	6545	171375	2086560
`nbnz`	195985	255960	295680	6080381	15707520
rank	1916	3240	4740	132043	1352011

Table 1: Matrices overview

For further information about these techniques on these rings and fast arithmetic in Galois extensions, see e.g. [7].

Note: in the sequel of the paper, unless otherwise mentionned, the timings will be done on `float` with $m = 31$ and the classical representation.

2.3 Adapting numerical libraries

Another speed-up consists in using existing numerical libraries. The ideas behind using them on the rings $\mathbf{Z}/m\,\mathbf{Z}$, is twofold. Firstly, we delay the modular reduction, secondly we can use highly optimized popular libraries and get instant speed-ups as compared to more naïve "home made" routines.

Just like BLAS libraries can be used to speed up modular linear algebra [9], we can use numerical libraries for our purposes, or get inspiration for our algorithms from their techniques. For instance, there is the OSKI library [19] for sequential numerical SPMV, or the GPU implementation of SPMV by Nathan Bell *et al.* in [1]. The BLAS specifications include Sparse BLAS[4] but these routines are seldom fully implemented in free BLAS implementations.

Unfortunately, numerical libraries usually cannot be used as-is. We need to extract submatrices from the sparse matrices, which is more complicated than for its dense counterpart when the use of strides and dimensions suffices. For instance, let us suppose one can do b accumulations on $\mathbf{y}[i]$ before an overflow may happen and we need to reduce. Suppose too that line i of A has r_i non zero elements. Then we want to split this line between $\lceil r_i/b \rceil$ matrices. We can improve this technique with a finer majoration. We split the elements in row i into a disjoint union of κ_i sets $S_{i,k}$. Let μ be the largest (in absolute value) element we can represent in the ring (usually m or $\lceil m/2 \rceil$). For all i, k, we demand that $\sum_{\alpha \in S_{i,k}} |\alpha|\, \mu < M$ and create $\max_i(\kappa_i)$ submatrices.

Eventually, we can use the numerical libraries on these submatrices we have created. The general algorithm reads as follows:

```
spmv(y,A,x){
    foreach submatrix Ai  in A  do {
        spmv_num(y,Ai,x); //no overflow guaranteed
        reduce(y,m);
    }
}
```

Figure 3: Using numerical routines

On the CPU, numerical routines have been written for every format and on the GPU we use an adaptation of Nathan Bell's `cusp` library.

2.4 Using OpenMP

For the ease of implementation and for it is becoming a well-used standard, we chose `OpenMP` to parallelise our code on the CPU. In most cases, we simply added a `#pragma omp` above the outer loops. This technique gave good performances as shown in figure 4.

[4]`www.netlib.org/blas/blast-forum/chapter3.pdf`

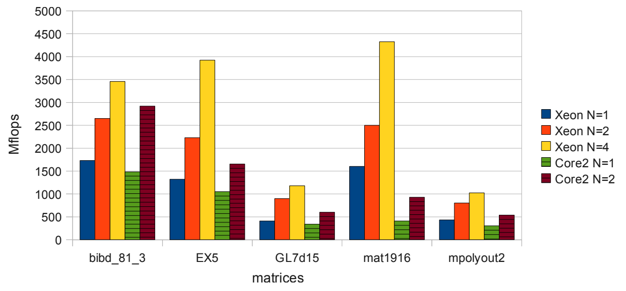


Figure 4: OpenMP-parallelised `CSR` with N cores, on an 8 core 3.2GHz Intel Xeon and a 3GHz Intel Core2Duo processor

As we can notice on this figure, `OpenMP` help scale well the performances, with a few processors.

2.5 New formats

Most of the formats implemented show a row-level parallelism, except `COO` that has element-wise parallelism. The `COO` case is not obvious to implement and is generally much slower. The parallel efficiency of other formats will depend then on the length of the rows as well as the data regularity. Unbalanced rows on a GPU architecture will produce many idle threads. Two solutions exist: the vector approach of Bell *et al.* (they split the rows into shorter chunks and reduce) or the rearranging of rows with permutations to sort the rows according to their length. The last idea will not work in e.g. a power distribution of the row lengths. The `ELL` format answers very well this problem because each row has the same length. However, one has to be very careful about the amount of unnecessary memory allocated by `ELL` in case of very uneven row lengths. Then, as proved in e.g. [1], yet another solution consists in splitting the matrix A into a sum of matrix, one dense part in `ELL`, the other in `COO` format.

An other way to parallelise the SPMV operation is to split the matrix A along rows to get smaller submatrices and treat them in parallel. We took this approach on the CPU `COO` algorithm.

Also, we have to keep in mind that we are dealing with large matrices, used many times as black-boxes. Therefore there is a trade-off between the time spent on optimizing the matrix and how much faster these optimizations will make SPMV run.

Things to consider during preprocessing may include for instance: reordering row-columns to create denser parts, choosing best-fitting formats, cutting the matrix into efficient sub-matrices ([20],[16])... The preprocessing approach is taken by OSKI: if the expected number of SPMV is very high, optimizing the matrix deeper will prove efficient.

2.5.1 *Base case: JIT*

One idea to improve SPMV on a given matrix is to hard code this operation in a static library. We read the matrix file and create a library that will apply this matrix to input vectors. For instance the $\mathbf{y} \leftarrow \mathbf{y} + A\,\mathbf{x}$ operation on the matrix $\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$ would be translated to (if $m = 27$):

```
void spmv(float * y, const float * x) {
   y[0] += 2*x[0] ;
   y[0] += x[1] ;
```

```
    y[0] = fmod(y[0],27);
    y[1] += 3*x[1] ;
    y[1] = fmod(y[1],27);
}
```

Then we compile this generated file a as static library and use `dlopen` to access its functions. As we can see in this example, one can implement various optimizations: rearranging the rows so that the work is more even, replacing the occurrences of ± 1 in the matrix by less costly additions or subtractions. However, large matrices take extremely long to compile, even if the matrix is divided into smaller (easier to compile) submatrices. Only then for instance, we could compile `bibd_81_3` but it took 63s on the same Xeon machine. Once it is compiled, the CPU version runs at 620 Mflops, which is reasonably fast but not usable. This idea did not take into account the pressure on the instruction cache and the possibly small bandwidth of certain architectures. However, it produced the idea for the following data formats.

2.5.2 *Taking into account the* ± 1

The example of JIT and the observation that many matrices arising from different applications have a lot of $\pm 1_F$ attracted our attention on this special case. Moreover, many matrices on a small fields also share this property. Thus we can extract two submatrices corresponding to the 1 and -1 from the rest of the matrix and replace multiplications by usually less expensive additions. Besides, the `data` field in most formats (except `ELL`, `DIA`) can be forgotten as we know they only consist of 1 or -1: this reduces the memory usage. Doing only additions as opposed to `axpy` also hugely delays reduction.

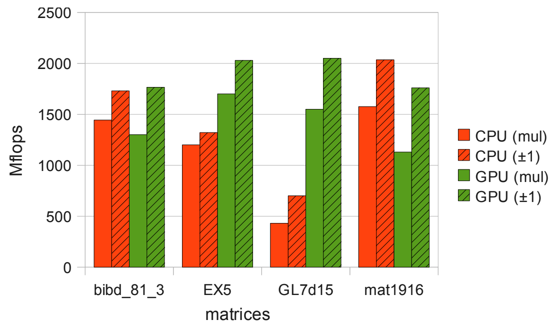


Figure 5: Speed improvement on one 3.2GHz Intel Xeon CPU and a Nvidia GTX280 GPU when segregating or not the ± 1 (`CSR` format)

Figure 5 shows a significative improvement on these matrices. `GL7d15` has close to half 1 and half -1, `bibd_81_3` and `EX5` are only constitued of 1 and 55% of the non zero coefficients of `mat1916` are 1s.

The only drawback in singling out the ± 1 and creating accordingly new formats is that, to our knowledge, there is no numerical library for them.

2.5.3 *Basic Formats*

As evoked earlier, the matrix A can be split into smaller submatrices. These submatrices can have a format adapted to them and/or can be treated differently. For instance, we can split row-wise and distribute these matrices for parallelism, or split them column-wise as in the delaying case (figure 3). This makes (possibly) many matrices that we each want to optimize individually so we get better overall performance.

We start with some observations. The `COO` format is slow due to the many `fmod` calls, it is best used when the matrix is extremely sparse. The `CSR` format is denser and can let delayed reduction occur, but one has to ensure the row lengths are well balanced when parallelizing. The `ELL` formats are very efficient on matrices that have roughly the same number of non zeros per line. The `ELL_R` format ([17]) is better for less even row lengths. One difference in the CPU and GPU architecture makes the `ELL` row-major on the CPU (for better cache use) and column-major on the GPU (for better coalescing). The following figure shows on one example (`bibd_81_3`) the variation of efficiency. The data is normalized so that `CSR` is 1 on the CPU or GPU.

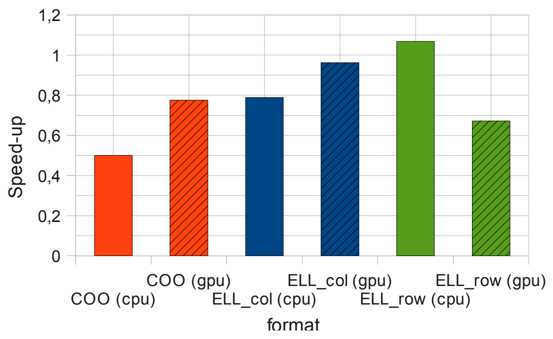


Figure 6: Speed-ups for various formats on matrix `bibd_81_3` both on one 3.2GHz Intel Xeon CPU and a Nvidia GTX280 GPU; reference is `CSR` on each architecture

2.5.4 *Hybridization*

The previous remarks lead us to combine these formats to take advantage of them. Hybrid formats such as `ELL(_R)+COO` or `ELL(_R)+CRS` give good performance on the GPU. When the `ELL` part is taken out of a matrix, many rows can be left empty. Then, we use a format called `COO_S` that is a `CSR` format with pointers only to the non empty rows. It has `data`, `colid` same as in `CSR` and `COO`. The number `rowid`$[k]$ corresponds to the k^{th} non empty row that starts in `data` and `colid` at `start`$[k]$. This format could be avoided if we used row permutations and ordered the lines according to their weight.

2.5.5 *Heuristic format chooser*

The previous remarks show a great complexity in the formats and the cutting of the matrix. We have implemented a user-helped heuristic format chooser. For instance, the user can indicate if she wants to try and make use of ± 1. If so, for each submatrix, the program tries to find an *a priori* efficient format for them or if it fails, does not separate the 1 or the -1 from the rest. She can also indicate what is the format she wants to fill in priority.

The hybridization of the matrix is usually done as follows. If the matrix is large enough and most of the lines are filled, it will try to fit a part of the matrix in an `ELL` or `ELL_R` format. This choice is supported by the observation that, on one hand, many matrices have a $c+r$ row distribution where c is some constant and $r \in \mathbf{Z}$ varies and that, on the other hand, `ELL` is generally much faster that other formats for matrices with even row weights. The rest of the matrix will be put in a CSR, COO or COO_S format, according to the number of empty lines and the number of residual non zero elements. Parameters that decide when segregating the 1s, that choose the best length for ELL matrix, etc., vary according to the architecture of the computer and need some specific tuning. This tuning is not yet provided at compile time but some of it could be automatically performed at install time.

Experiments (figure 7) show that this heuristic often gives equal or better results that simple formats on the CPU and the GPU.

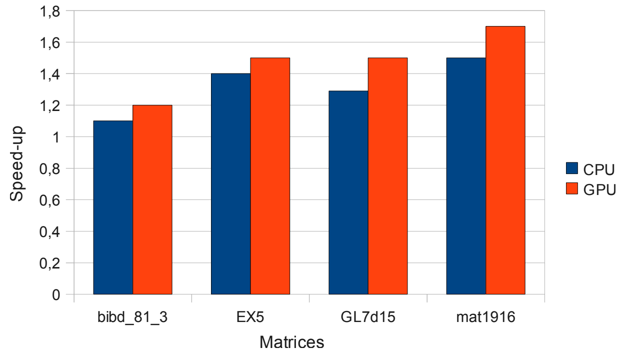


Figure 7: Speed-up of the auto-generated format over CSR.

2.6 Block and iterative versions

2.6.1 *Using multi-vectors*

We have described the SpMV operation $\mathbf{y} \leftarrow A\,\mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ are vectors. We also need $\mathbf{x}$ and $\mathbf{y}$ to be multi-vectors, for they may be used in block algorithms. There are at least two ways to represent them : row or column-major order. In the row-major order, we can use the standard SpMV many times (and align the vectors). In the column-major order, we can write dedicated versions that try and make use of the cache. Indeed, in this case, we traverse the matrix only once and $\mathbf{x}$ and $\mathbf{y}$ are read/written contiguously.

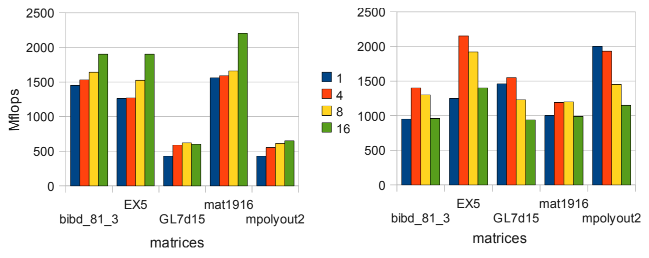


Figure 8: Matrix-multivector multiplication speed on one 3.2GHz Intel Xeon CPU (left) and a Nvidia GTX280 GPU (right) for column-major multi-vectors, with $1, 4, 8$ and 16 vectors. (ELL_R format)

On figure 8, we note that on the CPU, using column-major multivectors is a non negligible gain of speed. On the contrary, the GPU implementation fails to sustain good efficiency for blocks of more than 8 vectors. We suspect the problem comes from a bad use of local memory. Besides, some large matrices start to reach the memory limitation.

2.6.2 *Performance issues*

The GPU operation on a single SpMV call from the host point of view is very slow because we need to move the vectors between the host and the device. It is therefore only usable on operations that need no data moving between the host and the device. Examples include the computation $y \leftarrow A^n\,x$ or the computation of the sequence $\left\{A^i x\right\}_{i\in[\![0,m]\!]}$ that are used in many of the black box methods.

On figure 10, we illustrate this difference, mostly reusing or not the data on the GPU, by comparing the performance of the following two pseudo-codes (figure 9):

```
void smpv_n(y,A,x,n){
    y_d = copy_on_gpu(y);
    x_d = copy_on_gpu(x);
    A_d = copy_on_gpu(A);
    for (i=0 ; i<n ;++i) {
        y_d = A_d * x_d ; // spmv on GPU
        x_d = y_d;        // full copy
    }
}

void n_spmv(y,A,x,n){
    A_d = copy_on_gpu(A);
    for (i=0 ; i<n ;++i) {
        y_d = copy_on_gpu(y_i);
        x_d = copy_on_gpu(x_i);
        y_d = A_d * x_d ; // spmv on GPU
    }
}
```

Figure 9: Pseudo code for $y \leftarrow A^n x$ and n times $y \leftarrow Ax$ on the GPU with x randomly generated on the CPU.

As expected, figure 10 clearly supports that it is highly desirable not to move data between CPU and GPU when avoidable. The speed-up is noticeable even from a very small number of iterations.

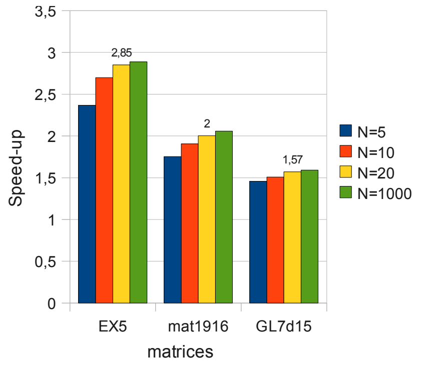


Figure 10: Nvidia GTX280 GPU speed up of $y \leftarrow A^n x$ compared to n times $y \leftarrow Ax$, with $n = 5, 10, 20$; CSR format is used.

Note : plots thoughout this paper all take into account the data transfer between CPU/GPU and back at each SpMV operation.

3. PARALLEL BLOCK WIEDEMANN ALGORITHM

Some of the most representative applications requiring efficient sparse matrix-vector product are blackbox methods based on the Lanczos/Krylov approach. In particular, the method proposed by Wiedemann [21] and its block version proposed by Coppersmith [5] are well suited to highlight efficiency of sparse matrix-vector product since the latter is quite often their bottleneck.

As an application, we propose to improve the implementation of the Block Wiedemann rank algorithm presented in [8]. Let us first briefly recall the outline of this algorithm, we let the reader refer to e.g. [15] for further details.

Let $A \in \mathsf{F}^{n\times n}$ be a matrix satisfying the preconditions of [14]. Then the algorithm can be decomposed in three steps:

1. Compute the matrix sequence $S_i = Y^T A^i Y$ for $i = 0..2n/s + O(1)$, with $Y \in \mathsf{F}^{n\times s}$ chosen at random

2. Compute the minimal matrix generator $F_Y^A \in \mathsf{F}^{s\times s}[x]$ of the matrix series $S(x) = \sum_i S_i x^i$

3. Return the rank $r = \deg(\det(F_Y^A)) - \operatorname{codeg}\det(\mathrm{F}_\mathrm{Y}^\mathrm{A})$.

Our approach is to separate the parallelization of each step. The first step is clearly related to sparse matrix-vector product and we will re-use our tools presented in previous sections. The second step needs the computation of a minimal matrix generator. This can be achieved by a σ-basis computation as explained in [8, section 2.2]. Finally, the last step reduces to computing the co-degree of the determinant of the σ-basis . The degree of the determinant being directly computed as the sum of the row degrees of F_Y^A since, due to the σ-basis properties, the matrix is already in Popov form.

3.1 Parallelization of the matrix sequence generation

The parallelization proposed in [8] was to ship independent set of vector blocks of V to different cores and apply them in parallel. Then gather the results to compute the dense dot products by U^T.

An alternative is to use the SpMV library and let it take care of the iteration with the algorithm of the preceding section.

In figure 11 we compare both approaches:

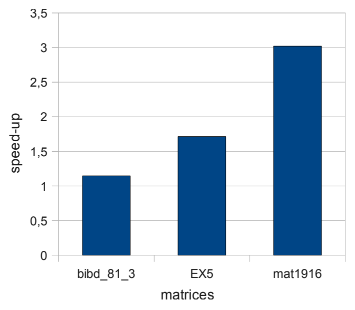


Figure 11: Speed up from the new SpMV library compared to the native LinBox implementation in the generation of the matrix sequence ($2n$ iterations) on one core of a 2.33GHz Intel Xeon E5345 CPU

3.2 Parallelization of the σ-basis computation

One can efficiently compute σ-basis using the algorithm PM-Basis of [11]. This algorithm mainly reduces to polynomial matrix multiplication. Therefore a first parallelization approach is to parallelize the polynomial multiplication.

3.2.1 Parallel polynomial matrix multiplication

Let $A, B \in \mathsf{F}^{n\times n}[x]$ be two polynomial matrices of degree d. One can multiply A and B in $O(n^3 d + n^2 d \log d)$ operations in F assuming F has a d-th primitive root of unity [3]. Assuming one has k processors such that $k \leqslant n^2$, one can perform this multiplication with a parallel complexity of $O(\frac{n^3 d}{k} + \frac{n^2 d \log d}{k})$ operation in F. Let us now see the sequential fast polynomial matrix multiplication algorithm and how it achieves such a parallel complexity:

```
*Fast Polynomial Matrix Multiplication:*
**Inputs:** $A, B \in \mathsf{F}^{n\times n}[x]$ of degree $d$, $\omega$ a $d$-th primitive root
of unity inf $\mathsf{F}$.
**Outputs:** $A \times B$
  1. $\bar{A} := DFT(A, [1, \omega, \omega^2, ..., \omega^{2d}])$
  2. $\bar{B} := DFT(B, [1, \omega, \omega^2, ..., \omega^{2d}])$
  3. $\bar{C} := \bar{A} \otimes \bar{B}$
  4. $C := \frac{1}{2d} DFT(\bar{C}, [1, \omega^{-1}, \omega^{-2}, ..., \omega^{-2d}])$
  **return** C.
```

Here, $DFT(P, L)$ means the multi-points evaluation of the polynomial P on each points of L, while $\otimes$ means the point-wise product.

- step 1,2 and 4 can be accomplished by using Fast Fourier Transform on each matrix entries which gives $n^2 \times O(d \log d)$ operations (see [10, Theorem 8.15]). This clearly can be distributed on k processors such that each processor achieves in parallel the FFT on $\frac{n^2}{k} + O(1)$ matrix entries. This gives a parallel complexity of $O(\frac{n^2 d \log d}{k})$ operations in F.

- step 3 requires the computation of $2d$ independent matrix multiplications of dimension n, which gives $O(n^3 d)$ operations in F. One can easily see how to distribute this work on k processors such that each processor has a workload of $O(\frac{n^3 d}{k})$ operations.

We report in figure 12 the performance of the implementation of this parallel algorithm in the LinBox[5] library. Our choice of using this parallel algorithm rather than another one, achieving a possible better parallel complexity, has been driven by the re-usability of efficient sequential components of the library (e.g. matrix multiplication) and the ease of use within the library itself (i.e. mostly the same code as sequential one, only some OpenMP pragmas have been added).

One can see on figure 12 that our coder does not completely match the theoretical parallel speedup. The best we can achieved with 16 processors is a speedup of 5.5, which is only one third of the theoretical optimality. Nevertheless, one can see that with less processors (e.g. less than 4) the speedup factor is closer to 75% of the optimality, which is quite fair. We think this phenomenon can be explained by the underlying many multi-core architecture (Quad-Core

[5]www.linalg.org

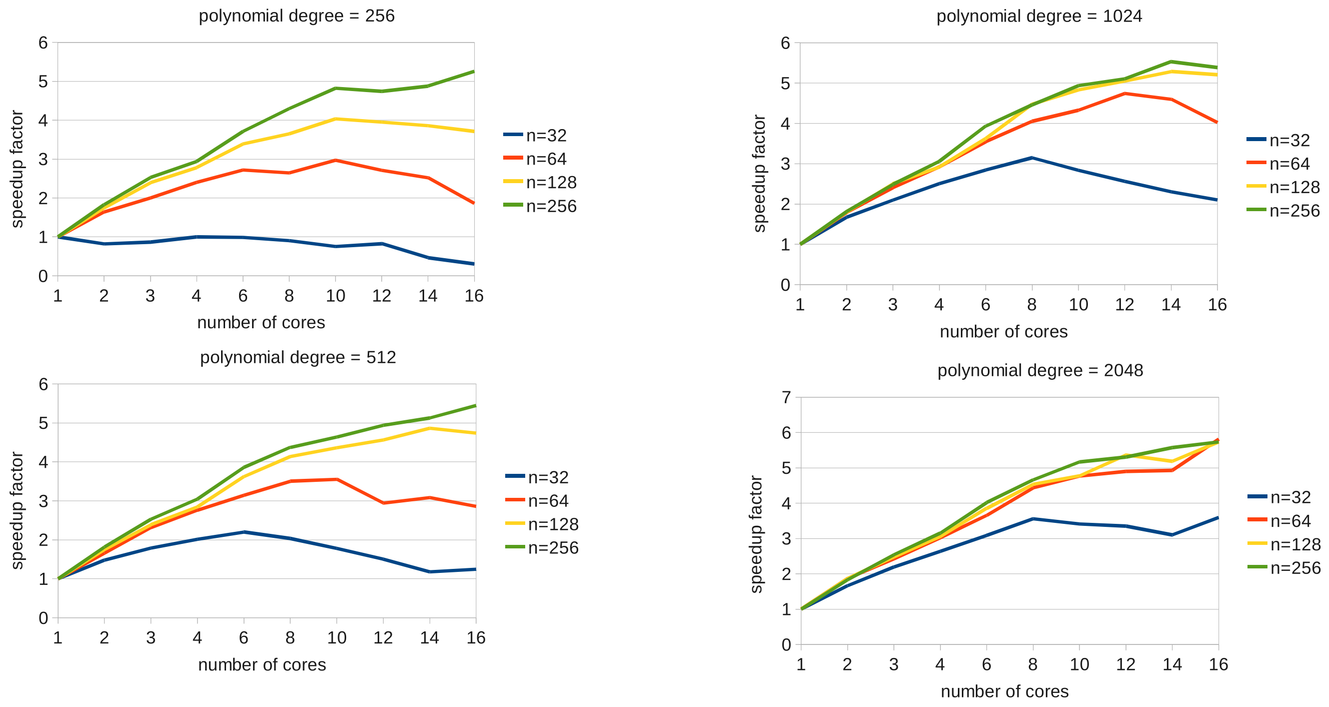


Figure 12: Scalability of parallel polynomial matrix multiplication with LinBox and OpenMP on a 16 core machine (based on Quad-Core AMD Opteron). n **is the matrix dimension.**

AMD Opteron), which may clearly suffers from cache effect if computation are done on same chip or not.

As expected, we can also point out from figure 12 that our implementation benefits at most from parallelism when matrices are larger. Since workload on each core is more important, this allows to hide the penalty from memory operations and threads management of OpenMP. This remarks also applies on the degree but the impact is less important.

3.2.2 *Parallel σ-basis implementation*

According to the reduction of PM-Basis to polynomial matrix multiplication, one can achieve a parallel complexity of $O\tilde{}(\frac{n^3 d}{k} + \frac{n^2 d \log d}{k})$ operations in F with k processors for σ-basis calculation, assuming $k \leqslant n^2$. Therefore, it suffices to directly plug in our parallel polynomial matrix multiplication into the original code of the LinBox library to get a parallel σ-basis implementation.

We report in figure 13 the performance of the parallel version of PM-Basis algorithm within LinBox. Here again, the speedup factor of parallelism is quite low when compared to the theoretical optimality. At most we were able to obtain a speedup of 3 with 16 processors. However, this timings are consistent with the previous ones in figure 12 where the best speedup was 5.

One may notice that reduction to polynomial matrix multiplication of the PM-Basis algorithm relies on a divide a conquer approach on the degree of the approximation (see [11, theorem 2.4]). Therefore, the recursion calls are made with smaller and smaller approximation's degrees, which leads to use less efficient parallel multiplications. Moreover, when the degree is too small, the use of the M-Basis algorithm of [11] should be prefered since it becomes more efficient in practice. We have not yet implemented a parallel implementation of this algorithm in LinBox and this clearly affects the performance of our implementation.

3.3 Parallel determinant co-degree

Here we just launch in parallel the evaluations of the matrix polynomial at different points, and the computation of the determinant of the obtained matrix at the given point, and gather the results sequentially with the `Poly1CRT` class of Givaro.

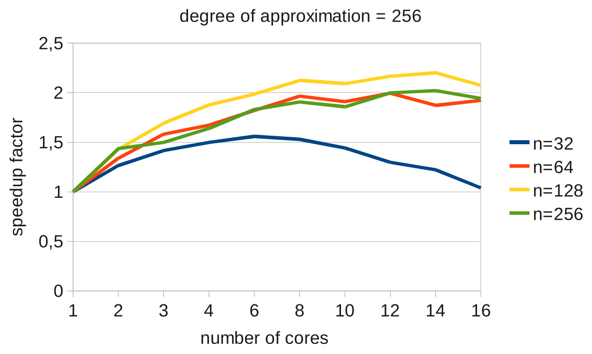
3.4 Parallel block Wiedemann performance

In table 2 we show the overall performance of our algorithm on an octo-processor Xeon E5345 CPU, 8 × 2.33GHz. `*-LB` shows the timings of the current LinBox implementation, where `*-SpMV` presents our new improvement, both in sequential and in parallel. The speed-up for SpMV between 1 and 8 processors is slightly larger than 5 for all the matrices where the speed-up for LinBox ranges from 4 to 4.9. Furthermore, the speed-up obtained with SpMV versus LinBox on the sequence generation seems scalable as it even improves when used in a parallel setting. Comparing with figure 11, we can confirm that the bottle-neck in the sequence phase is really the SpMV operation.

This table will be completed in the future with a line including the generation of the sequence on the GPU. It involves porting some FFLAS/FFPACK functionalities to the GPU using Cublas which is underway.

4. CONCLUSION

We have proposed a new SpMV library providing good results on $\mathbf{Z}/m\mathbf{Z}$ rings. To attain this efficiency it has been mandatory to augment the complexity of the SpMV algorithms, since OpenMP, Cuda et al. all manage differently



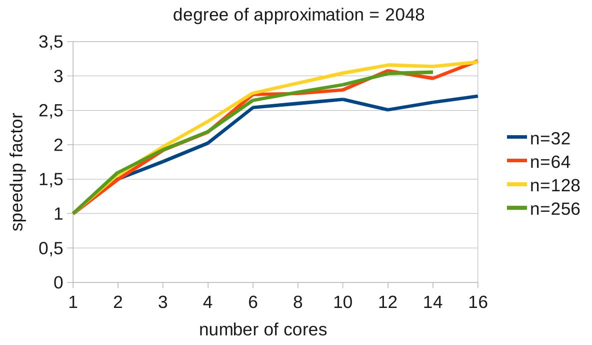


Figure 13: Scalability of parallel σ-basis computation with LinBox and OpenMP on a 16 core machine (based on a Quad-Core AMD Opteron). n is the matrix dimension of the series.

Matrix	mat1916		bibd_81_3		EX5	
Cores	1	8	1	8	1	8
Seq-LB	15.09	3.08	47.73	12.41	84.21	20.22
Seq-SpMV	5.02	0.91	41.28	7.56	49.66	7.36
σ-basis	9.02	1.64	18.45	3.63	37.45	8.39
Interpolation	0.37	0.29	1.07	0.82	2.29	1.75
Total-LB	24.48	5.01	67.25	16.86	123.95	30.36
Total-SpMV	14.41	**2.84**	60.80	**12.01**	89.40	**17.50**

Table 2: Rank modulo 65521 with OpenMP Parallel block Wiedemann on a Xeon E5345, 8 $\times$ 2.33GHz (timings in seconds)

the parallelization. Nonetheless, we provide new hybrid formats that improve the performance. Moreover we have also specialized it to the computation of a sequence of matrix-vector products together with a new parallelization of the sigma-basis algorithm in order to enhance e.g. rank computations of very large sparse matrices. As seen in 3.2.2, a first parallelization of the σ-basis computation has been achieved. Its efficiency is not matching the expected scalability and lot of work needs to be done to circumvent this problem. First, a deeper study on the parallelization of σ-basis computation has to be done. Beside the parallelization of PM-Basis and M-Basis algorithms themselves, we need to design new algorithms to avoid the numerous task dependencies, inherent to the existing methods. This will also enable an easier parallelization of early termination strategies (requiring to interleave the generation sequence and the σ-basis computation).

Another important task is to extend the sigma-basis algorithm to work on polynomial matrices over extension fields. Indeed the use of random projections U and V over extension fields might improve the probabilities to get the full minimal polynomial of the matrix [12, 18, 4]. As shown in this paper and in [8], σ-basis needs only a polynomial matrix multiplication implementation to work. In order to adapt current LinBox's implementation to extension field, we will use the same technique as [7]: first use Kronecker substitution to transform the extension field polynomial representation to an integer representation ; then use a Chinese remaindered version of the polynomial matrix multiplication to recover the resulting matrix polynomial over $\mathbf{Z}$; and finally convert back the integers using e.g. the REDQ inverse operation of [6].

The SpMV implementation also needs further work and other directions to be explored. For instance, we need to have dedicated implementations in $\mathbf{Z}/2\mathbf{Z}$ where $\mathbf{x}$ and $\mathbf{y}$ can be compressed. More formats, including dense submatrices, have yet to be explored, which is linked to spending some more time on pre-processing the matrix: for instance the use of Metis[6] for partitioning and reordering A would also improve the performance. It will be interesting to deal with matrices such that A and A^t cannot be simultaneously stored ([2]). This problem indeed occurs on GPU's where on-chip memory is very limited. Finally, we will also provide multi-GPU and hybrid GPU/CPU implementations.

5. REFERENCES

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[2] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, New York, NY, USA, 2009. ACM.

[3] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[4] L. Chen, W. Eberly, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications*, 343-344:119–146, 2002.

[5] D. Coppersmith. Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, Jan. 1994.

[6] J.-G. Dumas. Q-adic transform revisited. In D. Jeffrey, editor, *Proceedings of the 2008 ACM International Symposium on Symbolic and Algebraic Computation, Hagenberg, Austria*, pages 63–69. ACM Press, New York, July 2008.

[7] J. G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and*

[6]`http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

algebraic computation, pages 63–74, New York, NY, USA, 2002. ACM.

[8] J.-G. Dumas, P. Giorgi, P. Elbaz-Vincent, and A. Urbańska. Parallel computation of the rank of large sparse matrices from algebraic k-theory. In S. Watt, editor, *PASCO 2007*, pages 43–52. Waterloo University, Ontario, Canada, July 2007.

[9] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.

[10] J. v. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[11] P. Giorgi, C.-P. Jeannerod, and G. Villard. On the complexity of polynomial matrix computations. In R. Sendra, editor, *Proceedings of the 2003 ACM International Symposium on Symbolic and Algebraic Computation, Philadelphia, Pennsylvania, USA*, pages 135–142. ACM Press, New York, Aug. 2003.

[12] E. Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, Apr. 1995.

[13] E. Kaltofen and A. Lobo. Factoring high-degree polynomials by the black box Berlekamp algorithm. In ACM, editor, *ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation: July 20–22, 1994, Oxford, England, United Kingdom*, pages 90–98, pub-ACM:adr, 1994. ACM Press.

[14] E. Kaltofen and B. D. Saunders. On Wiedemann's method of solving sparse linear systems. In *Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)*, volume 539 of *Lecture Notes in Computer Science*, pages 29–38, Oct. 1991.

[15] W. J. Turner. A block Wiedemann rank algorithm. In J.-G. Dumas, editor, *Proceedings of the 2006 ACM International Symposium on Symbolic and Algebraic Computation, Genova, Italy*, pages 332–339. ACM Press, New York, July 2006.

[16] P. Tvrdik and I. Simecek. A new approach for accelerating the sparse matrix-vector multiplication. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, 0:156–163, 2006.

[17] F. Vazquez, E. M. Garzon, J. A. Martinez, and J. J. Fernandez. The sparse matrix vector product on GPUs. *Technical Report*, June 2009.

[18] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Technical Report 975–IM, LMC/IMAG, Apr. 1997.

[19] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005.

[20] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In L. T. Yang, O. F. Rana, B. D. Martino, and J. Dongarra, editors, *HPCC*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816, pub-SV:adr, 2005. Springer-Verlag Inc.

[21] D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, Jan. 1986.

Parallel Gaussian Elimination for Gröbner bases computations in finite fields

Jean-Charles Faugère
INRIA, Paris-Rocquencourt Center, SALSA Project
UPMC, Univ Paris 06, LIP6
CNRS, UMR 7606, LIP6
UFR Ingénierie 919, LIP6
Case 169, 4, Place Jussieu, F-75252 Paris
Jean-Charles.Faugere@inria.fr

Sylvain Lachartre
Thales Communications - Laboratoire Chiffre
160, boulevard de Valmy
92700 Colombes
Sylvain.Lachartre@fr.thalesgroup.com

ABSTRACT

Polynomial system solving is one of the important area of Computer Algebra with many applications in Robotics, Cryptology, Computational Geometry, etc. To this end computing a Gröbner basis is often a crucial step. The most efficient algorithms [6, 7] for computing Gröbner bases [2] rely heavily on linear algebra techniques. In this paper, we present a new linear algebra package for computing Gaussian elimination of Gröbner bases matrices. The library is written in C and contains specific algorithms [11] to compute Gaussian elimination as well as specific internal representation of matrices (sparse triangular blocks, sparse rectangular blocks and hybrid rectangular blocks). The efficiency of the new software is demonstrated by showing computational results fr well known benchmarks as well as some crypto-challenges. For instance, for a medium size problem such as Katsura 15, it takes 849.7 sec on a PC with 8 cores to compute a DRL Gröbner basis modulo $p < 2^{16}$; this is 88 faster than Magma (V2-16-1).

Categories and Subject Descriptors

I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Algorithms: Algebraic algorithms*; F.2.2 [**Theory of Computation**]: Analysis of algorithms and problem complexity—*Non numerical algorithms and problems: Geometrical problems and computation*; D.4.6 [**Software**]: Operating Systems—*Security and Protection: Cryptographic controls*

General Terms

Algorithms.

Keywords

Polynomial systems solving, Gröbner bases, Gaussian Elimination, High Performance Linear Algebra, Cryptography, Multi-core Programming.

1. INTRODUCTION

The most efficient algorithms [6, 7] for computing Gröbner bases [2] rely heavily on linear algebra techniques. More precisely, the main cost in Gröbner bases computation is the Gaussian reduction of matrices constructed from polynomials of the ideal generated by the input equations. The matrices generated by these algorithms have unusual properties: sparse, almost block triangular and not necessary full rank. Moreover, most of the pivots are known at the beginning of the computation.

Unfortunately, although `M4RI` [1] has good performances in $\mathbb{F}_2$, the best linear algebra packages such as `ATLAS` [13], `LinBox` [4], `FFLAS-FFPACK` [5] or `Sage` [12] are very efficient for dense linear algebra, but not tuned for F_4/F_5 matrices in word-size prime fields. In [11], we have presented a dedicated efficient algorithm for computing Gaussian elimination of such matrices. The main idea consists in decomposing the initial matrix in four submatrices obtained from both lists of pivot and non pivot rows and columns, and to treat them specifically. To benefit as much as possible from the cache memory, each matrix is split into small blocks and the reduction relies on three elementary block operations. To deal with the specific structures of the matrices occurring in a Gröbner basis computation we distinguish three block formats: sparse triangular blocks, sparse rectangular blocks and hybrid rectangular blocks (the internal representation can be sparse or dense, in adequation with the eventual rows densification occurring during the computation). At the end of this paper we report some timings and speedup to show the efficiency of the new library and to compare with existing linear algebra packages.

2. GRÖBNER BASES AND LINEAR ALGEBRA

Notions about Gröbner bases and how to compute them using linear algebra are not described here (see [3, 6, 7] for instance).

A list of polynomials $[f_1, \cdots, f_s]$ can be represented by a matrix as follows: columns correspond to all the monomials occurring in the polynomials (sorted with respect to a monomial ordering), and each row contains coefficients of a polynomial with respect to these monomials. The *leading* coefficient of a row denotes the column index of its first non zero coefficient.

$$\left\{\begin{array}{lcl} f_1 & = & \sum_{i=1}^{k} \alpha_{1,i}\, m_i \\ f_2 & = & \sum_{i=1}^{k} \alpha_{2,i}\, m_i \\ & \vdots & \\ f_s & = & \sum_{i=1}^{k} \alpha_{s,i}\, m_i \end{array}\right. \longrightarrow \begin{array}{c} \\ f_1 \\ f_2 \\ \vdots \\ f_s \end{array} \begin{array}{c} \begin{array}{cccc} m_1 & m_2 & \dots & m_k \end{array} \\ \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \dots & \alpha_{1,k} \\ \alpha_{2,1} & \alpha_{2,2} & \dots & \alpha_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{s,1} & \alpha_{s,2} & \dots & \alpha_{s,k} \end{pmatrix} \end{array}$$

To summarize, a Gröbner basis computation can be seen as a sequence of Gaussian eliminations of such matrices. In the next section, we present a new Gaussian elimination algorithm in which operations are performed in a different order. For that purpose, two types of columns in the matrix are distinguished: *pivot columns* (in which a row has its leading term), and *non pivot columns*. Similarly, a non null row chosen to reduce others is called a *pivot row*. The *column pivot set* (resp. *row pivot set*) is the set of all pivot column (resp. row).

The new algorithm use three elementary matrix operations:

- `Trsm`[1] : $Y \leftarrow X^{-1}Y$,
- `Axpy`[2]: $Y \leftarrow AX + Y$,
- `Gauss` : *classical* Gaussian elimination.

2.1 Structure of the matrices

The matrices occurring in the Gröbner basis computation have the following common properties:

- *sparse*: in degree δ a shift of a homogeneous degree d polynomial with n variables has less than $\binom{n+d-1}{d}$ non zero coefficients for $\binom{n+\delta-1}{\delta}$ total columns. For instance, if $d = 3$, $n = 5$ and $\delta = 10$, then the average density for this line is about 3.5%),
- several rows are monomial multiples of the same polynomial f: $(m_1 f, m_2 f, \dots, m_k f)$,
- the matrices are not necessary full rank (this is the main difference between F_4 and F_5).
- *almost block triangular*: each matrix is constructed by pairwise combinations from a set of polynomials with distinct leading terms (*S-polynomial*), to exhibit new polynomials with new leading terms.

The last point provides the predetermination of a part of the pivot columns. The efficiency of our new algorithm rely on this knowledge.

3. SKETCH OF THE SEQUENTIAL ALGORITHM

This algorithm has been introduced in [11] and it takes into consideration sparsity and almost block triangular shape, with different treatments and representations for the preselected pivot rows and columns, and the other ones. It inputs a $n_0 \times m_0$ matrix M_0 and performs its Gaussian elimination.

3.1 Analysis

The first stage consists in looking for columns clearly identified as pivot. For that purpose, it is enough to sweep the row leading terms. The list of the corresponding columns indices is called C_{piv}, and is of size N_{piv}. Then, one pivot row is chosen from several candidates (in fact all the rows which have the same leading index), to obtain the list R_{piv}, also of length N_{piv} : the coordinates of the i-th pivot will be $\left[R_{piv}[i], C_{piv}[i]\right]$. The list of non pivot rows (resp. columns) is denoted $\overline{R_{piv}}$ (resp. $\overline{C_{piv}}$).

[1]Trsm: TRiangular Solve with Multiple right-hand sides
[2]Axpy: "A X plus Y"

3.2 Decomposition into submatrices

M_0 can be decomposed in 4 submatrices A, B, C, D using the row and column pivot lists:

- A is made from the elements indexed by R_{piv} and C_{piv} (upper triangular $N_{piv} \times N_{piv}$ matrix with diagonal coefficients equal to 1).
- B of dimensions $N_{piv} \times (m_0 - N_{piv})$ contains the elements indexed by R_{piv} and $\overline{C_{piv}}$.
- C is a $(n_0 - N_{piv}) \times N_{piv}$ matrix built from the elements indexed by $\overline{R_{piv}}$ and C_{piv}. Its rows are sorted by increasing leading term indices and with leading coefficient equal to 1.
- D is obtained from the $(n_0 - N_{piv}) \times (m_0 - N_{piv})$ remaining elements (indexed by $\overline{R_{piv}}$ and $\overline{C_{piv}}$).

Figure 1 shows these four submatrices with their respective dimensions.

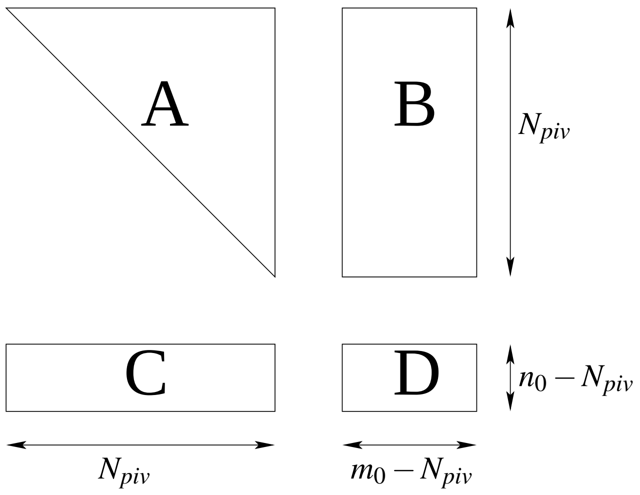


Figure 1: ABCD decomposition

3.3 Pivot row reduction (Trsm)

The third step of the algorithm consists in reducing the pivot rows by themselves. From linear algebra point of view, this means computing $B \leftarrow A^{-1}B$ since A is non singular (upper triangular with 1s on the diagonal). This operation is a basis change for the non pivot columns: it computes their expression in the vector space generated by the pivot columns. Each submatrix is treated differently: A is only read so it remains sparse, whereas the matrix B is accessed in read/write mode, so its density may increase. When B is a dense matrix, this computation can be made "in place" to save memory. At this step, the matrix M_0 is equivalent to:

$$M_0 \sim \left(\begin{array}{c|c} Id & A^{-1}B \\ \hline C & D \end{array}\right)$$

3.4 Non pivot rows reduction (Axpy)

Once the pivot rows are reduced, the non-pivot rows must be reduced by these new pivot rows by computing $D \leftarrow D - CB$ (here B denotes the *new* matrix $B \leftarrow A^{-1}B$) and C is set to zero, since all its coefficients are reduced by those of A. M_0 is now equivalent to (wrt. initial matrices A, B, C and D):

$$M_0 \sim \left(\begin{array}{c|c} Id & A^{-1}B \\ \hline 0 & D - CA^{-1}B \end{array}\right)$$

3.5 New pivot row computation (Gauss)

At this point, all the rows of M_0 have been reduced by pivot rows. The next step is to look and find new pivots in the matrix D, with a *Gaussian elimination* (row version of the classical and well-known algorithm): $D \leftarrow \texttt{Gauss}(D)$. Note that the leading terms of D are not necessary equal to 1 anymore, and some field inversions may be further required. Now:

$$M_0 \sim \left(\begin{array}{c|c} Id & A^{-1}B \\ \hline 0 & \texttt{Gauss}(D - CA^{-1}B) \end{array}\right)$$

3.6 Reconstruction

At last, the final matrix Gauss(M_0) is reconstructed from rows and pivots lists R_{piv} and C_{piv}, and from new matrices B and D.

REMARK 1. *The final matrix is not in row reduced echelon form. To obtain $rref(M_0)$ a second iteration of this new algorithm must be applied (see [11] for details) : the last step (3.5) gives two new lists of pivot rows and columns (so a new decomposition and the* `Trsm` *and* `Axpy` *steps can be performed once again before reconstructing the final rref matrix).*

4. PARALLEL IMPLEMENTATION

Operations on $\overline{C_{piv}}$ columns are independent, so they can be performed in parallel. In this section, we present the data structures we used to implement a parallel version of the new algorithm.

4.1 Data structures

We take into account the architecture with last generation processors, while respecting the structure of matrices from F_4/F_5 algorithms. To benefit from the cache processor memory, and to maintain an optimal stream of data, matrices are reorganized by row and column blocks. We use three block matrix formats : *sparse*, *dense* and *hybrid* (rows are stored in sparse or dense format according to their density). Moreover, three blocks sizes have to be fixed:

- K_{AB} (resp. $K_{A_{rl}}$): row and column block (resp. incomplete block) size of matrix A (common block size of columns of A and C, and rows of B),
- K_{CX} (resp. $K_{C_{rl}}$): row block (resp. incomplete block) size of matrices C and D,
- K_{BY} (resp. $K_{B_{rc}}$): column block (resp. incomplete block) size of the matrices B and D,

where $K_{A_{rl}}$, $K_{C_{rl}}$ and $K_{B_{rc}}$ are the dimensions of incomplete blocks, respectively equal to:

$$\begin{cases} K_{A_{rl}} & \equiv & N_{piv} \mod K_{AB}, \\ K_{C_{rl}} & \equiv & n_0 - N_{piv} \mod K_{CX}, \\ K_{B_{rc}} & \equiv & m_0 - N_{piv} \mod K_{BY}. \end{cases}$$

Before giving a more formal description, figure 2 presents the global block layout: the numbered blocks in matrices and the dotted arrows of a block inner row symbolize the storage order of the elements in the memory.

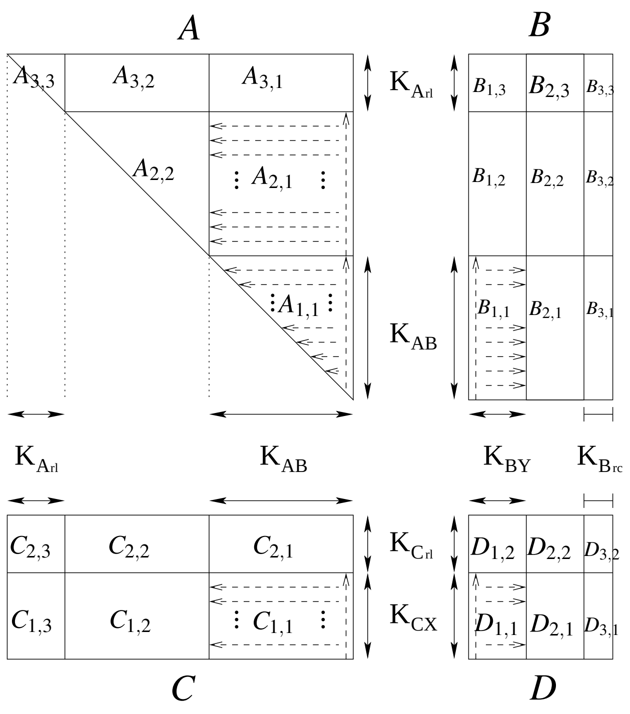


Figure 2: Matrices A, B, C and D block division

4.2 Block inner operations

This section deals with operations within a block. We distinguish three block formats:

1. *Sparse triangular block format*: applies to triangular blocks of the matrix A. It uses three lists: A_{val}, A_{pos} and A_{nb}, which represent respectively the values, the positions and the number of non zero elements in each row of the matrix A. Elements as well as rows are sorted by increasing order, from bottom to top. Row leading coefficients (equal to 1) and the last row of the block are not stored.
2. *Sparse rectangular block format*: this is the format of the rectangular blocks of matrices A and C. Three lists are also necessary to store the value, the position and the number of non-zero elements of each row in the block. Rows are sorted by decreasing order, from bottom to top. For the blocks of A, the positions of the non-zero elements are decreasing, from right to left, while for C, these are written in increasing order, from left to right.
3. *Hybrid rectangular block format*: used for the blocks of matrices B and D. Rows are stored in *hybrid* format: their representation is sparse or dense, according to the number of non-zero elements. Rows are ordered by decreasing indices, from bottom to top, while the row elements by growing indices, from left to right.

The layout of blocks in matrices is one of the following three formats:

1. *Block format of sparse triangular matrix:* uses sparse triangular and sparse rectangular blocks. Blocks are ordered by rows from right to left, and from bottom to top. Rectangular blocks have K_{AB} rows while triangular blocks have $K_{AB} - 1$

rows (since the leading coefficients, always equal to 1, are not stored).

2. *Block format of sparse rectangular matrix:* only contains *rectangular sparse blocks* stored by rows. The block layout is the same that the *sparse triangular matrix* format.

3. *Block format of hybrid rectangular matrix:* consists of hybrid rectangular blocks ordered from top to bottom, and from left to right.

EXAMPLE 1. *To illustrate each one of these three formats, we present three matrices A, B and C of dimensions $n \times m$ with block size K and density threshold d (for better legibility, a zero row or column is represented by the the empty set $\emptyset$ for value and position, and 0 for the number). For hybrid blocks, a threshold density d is chosen to determine whether a row has a sparse or a dense representation (ie. if the density is greater than the threshold):*

- *Sparse triangular block matrix format $n = m = 5$, $K = 2$:*

$$A = \left(\begin{array}{c|cc|cc} 1 & 5 & 2 & 0 & 0 \\ \hline 0 & 1 & 4 & 8 & 3 \\ 0 & 0 & 1 & 6 & 0 \\ \hline 0 & 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

$$A_{val} \quad \begin{array}{|c|ccc|c|c|cc|} \hline 7 & 6 & 3 & 8 & 4 & \emptyset & 2 & 5 \\ \hline \end{array}$$

$$A_{pos} \quad \begin{array}{|c|ccc|c|c|cc|} \hline 1 & 2 & 1 & 2 & 1 & \emptyset & 1 & 2 \\ \hline \end{array}$$

$$A_{nb} \quad \begin{array}{|c|cc|c|c|c|} \hline 1 & 1 & 2 & 1 & 0 & 2 \\ \hline \end{array}$$

- *Sparse rectangular block matrix format, $n = 3$, $m = 5$ and $K = 2$:*

$$C = \left(\begin{array}{c|cc|cc} 8 & 6 & 1 & 9 & 0 \\ \hline 4 & 0 & 0 & 5 & 0 \\ 7 & 0 & 0 & 2 & 3 \end{array}\right)$$

$$C_{val} \quad \begin{array}{|ccc|c|cc|c|cc|c|} \hline 3 & 2 & 5 & \emptyset & 7 & 4 & 9 & 1 & 6 & 8 \\ \hline \end{array}$$

$$C_{pos} \quad \begin{array}{|ccc|c|cc|c|cc|c|} \hline 1 & 2 & 2 & \emptyset & 1 & 1 & 2 & 1 & 2 & 1 \\ \hline \end{array}$$

$$C_{nb} \quad \begin{array}{|cc|cc|cc|c|c|c|} \hline 2 & 1 & 0 & 0 & 1 & 1 & 1 & 2 & 1 \\ \hline \end{array}$$

- *Hybrid rectangular block matrix format, $n = 5$, $m = 3$, $K = 2$ and $d = 50\%$:*

$$B = \left(\begin{array}{cc|c} 0 & 2 & 5 \\ \hline 4 & 0 & 0 \\ 7 & 1 & 0 \\ \hline 0 & 0 & 3 \\ 6 & 8 & 0 \end{array}\right)$$

$$B_{val} \quad \begin{array}{|cc|ccc|c|c|c|c|} \hline 6 & 8 & 7 & 1 & 4 & 2 & 3 & \emptyset & 5 \\ \hline \end{array}$$

$$B_{pos} \quad \begin{array}{|c|c|c|c|c|c|} \hline \emptyset & 1 & 2 & \emptyset & \emptyset & \emptyset \\ \hline \end{array}$$

$$B_{nb} \quad \begin{array}{|cc|cc|c|cc|cc|c|} \hline 2 & 0 & 2 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

This ordering is in perfect adequacy with the *double spacial and temporal principle* (see [13] for example) and so, benefits from the *cache memory* (small and fast memory taking advantage of two principles : a program is more likely to spend its time executing code around the same set of instructions, and tend to run in loops repeating the same instructions).

Algorithm 1 performs $B \leftarrow A^{-1}B$ between *sparse triangular block* A and *hybrid rectangular block* B ($D \leftarrow D - CB$ block algorithm follows the same philosophy). It uses a dense temporary row denoted `Temp` (rows must be converted from hybrid to dense format when copying rows from B to `Temp`, and from dense to hybrid format when updating B from `Temp`). Sparse or dense linear algebra (Axpy) is used according to the density of hybrid rows of B.

Algorithm 1: $B \leftarrow A^{-1}B$: block "hybrid" version

```
Inputs   : sparse triangular block block $A$
           hybrid rectangular block $B$
Output   : hybrid rectangular block $B = A^{-1} B$
Local    : Temp is a $m_0 - N_{piv}$ temporary dense row
Notation: $X[i,*]$ is the i-th row of $X \in \{A, B\}$
   /* $A$ rows loop                                                  */
1  for $i \leftarrow N_{piv} - 1$ to 1 do
       /* Hybrid format to dense format                              */
2      Temp ← Hybrid2Dense($B[i,*]$)
       /* $A$ i-th row loop ($A_{nb}[i] - 1$ elements)                */
3      for $j \leftarrow 2$ to $A_{nb}[i]$ do
4          $Av \leftarrow A_{val}[i,j]$, $Ap \leftarrow A_{pos}[i,j]$
5          if Density($B[Ap,*]$) $\leq Threshold$ then
               /* Sparse :  Temp ← Temp $- Av * B[Ap,*]$              */
6              Temp ← SparseAxpy(Temp,$Av$,$-1$,$B[Ap,*]$)
7          else
               /* Dense :  Temp ← Temp $- Av * B[Ap,*]$               */
8              Temp ← DenseAxpy(Temp,$Av$,$-1$,$B[Ap,*]$)
       /* Dense format to hybrid format                              */
9      $B[i,*]$ ← Dense2Hybrid(Temp)
10 return $B$
```

4.3 Block outer operations

The outer operations are performed on matrix blocks: each operation $B \leftarrow A^{-1}B$ and $D \leftarrow D - CB$ uses *block hybrid* algorithms. It is also possible to use a temporary dense block to store the results of the partial block products.

4.4 Block hybrid Gaussian elimination

The search of new pivots (see 3.5) has to be adapted to the *block hybrid* format of the matrix D (Gauss algorithm operating on *hybrid blocks*). Here, the Gaussian elimination is performed on successive blocks (by increasing indices) of the new matrix D obtained in 3.4. A $\#\overline{R_{piv}} \times \#\overline{R_{piv}}$ matrix P, equivalent to a *pseudo inverse*, is introduced to keep a track of the successive row operations. In the i-th stage, the i-th block D_i is updated by left-product by P, and then Gaussian elimination is performed on $D_i \mid P$ (D_i concatenated with P), from rows of indices greater than the partial rank $r_D^{(i)}$. Note that a temporary block is used to store the rows of D_i and P which have to be reduced.

Initially, P is equal to the identity matrix. The Gaussian reduction of the first block concatenated with P is computed. Then, in the i-th stage, the i-th block is updated by a simple left matrix multiplication by P, and then, the a Gaussian reduction is performed on this block concatenated with P. We denote $nz^{(i)}$ the number of non-null rows in the i-th block D_i. Identically, $r_D^{(i)}$ is the rank of the i-th block after Gaussian elimination.

On figure 3, the matrix is represented after the reduction of the first block and has "$nz^{(1)}$" non-null rows. After the first Gaussian block reduction, the first block contains the up-triangular matrix of rank $r_D^{(1)}$. The $nz^{(1)}$ first rows of P contain the linear operations needed by the Gaussian reduction of the first block.

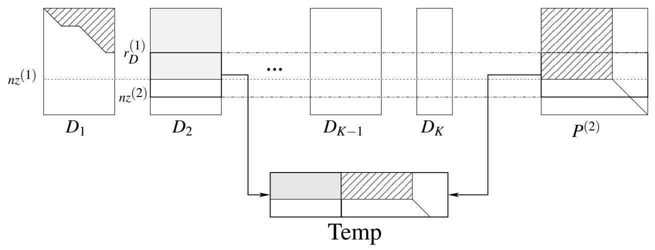


Figure 3: Gaussian block reduction

Then, the non null rows from index $\left(r_D^{(i)}+1\right)$ of D_2 and of P are copied in the temporary block. Finally, the temporary block is reduced by Gaussian elimination, and the submatrices are updated. The temporary rank is then denoted $r_D^{(2)}$.

This process is iterated to obtain the Gaussian reduction of the matrix, which final rank is denoted r_D. Therefore, $\text{rank}(M_0) = N_{piv} + r_D$. The efficiency relies on the number N_{piv} (trivial pivots in M_0): if $\#\overline{R_{piv}} = n_0 - N_{piv}$ is small with respect to n_0, the cost of this hybrid Gaussian block elimination is negligible both in time and memory, comparing to the whole process cost.

Algorithm 2 present this hybrid Gaussian algorithm. The function `FistNonZeroRow`(l,M) returns the index of the first non-null row in the list l of rows of the matrix M. The function `Update`(Temp) copies the temporary rows of Temp in the corresponding rows of D_i and P in a hybrid format. At the end of this algorithm, both matrices P and Temp can be freed from memory.

4.5 Parallelization

During the computation of $B \leftarrow A^{-1}B$ thus $D \leftarrow D - CB$, the operations on the columns of matrices B and D are independent. They can be realized in parallel. For that purpose, matrices B and D must be considered from columns blocks point of view (noted B_i and D_i), and the two elementary parallelizable operations are:

- `Trsm`(i): inputs the block index i and outputs
$$B_i \leftarrow A^{-1}B_i,$$
- `Axpy`(i): inputs the block index i and outputs
$$D_i \leftarrow D_i - CB_i.$$

The hybrid Gaussian elimination algorithm is applied to D_i (denoted `Gauss`(i)) to search for new pivots (after both previous reductions).

During the whole process, Gaussian elimination must be performed as soon as possible. So, we define priority rules between

Algorithm 2: Block hybrid Gaussian elimination

```
Input   : Block hybrid matrix $D$ (dimension $n_D \times m_D$).
Outputs: Block hybrid matrix Gauss($D$) and its rank $r_D$.
   /* Init parameters                                              */
1  $P \leftarrow Id_{n_D}, r_D^{(1)} \leftarrow 0, N \leftarrow \lceil m_D/K_{BY} \rceil$
   /* D blocks loop                                                */
2  for $i \leftarrow 2$ to $N-1$ do
3      $nz^{(i)} \leftarrow$ FirstNonZeroRow($\{r_D^{(i-1)}+1,\dots,n_D\},D_i$)
4      Temp $\leftarrow$ Gauss $\left(\text{SubMatrix}\left(\{nz^{(i-1)},\dots,nz^{(i)}\},D_i|P\right)\right)$
5      $r_D^{(i)} \leftarrow r_D^{(i-1)} +$ Rank(Temp)
6      $(D_i,P) \leftarrow$ Update(Temp)
   /* Last block of D                                              */
7  Temp $\leftarrow$ Gauss $\left(\text{SubMatrix}\left(\{r_D^{(N-1)}+1,\dots,n_D\},D_N\right)\right)$
8  $r_D \leftarrow r_D^{(N-1)} +$ Rank(Temp)
9  $D_N \leftarrow$ Update(Temp)
10 return $D$ and $r_D$
```

the three operations `Trsm`, `Axpy` and `Gauss`. Four priority constraints and synchronization points (denoted S_i for i from 1 to 4) are introduced for the parallel algorithm (see figure 4):

- S_1 (from `Analysis` to `Trsm`): no constraint of synchronization,
- S_2 (from `Trsm` to `Axpy`): to compute `Axpy`(i), the computation of `Trsm`(i) must be completed,
- S_3 (from `Axpy` to `Gauss`): to process the reduction `Gauss`(i), `Axpy`(i) must be completed as well as the operation `Gauss`(j) for j between 1 and $i-1$,
- S_4 (from `Axpy` step to the reconstruction step): all the operations of type `Axpy` must be completed.

To keep track of all the operations on reduced blocks by each of the operations, the list of remaining tasks is shared by all processors.During its update, we make sure that no other processor has access to this *critical section*. For that purpose, we use *Mutex* (MUTual exclusion). Algorithm 3 presents a way of parallelizing the computation in order to lower the latency. It uses four lists:

- Function: list of the three block operations (`Trsm`, `Axpy` and `Gauss`),
- Todo: list of the lists of not treated yet block indices for each of the three functions,
- Done: list of the block indices for which the three operations have been performed,
- Pr: list of priorities of each function (since Gauss is sequential, it must be computed as soon as possible, so its priority is 1 and the priority of Axpy is 2; Trsm is the function with less priority).

This algorithm is executed by all the threads and ends when the blocks of all the matrices have been treated by the three operations. At the beginning of the while loop, the thread looks for a task (searching first in the most priority list – ie. Todo_3, then Todo_2, etc – and denoting $ind = \text{Todo}_{Pr[i]}$ this block index), locks the mutex to update Todo (ie. remove ind from Todo$[i]$: the chosen task has no longer to be treated by the other threads), and performs the

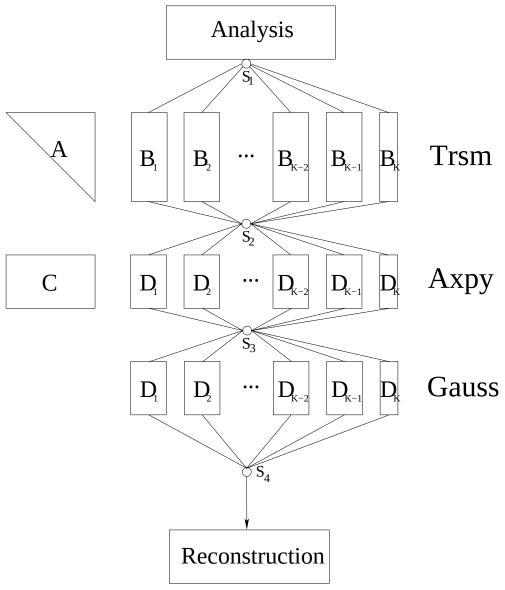


Figure 4: New Gaussian algorithm (parallel version)

Algorithm 3: Parallel Gaussian algorithm

```
Inputs     : matrices $A$, $B$, $C$ and $D$
Outputs    : matrices $B$ and $D$ after reduction
Notations: Todo: lists of blocks to be treated by functions,
              Pr: list of function priorities.
1  Todo ← [ $[1,\dots,K]$, [ ], [ ] ], Done ← [ ]
2  Function ← [Trsm, Axpy, Gauss], Pr ← $[3,2,1]$
   /* Something to do                                  */
3  while Done $\neq [1,\dots,K]$ do
       /* search a task from high to low priority       */
4      for $i \leftarrow 1$ to 3 do
5          if $\mathrm{Todo}_{\mathrm{Pr}[i]} \neq [\,]$ then
6              Lock()
7              $ind \leftarrow \mathrm{Todo}_{\mathrm{Pr}[i]}[1]$
8              $\mathrm{Todo}_{\mathrm{Pr}[i]} \leftarrow \mathrm{Todo}_{\mathrm{Pr}[i]} \setminus [ind]$
9              Unlock()
               /* The computation is performed           */
10             Function$[i](ind)$
11             Lock()
12             if $i \leq 2$ then
                   /* Next operation must be performed on
                      this block                           */
13                 $\mathrm{Todo}_{\mathrm{Pr}[i+1]} \leftarrow \mathrm{Sort}(\mathrm{Todo}_{\mathrm{Pr}[i+1]} \cup [ind])$
14             else
                   /* All the operations are done          */
15                 Done ← Done $\cup [ind]$
16             Unlock()
```

computation Function$[i](ind)$. If $i \leq 2$, *ind* is added to the next list Todo[i + 1], else the *ind*-th block is added to Done (nothing to do with it anymore). Then, the thread goes on until all the blocks have been treated by the three operations.

5. PRACTICAL EXPERIMENTS

We have implemented a small finite field version ($\mathbb{F}_p$ with $3 \leq p \leq 65521$) of this new algorithm in C language (approximately 15000 lines of code) using POSIX threads.

5.1 Comparison with existing linear algebra packages

First, we compute the row echelon form (in [11] we have also described a `Rref` algorithm to compute a row echelon form of matrix) of small matrices occurring in some Gröbner bases applications. We compare the computations in $\mathbb{F}_{65521}$ with several linear algebra tools: Maple 13 (function `RowReduce` from LinearAlgebra and Modular packages), Magma 2.16.1 (function `NullspaceOfTranspose` on sparse matrices), Sage 3.0.5 (echelon_form on sparse matrices) and Linbox 1.1.6 (`rowReducedEchelon` on SparseMatrix), on the six matrices:

Name	Dimension	Density	Rank
robot	404×302	12.39%	262
katsura7	694×738	7.44%	611
f855	2456×2511	2.78%	2331
cyclic8	4562×5761	9.37%	3903
katsura12	18285×19607	10.50%	15810
cyclic9	72552×93913	0.70%	71872

The tests are run on a pc with two Intel Xeon E5420 processors (with four 2.5 GHz cores each), and 6 Go of RAM, and obtain the following table (*MT* refers to the case of a *memory trash*):

Name (version)	New library	Maple (13)	Sage (3.0.5)	Magma (2.16.1)	Linbox (1.1.6)
robot	<0.1	6.4	2.4	<0.1	<0.1
katsura7	<0.1	40.8	20.92	0.2	0.2
f855	<0.1	841.2	257.11	3.3	4.3
cyclic8	**1.8**	$> 10^5$	$> 10^5$	54.9	33.0
katsura12	**28.5**	MT	MT	1036.81	1166.8
cyclic9	**46.6**	MT	MT	MT	MT

Although these matrices are sparse, for Maple and Sage dense linear algebra is more efficient. Our `Rref` version is more efficient (wrt. to memory and time) than the other tools.

At last, the results of the parallel version of the new algorithm using POSIX threads:

Name	Seq. (s)	Thread number / SpeedUp				
		1	2	4	8	12
cyclic8	**1.8**	1.0	1.8	3.1	**4.7**	4.4
cyclic9	**46.6**	1.0	1.9	3.4	**5.7**	5.4

Note that with two threads, latency periods are almost null, both processors are used at full capacity. The best real times are obtained with eight threads using the eight cores of the machine. However, sequential hybrid last blocks computations and/or bus memory engorgement prevent from optimal performances.

5.2 Comparison with existing Gröbner bases tools

All the timings given in this section are in elapsed seconds and are obtained using our library on a 64 bit Intel Xeon CPU X5570 @ 2.93GHz with 8 cores.

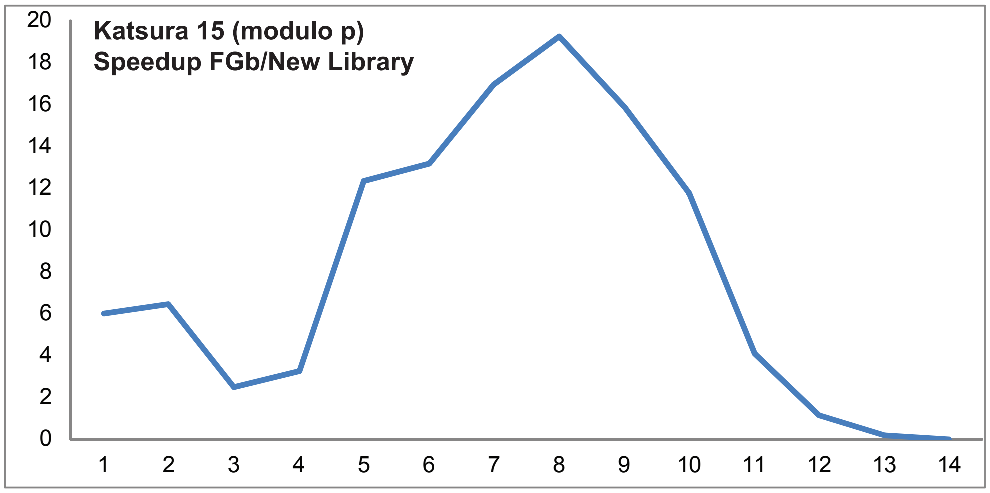


Fig.5.2: relative speedup for the Katsura 15 problem over $\mathbb{F}_{65521}$
(the abscissa corresponds to the stage of the GB computation and the ordinate to the speedup).

The goal now is to try to estimate the real speedup that we can achieve using the new library. In contrast with the previous subsection we have thus to perform Gaussian elimination on several matrices. To start with a well known benchmark we run our new library on the Katsura n problem [10]: since this system is a set of n quadratic equations we know that we have to perform $n+1$ Gaussian eliminations (this is the Macaulay bound for regular systems). In figure 5.2, we compare the new implementation with our reference library FGb. The conclusion is that the new library is always more efficient than the original implementation in FGb except for the last two computations: in that cases the matrices are *quasi-triangular* (triangular with few more rows) the new algorithm is not optimal (the cost of `Trsm` is too important with respect to a classical Gaussian elimination performed in FGb). The same phenomenon occurs for the steps 3 and 4 and that is why the speedup decreases.

In the current state of the implementation we have to devise the following strategy: by default to perform Gaussian elimination we call the new library except when the matrix is *quasi-triangular* (there is a threshold to find). When the matrix is *quasi-triangular* we call the old sequential implementation. Note that in practice the previous restriction is not a big deal: the CPU needed to perform Gaussian elimination on the first/last matrices occurring in the computation is negligible compared with the total CPU time. In the rest of the paper, we assume that we always apply this strategy.

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1042 x 2135	0.02	0.01	2.00	0.0	2.4
3827 x 6207	0.29	0.06	4.83	0.3	4.7
10014 x 14110	2.10	0.33	6.36	1.8	5.2
19331 x 25143	9.30	1.27	7.32	7.6	6.0
28447 x 35546	23.36	2.87	8.14	18.1	6.3
34501 x 42315	36.38	4.5	8.08	29.4	6.4
38165 x 46265	34.79	5.78	6.02	38.0	6.5
39590 x 47768	19.28	5.94	3.25	38.7	6.5
39965 x 48156	5.90	5.65	1.04	36.3	6.4
40035 x 48227	1.08	1.08	1.00	35.5	6.3
40042 x 48234	0.07	0.07	1.00	35.4	6.3
Total	191.69	27.56	**6.96**		

Katsura 13 modulo 65521 with 8 cores

5.2.1 *Katsura modulo p*

We present here the detailed results of the Katsura n problems for n from 13 to 16. In some table we also include a comparison between the sequential version of the library (`New Seq Library`) and the 8-cores version of the library (`New Seq Library (8)`). All the timings are in seconds.

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1333 x 2804	0.04	0.01	4.00	0.05	4.8
5559 x 9032	0.63	0.11	4.50	0.65	5.9
11683 x 18005	0.52	0.36	1.33	1.8	5.0
21717 x 30783	5.85	1.31	4.50	7.7	5.8
39001 x 50484	93.65	7.11	13.12	49.9	6.7
67933 x 82582	322.19	27.42	12.08	182.85	5.8
70411 x 85376	218.69	21.46	10.33	141.4	6.6
81277 x 97202	332.68	30.72	10.99	215.4	7.0
86547 x 102826	258.66	30.64	8.58	208.5	6.8
88417 x 104786	105.31	28.18	3.76	189.3	6.7
88874 x 105257	28.70	26.92	1.08	176.5	6.6
88954 x 105338	4.72	4.72	1.00	175.1	6.6
88962 x 105346	0.32	0.32	1.00	175.2	6.7
Total	1881.29	180.68	**10.55**		

Katsura 14 modulo 65521 with 8 cores

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1667x3608	0.05	0.01	6.00	0.1	8.8
7312x12257	1.40	0.21	6.43	1.2	5.6
17248x26575	2.05	0.82	2.46	4.6	5.6
32109x46154	9.36	2.85	3.25	17.3	6.1
60801x79831	289.77	23.39	12.33	177.4	7.6
114563x140832	830.56	62.93	13.17	422.8	6.7
142062x170248	1454.32	85.78	16.92	558.2	6.5
170221x201111	2351.63	121.53	19.26	858.9	7.1
187664x219868	2275.85	142.23	15.87	865.1	6.1
195325x227973	1513.69	127.6	11.75	871.4	6.8
197778x230530	533.53	129.95	4.06	790.8	6.1
198335x231102	133.15	115.03	1.14	760.1	6.6
198426x231194	20.40	20.40	1.00	729.8	6.5
198434x231202	1.25	1.25	1.00	738.6	6.5
Total	11948.14	849.68	**14.06**		

Katsura 15 modulo 65521 with 8 cores

Steps	Dimension	FGb	New library (8)	SpeedUp FGb/New
1	271 x 968	0	0	
2	2048 x 4565	0.08	0.02	4.00
3	9953 x 16839	2.58	0.38	6.79
4	23290 x 36757	3.86	1.50	2.57
5	45844 x 67046	18.70	6.25	2.99
6	83046 x 114252	108.55	23.96	4.53
7	160426 x 204782	3326.63	186.06	17.88
8	175286 x 214892	3822.91	194.53	19.65
9	328980 x 385905	11295.82	700.92	16.12
10	373624 x 432524	16441.15	890.49	18.46
11	401429 x 464523	19090.29	733.58	26.02
12	426807 x 491659	15294.66	728.41	21.00
13	437603 x 503003	8912.21	867.45	10.27
14	440754 x 506273	3035.39	622.11	4.88
15	441423 x 506958	603.01	595.60	1.01
16	441525 x 507061	84.23	84.23	1.00
17	441534 x 507070	4.84	4.84	1.00
Total		103180.96	5687.29	**18.14**

Katsura 16 modulo 65521 with 8 cores

We can deduce from the previous table that the new library is very efficient. Better results can still probably obtained since we have sometimes a maximal speedup of 26 and sometimes a much lower speedup.

5.2.2 Minrank

The Minrank problem is a fundamental linear algebra problem (generalisation of the eigenvalues problem) as was studied recently in Cryptology [8] or in Computer Algebra [9]. In that case, the polynomial system is a list of polynomials of degree 4.

Steps	Dimension	FGb	New library (8)	SpeedUp FGb/New
1	441 x 2002	0.47	0.17	2.76
2	1676 x 4231	0.70	0.10	7.00
3	3657 x 7058	2.06	0.31	6.65
4	5089 x 8985	4.54	0.53	8.57
5	6204 x 10265	4.88	0.85	5.74
6	6594 x 10700	2.06	0.87	2.37
7	6720 x 10835	0.63	0.63	1.00
8	6753 x 10869	0.14	0.14	1.00
9	6758 x 10874	0.02	0.02	1.00
Total		28	3.62	**7.73**

Minrank (9,7,4) with 8 cores

Steps	Dimension	FGb	New library (8)	SpeedUp FGb/New
1	784 x 5005	3.89	0.07	2.76
2	3145 x 10201	7.43	0.68	5.99
3	6989 x 16880	24.25	2.44	7.01
4	11160 x 23270	51.36	4.88	6.94
5	14947 x 28344	96.73	10.72	6.31
6	17421 x 31313	109.04	15.52	5.54
7	18420 x 32477	52.34	15.59	2.85
8	18810 x 32912	20.08	15.52	1.11
9	18936 x 33047	5.92	5.92	1.00
10	18969 x 33081	1.3	1.3	1.00
11	18974 x 33086	0.14	0.14	1.00
Total		512.32	72.78	**7.04**

Minrank (9,8,5) with 8 cores

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
784 x 5005	3.89	0.07	2.76	1.7	1.4
3145 x 10201	7.43	0.68	5.99	3.9	5.9
6989 x 16880	24.25	2.44	7.01	17.1	7.1
11160 x 23270	51.36	4.88	6.94	35.7	7.4
14947 x 28344	96.73	10.72	6.31	83.1	7.9
17421 x 31313	109.04	15.52	5.54	123.0	8.0
18420 x 32477	52.34	15.59	2.85	122.7	8.0
18810 x 32912	20.08	15.52	1.11	119.6	7.7
18936 x 33047	5.92	5.92	1.00	116.5	7.9
18969 x 33081	1.3	1.3	1.00	108.3	7.6
18974 x 33086	0.14	0.14	1.00	115.4	7.8
Total	512.32	72.78	**7.04**		

Minrank (9,8,5) with 8 cores

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1296 x 11440	25.26	7.19	3.51	0.0	1.3
5380 x 22400	55.48	4.16	13.34	23.1	5.6
12224 x 36784	225.41	14.74	15.29	106.9	7.3
21066 x 52502	567.18	46.77	12.13	322.2	6.9
30519 x 67094	1119.4	91.61	12.22	643.8	7.0
38109 x 77687	1724.84	192.08	8.98	1436.6	7.5
43162 x 84027	1808.62	259.87	6.96	2071.4	8.0
45441 x 86801	956.11	245.61	3.89	1953.4	8.0
46440 x 87965	420.85	264.91	1.59	1813.8	6.8
46830 x 88400	154.03	154.03	1.00	1732.6	7.8
46956 x 88535	45.79	45.79	1.00	1697.6	7.2
46989 x 88569	10.03	10.03	1.00	1695.2	5.8
46994 x 88574	1.11	1.11	1.00	1673.5	7.5
Total	8757.62	1337.90	**6.55**		

Minrank (9,9,6) with 8 cores

Even if the new library is less efficient on this example than for the Katsura *n* problem we observe a non linear speedup for huge computations. The 8-core version is also always 6 to 8 times more efficient than the sequential version showing that the parallelization of the algorithm is quite efficient.

5.2.3 Comparison with Magma 2.16.1

We compare now our new algorithm with a recent version of the F_4 implantation in Magma.

	F_4 Kat11	F_4 Kat12	F_4 Kat 13
Magma	19.5	151.2	1091.4
FGb	40.6	342.6	2550.65
New library	**2.85**	**19.45**	**149.6**

	F_5 Kat 12	F_5 Kat 13	F_5 Kat 14
Magma	151.2	1091.4	9460.35
FGb	32.8	191.7	1881.3
New library	**4.6**	**27,6**	**180,7**

6. CONCLUSIONS AND PERSPECTIVES

We have shown a parallelized algorithm to perform Gaussian elimination in in order to compute efficiently Gröbner bases. We have applied our implementation on real size and difficult problems (for instance the Minrank problem in Cryptology). Hence our approach is very effective for computing Gröbner bases on a multicore PC. Some work is still necessary to obtain a maximal speedup and to decrease the memory requirement of the new library.

Acknowledgements: The authors would like to thank Olivier Orcière for his helpful remarks.

7. REFERENCES

[1] M. Albrecht and G. Bard. *The M4RI Library – Version 20090409*. The M4RI Team, 2009.

[2] B. Buchberger. An Algorithmical Criterion for the Solvability of Algebraic Systems. *Aequationes Mathematicae*, 4(3):374–383, 1970. (German).

[3] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 7 1997.

[4] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. Saunders, W. J. Turner, and G. Villard. Linbox: A Generic Library For Exact Linear Algebra, 2002.

[5] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.

[6] J.-C. Faugère. A New Efficient Algorithm for Computing Groebner bases (F_4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, 1999.

[7] J.-C. Faugère. A new efficient algorithm for computing Groebner bases without reduction to zero F_5. In *Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 2002.

[8] J.-C. Faugère, F. Levy-dit Vehel, , and L. Perret. Cryptanalysis of Minrank. In D. Wagner, editor, *Advances in Cryptology CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 280–296, Santa-Barbara, USA, 2008. Springer-Verlag.

[9] J.-C. Faugère, M. Safey El Din, and P.-J. Spaenlehauer. Computing Loci of Rank Defects of Linear Matrices using Gröbner Bases and Applications to Cryptology. In S. Watt, editor, *ISSAC '10: Proceedings of the 2010 international symposium on Symbolic and algebraic computation*, New York, NY, USA, 2010. ACM.

[10] K. Katsura. Theory of spin glass by the method of the distribution function of an effective field. *Progress of Theoretical Physics*, 87:139–154, 1986. Supplement.

[11] S. Lachartre. *Algèbre linéaire dans la résolution de systèmes polynomiaux Applications en cryptologie*. PhD thesis, Université Paris 6, 2008.

[12] W. Stein et al. *Sage Mathematics Software (Version 3.3)*. The Sage Group, 2009. `http://www.sagemath.org`.

[13] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

A Quantitative Study of Reductions in Algebraic Libraries

Yue Li
Texas A&M University
College Station, TX, USA
yli@cse.tamu.edu

Gabriel Dos Reis
Texas A&M University
College Station, TX, USA
gdr@cse.tamu.edu

ABSTRACT

How much of existing computer algebra libraries is amenable to automatic parallelization? This is a difficult topic, yet of practical importance in the era of commodity multicore machines. This paper reports on a quantitative study of reductions in the AXIOM-family computer algebra systems. The experiment builds on the introduction of *assumptions* in OpenAxiom. It identifies a variety of reductions that are candidate for implicit concurrent execution. An assumption is an axiomatic statement of an algebraic property. We hope that this study will encourage wider adoption of axioms, not just for the purpose of expression simplification and provably correct libraries, but also to enable derivation of implicit concurrency in a scalable fashion.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*Special-purpose algebraic systems*

General Terms

Algorithms, Design, Languages

Keywords

Parallel reduction, assumptions, computer algebra, OpenAxiom

1. INTRODUCTION

Reduction is a standard operation in computational algebra. For instance, the content of an univariate integer polynomial $P \in \mathbb{Z}[x]$ of the form $P(x) = \sum_0^n a_n x^n$ is the greatest common divisor of all its coefficients [2]:

$$\text{cont}(P) = \text{greatest-common-divisor}(a_0, a_1, ..., a_n).$$

Equivalently, it is the reduction of the monoid operator gcd over the coefficients of P. Furthermore, the computation can be arranged in a divide-and-conquer fashion, say

$$\text{cont}(P) = \text{combine}(\text{gcd}(a_0, a_1), \dots, \text{gcd}(a_{n-1}, a_n)),$$

where operator "combine" further computes the greatest common divisor of the results produced by each pair. The above expression may be evaluated concurrently by distributing pairs of polynomial coefficients to different computation engines, and then combine the results, which is known as parallel reduction pattern.

Detecting reduction patterns in large and typed programs presents interesting challenges. A programmer can discover implicit concurrency by observation, and further rewrite them with parallelism. However, such manual investigation is not scalable. On the other hand, automating detection of reduction pattern is also not trivial. A static analysis tool needs semantic information to compute facts underlying reductions. The process is much harder for reductions of user-defined operators. The principal reason is that most users write algorithms in impoverished programming languages, thereby leaving out essential semantic information.

This paper suggests a solution via an extension to an existing computer algebra programming language. We take OpenAxiom [8] as our experimental platform. We extend it with a new construct called *assumption*. An *assumption* is an axiomatic statement of mathematical properties. For instance:

```
forall(T: EuclideanDomain)
  assume MonoidOperator(T, gcd) with
    neutralValue = 0$T
```

That *assumption* states that gcd is a monoid operator over Euclidean domains, admitting 0 as neutral value. `MonoidOperator` is the name of the Spad category defining that property. We note that in the current system, we do not attempt to prove assumptions. As the keyword `assume` indicates, we just take the assumptions on faith, as if they were axioms.
The contributions of this paper include:

1. A quantitative study of reductions in the OpenAxiom algebras. We analyze the complete set of OpenAxiom algebra library, and show rich parallelization opportunities brought by monoid operator reductions.

2. A static analysis tool for reduction detection. The detector utilizes the algebraic properties of operators specified via user assumptions to identify implicit concurrency in user code.

3. Extension of the Spad programming language with *assumptions*.

4. A library of category hierarchies for algebraic properties in OpenAxiom.

The rest of this paper is organized as follows: section 2 gives a short introduction to Spad programming language; section 3 discusses the design principle of a hierarchy of algebraic operator categories; section 4 studies the extension of Spad programming language with *assumption*; our reduction detector is discussed in section 5; and the implementation details of the reduction detector are discussed in section 6; we present and discuss experimental results and running examples in section 7; Related work is in section 8; We conclude and discuss future directions in section 9.

2. EXTENDING AXIOM LIBRARIES

Spad is the library extension language of the AXIOM-family systems. It is a strongly typed language, with a two-level type system—categories and domains. We briefly discuss the key ideas of the Spad language relevant to this experiment. An in-depth coverage of the essentials of Spad is available from the AXIOM book [4]. Abstract algebraic structures are defined using *categories*. Categories can be thought of as specifications of algebraic concepts and properties. For instance, the monoid algebraic structure can be expressed in Spad as:

```
Monoid(): Category == Type with
  *: (%, %) -> %
  1: %
```

The category Monoid above specifies that the monoid algebraic structure has a binary operator * whose neutral element is 1. Specifications declared by categories are implemented by *domains*. The following fragment shows a version of domain ListMonoid, which implements the specification of the category Monoid:

```
ListMonoid(T: Type): Monoid == add
  import List(T)
  Rep == List(T)
  (x:%) * (y:%) == per concat(rep x, rep y)
  1: % == per empty()
```

ListMonoid domain is internally represented by domain List(T). It asserts membership to the category Monoid, and implements the binary operator and the neutral element with list concatenation operator concat and an empty list empty(), respectively. In the next section, we show how categories can be used for specifying algebraic properties of operators.

3. PROPERTY CATEGORIES

Most algebraic operators enjoy non-trivial properties. And most interesting algebraic algorithms are the result of skillful exploitations of operator properties, and structures of data they manipulate. Our approach to implicit concurrency rests upon the idea that data structures from computational algebra usually have rich algebraic properties, and algorithms operating on those structures should somehow be "tainted" by those properties. In particular, we are interested in those properties that enable automatic exploitation of implicit concurrency. To that end, we need mechanisms to express algebraic properties directly in programs. Furthermore, we need a way to organize those properties as library components so that they can be reused in large scale development—after all, mathematics are about organization of facts.

An effective operational way to think about AXIOM categories is to consider them as specifications. However, in their existing form they are closer a sub-language for initial algebra specification of data types without laws or equations. Support for axioms in categories has been considered by several researchers over the year, but there is no concrete implementation to date. We extent that notion of data type specification to cover specification of operator properties. Let us consider a simple hierarchy of algebraic operator categories, sufficient for our discussion in this paper. The hierarchy starts with the notion of binary operation:

```
MagmaOperator(T: SetCategory, op: (T,T) -> T): Category
  == Type
```

This definition says a binary operation on a domain satisfies the *magma operator* property. That is just a statement of the obvious. Now, we move on a more computationally interesting property: *associative operator*

```
AssociativeOperator(T: SetCategory,op: (T,T)->T): Category
  == MagmaOperator(T,op) with
    associativity:
      rule forall(a:T, b: T, c: T)
             op(a,op(b,c)) == op(op(a,b),c)
```

This category definition is essentially a logical statement (as Spad expression) of the property that an operator is associative if it is a magma operator and follows of a certain rule of re-association.

Next we consider monoid structures. An operator is a *monoid operator* if, in addition to being associative, it admits a neutral element:

```
MonoidOperator(T: SetCategory, op: (T,T) -> T): Category
  == AssociativeOperator(T, op) with
    neutralValue: T
```

Note that the neutral element is not a parameter to MonoidOperator. Rather, it is specified as a constant depending on the parameter op. This reflects the mathematical fact that the neutral value of a monoid operator is unique, therefore completely determined by that operator.

4. ASSUMPTIONS

Users express operator properties as *assumptions*. Properties are stated as facts without any attempt at proof beyond conventional type checking. We distinguish two kinds of assumptions: *ground assumptions* and *parameterized assumptions*.

4.1 Ground assumptions

A ground assumption states properties for a specific operator, whose input and output types are concrete non-parametrized domains. For example, the assumption

```
assume AssociativeOperator(NonNegativeInteger, max)
```

says the function max is a monoid operator over the domain of non-negative integers, with 0 as neutral value.

4.2 Parameterized assumptions

A parameterized assumption states properties for a family of operators. For example,

```
forall(T: Type)
  assume MonoidOperator(List(T), concat) where
    neutralValue == empty()$List(T)
```

asserts that concat is a monoid operation over List T for any type T. Type variables may be constrained, reflecting constraints on domains or operations. For instance, the parameterized assumption

```
forall(T: GcdDomain)
  assume MonoidOperator(T, gcd) where
    neutralValue == 0$T
```

states that gcd is a monoid operation over all GCD domains. whose neutral value is 0 of type T.

As illustrated in both examples, type variables in parameterized assumptions are introduced by declarations announced by the keyword forall. They must occur in deducible positions in assumptions.

In the experiments reported in this paper, an assumption is viewed like an annotation. It conveys a user's knowledge about an operator.

5. REDUCTION DETECTOR

The reduction detector takes a library file and assumptions from user, and attempts to extract reductions from the library file. The detection proceeds in two steps. First, it performs pattern matching on the fully typed abstract syntax trees obtained from the input program, after type checking and semantics elaboration. This phase yields a set of candidate reduction forms and scans forms. The detector then proceeds with only those binary operators for which associativity could be "certified" — either because they are built-ins or because they have matching assumptions.

5.1 Reduction forms

There are three ways to write reductions in AXIOM:

- explicit accumulation loop
- reduction operator form
- call to library function reduce

Explicit accumulation loop is probably the most widely known form. Consider the task of multiplying all integer values in a sequence seq. Writing that in AXIOM is just as simple as writing it in most programming languages:

```
result := 1
for v in seq repeat
  result := result * v
```

Here, the variable result is initialized with the neutral value of multiplication; then it is updated at every execution of the body of the loop. The final result is the accumulated value.

In the tradition of APL, AXIOM offers a short notation for reduction. The previous loop can be written as */seq. Here, / is the built-in reduction operator, not the ratio operator. It should be noted that in all flavours of AXIOM systems (including current releases of OpenAxiom), the built-in reduction operator is applicable only to a handful known monoid operators.

Finally, one can just call a library function: reduce(*,seq,1). Note that this form explicitly specifies the neutral value, just like in the accumulation loop case. That value was left implicit in the form using built-in reduction operator. Ideally, one should not have to supply the neutral value, since a monoid operation uniquely defines its neutral value.

5.2 Pattern matching reduction forms

Several semantic based pattern matching strategies are designed for extracting the various reduction forms.

Accumulation loop.

We introduce the concept *basic loop* for the purpose of describing our algorithms for detecting accumulation loops. A basic loop is a loop controlled by for-iterators, such that each statement in its body is either a variable definition, or an assignment to a previously defined variable. Example:

```
result : Integer := 0
for i in 1..10 repeat
  x : Integer := i+1         -- variable definition
  result := x * 2            -- variable assignment
```

We define an *accumulation loop* as basic atomic loop, where definitions or assignments involve certain expressions in *recognizable form*:

$$\iota_1 \cdots \iota_n \texttt{ repeat } \vec{\beta}$$

where $\iota_1, \ldots, \iota_n$ are for-iterators of the forms

- for v in e, with v a variable, and e a sequence
- for v in e_1..e_2, with v a variable, and e_1 and e_2 integer-valued expressions denoting the bounds of the loop-control variable v

An expression β is either a basic assignment of the form $v := f(\chi, e)$ or $v := f(e, \chi)$, or a conditional controlled by a side-effect free predicate and whose branches are sequences of basic assignments. The operand χ is either the same variable v being assigned to, or another expression of the form $f(\chi, e)$ or $f(e, \chi)$. We don't allow the control loop variables to be modified in the body of the loop. The accumulating variable v should have a linear occurrence in the right hand side of the assignment.

For simplicity, we take side-effect free predicate to mean a call to Boolean expression that does not use effectful functions. This is a semantics notion, therefore hard to check in practice. However, there is a notational convention used in AXIOM libraries, where a function name ending with symbol "!" indicates possibly effectful functions. Examples include concatenation function concat!, duplicate removal function removeDuplicate!, etc. The current implementation of the reduction detector does not allow effectful functions in accumulation—not just in the predicates.

The notion of recognizable accumulation loop is adapted from the aggregate array computation form studied by Liu and Stoller [6]. This adaptation is semantic-based since it draws heavily from type information. In each accumulation assignment, the same operator f has to be used consistently to accumulate values into the accumulation variable v. Because of operator overloading, we need to make sure that the same operator is applied consistently, and that requires overload resolution.

Built-in reduction operator.

The detection of built-in reduction operator is purely syntactic, unfortunately. For instance, parsing of +/[1,2,3,4] gives:

```
(REDUCE + 0
   (COLLECT (IN G784 (construct (One) 2 3 4)) G784))
```

The value 0 is automatically generated by compiler, and is inserted into the AST as the built-in neutral element of +. The reduction detector typechecks each parameter of the REDUCE operator except the neutral value which is automatically generated, and verifies that the operator parameter is a binary operator over some domain d, and the other parameter of the reduce form is a list whose element has type d.

Library function call.

The function reduce is heavily overloaded in AXIOM algebra libraries for implementing different functionalities or semantics. Therefore, given an AST whose operator is reduce, the reduction detector needs to type check all arguments, given enough seed to proceed with overload resolution.

5.3 Parallel prefix forms

A *parallel prefix* form (or scan) is a generalization of reduction. A parallel prefix form takes a sequence and a binary operator, and returns a sequence. Each entry of the result is filled by a reduction, *i.e.*, the value of the i-th element is given by the reduction which applies the input binary operator to combine the values up to the i-th element of the input sequence. That can be expressed in at least two forms:

- explicit scan loop
- call to library function `scan`

We elaborate only on loops. The following example illustrates a prefix sum of the sequence `seq`:

```
sum := first seq
for i in 2.. #seq repeat
  sum := sum + seq.i
  res.i := sum
```

The implementation of OpenAxiom library functions `scan` are based on scan loops as well as self recursions.

Matching parallel prefixes.

Strategies for extracting scan operations are again based on pattern matching. We use two kinds of recognizable form.

$$\iota_1 \cdots \iota_n \; \texttt{repeat} \; \vec{\xi}$$

where $\iota_1, \ldots, \iota_n$ are `for`-iterators. The form of the loop body $\vec{\xi}$ is partially determined by the specific form of `for`-iterators:

- If the iterators are of the form `for` v `in` e, with v a variable, and e a sequence, then $\vec{\xi}$ is a two statement sequence, where the first statement is an accumulation of the form $v := f(v, e)$ or $v := f(e, v)$ where v is a variable, and the second statement is for updating the vector for storing the prefix result, this can be a sequence concatenation expression such as $r := \text{concat}(r, v)$, where r is the resulting sequence of scan, and operator concat appends the value of v to the end of the resulting sequence v.
- When the iterators are of the form `for` i `in` $e_1 .. e_2$, with i a variable, and e_1 and e_2 integer-valued expressions denoting the bounds of the loop-control variable i. The loop can be either one statement of the form $r.i := f(r.(i-1), s.i)$ where the $(i-1)$-th element of the resulting sequence r and the i-th element of the input sequence s are combined together via binary operator f, the result is written into i-th element of r; Or $\vec{\xi}$ can be a sequence containing two assignments, where the first statement is of the form $v := f(v, s.i)$ or $v := f(s.i, v)$ where v is a variable, and the second statement updates value of v into the resulting vector r, *e.g.*, $r.i := v$, or $r := \text{concat}(r, v)$.

5.4 Assumption uses

Assumptions, as reported in this paper, are not used directly by users. Rather, they form an external knowledge database consulted by static analysis tools such as the reduction detector. For implicit concurrency, the most important property we focus on is associativity. To get there, the reduction detector needs to gather properties from the assumption database supplied by the user. Then, the computed information is passed to the associativity checking engine.

If the operator in a reduction form is associative, the form is deemed *fully parallelizable*. On the other hand, if associativity cannot be decided from the set of available assumptions, the accumulation loop is said to be *partially parallelizable*. For instance, the following loop

```
for i in 1..10 repeat
  x := x + i
  y := y * i
  z := z quo i
```

is only partially parallelizable. However, it can be rewritten as two loops: one that is fully parallelizable:

```
for i in 1..10 repeat
  x := x + i
```

and a second that is not readily so (according to our recognizable form definition):

```
for i in 1..10 repeat
  y := y * i
  z := z quo i
```

6. IMPLEMENTATION

The reduction detector is implemented as a Spad library of a branch of the OpenAxiom system [1]. The implementation of the reduction detector accounts for approximately 2000 lines of Spad code. It uses the AST library component of the standard OpenAxiom system. The overall workflow is illustrated in Figure 1. To maintain portability to other AXIOM systems not implementing assumptions, users are required to write their Spad code and assumptions in different files.

6.1 A Spad typechecker in Spad

The pattern matching and assumption verification steps of reduction detection requires typechecking. Instead of exporting the typechecking function from the Spad compiler which is written in a lower-level language named Boot, we built an independent typechecking library in Spad. The implementation of the typechecker consists of about 1000 lines of Spad code. It is used for typechecking, overload resolution, and assumption checking. The library is tested with the complete set of OpenAxiom algebra library, where some programs were edited before passing to the typechecker. It is still experimental.

6.2 Preprocessing before pattern matching

A basic atomic loop needs to be preprocessed before pattern matching. The preprocessing step essentially consists of forward substitution of expression [7], eliminating assignments to the variables used as intermediate stores.

The purpose of this pass is to expose “hidden” accumulations. Indeed, some intermediate stores may introduce unnecessary dependencies, which may prevent an accumulation loop from being identified.

6.3 Inferring properties from assumptions

Assumptions are organized in hierarchies. So, it can happen that a property the reduction detector is looking for is not textually present in the assumption database, but must be derived using the semantics of hierarchy as entailment. Consider

```
forall(D: IntegralDomain)
  assume MonoidOperator(D, *) where
    neutralValue = 1$D
```

The assumption states that the operator * specified by `IntegralDomain` category is a monoid operator. This implies that all other properties entailed by the parents categories of `MonoidOperator` (e.g. associativity) are also inherited.

[1]The source code is at `http://open-axiom.svn.sf.net/svnroot/open-axiom/yli-sandbox/`

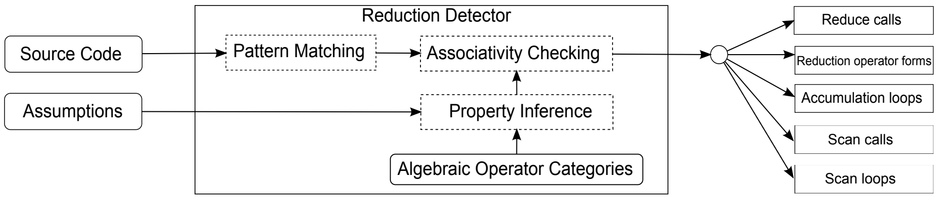


Figure 1: The workflow of our reduction detector.

7. EXPERIMENTS

The experiment has two parts. The first only uses the pattern matching component to identify all reductions used in OpenAxiom algebra libraries. The second identifies those reductions that are parallelizable.

7.1 OpenAxiom algebra library

The OpenAxiom library consists of type definitions. Each type is a package, a domain, or a category.

Item	Number
Type definitions	1129
Category definitions	222
Domain definitions	434
Package definitions	473
Function definitions	12748

Table 1: Statistical properties of OpenAxiom algebra library.

From Table. 1, we observe that categories account for the smallest portion of the library. This is expected since category are specifications, and it is reasonable to have fewer specifications than implementations. Packages are almost as prevalent as domains. On average, there are 11 function definitions per type definitions. Note that a category definition can also contain function definitions in its body, *e.g.*, the generic operator `gcd` of category `GcdDomain`.

7.2 Reduction extraction

The results of running the pattern matching engine over the algebra library files is summarized in Table. 2 and Table. 3. A significant portion (about 32%) of loops are atomic loops, and about 46% of atomic loops implement reductions. The number of reductions per 100 function definitions is smaller than our expectation, this is mainly because the current implementation does not compute the transitive closure of call chains to functions implementing reductions. Among the three kinds of reduction forms, built-in reduction operator is the most frequently used.

Table. 4 gives the distribution of accumulation loops according to the number of accumulation statements in their bodies. The data, with the limitations of the current implementation, suggest that most accumulation loops contain a single accumulation statement; those containing more than 2 accumulations are rare.

7.3 Distribution of reduction forms

A set of 13 operators were manually annotated as associative: + of category `AbelianMonoid`, * of category `Monoid`, `gcd` and `lcm` of category `GcdDomain`, `max` and `min` of category `OrderedSet`, two list concatenation operators `concat` and `append`, one list union operator `setUnion` of domain `List(T)` where `T` has category `Type`, two matrix concatenation operators `horizConcat` and `vertConcat` over domain `Matrix(T)` where `T` has `Type`, and two local functions

Item	Number
Loops	2181
Atomic loops	689
Reductions	820
Reductions/100 function defs	6

Table 2: Statistical properties of loops and reductions.

Item	Number	Percentage in reductions
Accumulation loops	333	41%
Function `reduce` calls	73	9%
Built-in reduce operator	414	50%

Table 3: Statistical properties of reduction forms.

No. of Acc. Stmt.	No. of Acc. Lps.	Percent. in Acc. Lps.
1	305	91.6%
2	22	6.6%
3	6	1.8%

Table 4: Distribution of the number of accumulation loops with different number of accumulation statements in their bodies.

`pairsum` of domain `List(T)` where `T` has category `Type`, and the operator `sum` over domain `Expression(DoubleFloat)`.

All algebraic properties in this experiment entail associativity. Table. 5 lists the distribution of the different parallel reductions regarding each reduction operator. Partially parallelizable loops (PPL) seem to be less frequent than other reduction loops. This indicates that these 13 operators rarely appear in an accumulation loop with more than one accumulation statements involving a different accumulation operator. Addition, multiplication, and list concatenation are dominant in parallelizable reduction forms.

7.4 Parallel prefix extraction

The statistics obtained for parallel prefix suggests that it is barely used in current versions of OpenAxiom library. The detector discovers 4 scan loops and 11 `scan` function calls. We found only two scan function calls which are parallelizable due to the use of associative operator + over `PolynomialCategory` and `Ring`, respectively.

7.5 Rejected cases

The analysis is conservative; that is it may reject some reduction forms on the ground that it cannot certify — based on available static information — that they are indeed bona fide parallel (prefix) reduction forms. For instance, the recognizable form requires that each accumulation assignment uses only one binary operator. In particular, it forbids cases where different accumulation operators appear in one accumulation. For example, the following accumulation loop from function `basisOfRightNucleus` of package

Operator	PPL	FPL	PRC	PRF
+ of AbelianMonoid	7	76	8	148
* of Monoid	6	21	1	76
gcd	1	0	4	4
lcm	0	0	3	7
concat	1	18	1	1
append	1	8	1	17
max	2	1	8	36
min	2	0	3	12
horizConcat	0	2	2	0
vertConcat	0	3	0	0
setUnion	0	2	1	35
pairsum	0	1	0	0
sum	0	0	3	0

Table 5: Amount of parallel reductions based on different user assumptions. PPL: Partially Parallelizable Loop, FPL: Fully Parallelizable Loop, PRC: Parallelizable `reduce` call, PRF: Parallelizable Reduce Form.

AlgebraPackage:

```
for l in 1..n repeat
  entry :=  entry + elt(gamma.l,k,i)*elt(gamma.s,j,l)_
                - elt(gamma.l,j,k)*elt(gamma.s,l,i)
```

was rejected. Indeed, parsing of the loop above gives the AST:

```
(REPEAT  (STEP l (One) 1 n)
 (%LET  entry
  (- (+ entry (* (elt (gamma l) k i) (elt (gamma s) j l)))
   (* (elt (gamma l) j k) (elt (gamma s) l i)))))
```

The appearance of the operator + does not match our definition of recognizable form. An algebraic term rewriting may help the detector. For instance, by applying the rewriting rule $-x \equiv +(-x)$ to the example above, we obtain:

```
for l in 1..n repeat
  entry := entry + elt(gamma.l,k,i) * elt(gamma.s,j,l)_
               + (-elt(gamma.l,j,k) * elt(gamma.s,l,i))
```

which becomes:

```
(REPEAT  (STEP l (One) 1 n)
 (%LET  entry
   (+ (+ entry (* (elt (gamma l) k i) (elt (gamma s) j l)))
    (- (* (elt (gamma l) j k) (elt (gamma s) l i))))))
```

The detector reported 14 cases of this kind. Other examples include 3 uses of side-effecting functions, and 2 of the unsupported recursive parallel prefix patterns.

8. RELATED WORK

8.1 Reduction detection

Reduction detection is a well developed program analysis technique in the compiler construction community. It is widely used in automatic parallelization of loops. An internal representation of loops needs to be specified, the definition of reduction patterns and the design of pattern matching algorithms are further based on that internal representation. Jouvelot and Dehbonei [5] represent loops symbolically, and reductions patterns are formalized as values of symbolic stores so that pattern matching is applied to the symbolic stores. Pinter and Pinter [9] use dependence graphs. Redon and Feautrier [10] use a preprocessor that taking loops and generates linear recurrence equations as internal representations, and reductions are discovered via reasoning on the generated linear equations. Liu and Stoller [6] describe how incrementalization aids optimizing aggregate array computation, which is a very typical generalization of reductions. In that paper, a recognizable form for reduction semantics is derived directly from the abstract syntax tree of an accumulation loop. The recognizable accumulation loop discussed in this paper builds on that work. This form is syntax directed, which simplifies implementation. Indeed, instead of building a tools for between several representations, the reduction detector presented in this paper shares the fully typed AST with the type checker. Recently, Gautam and Rajopadhye [3] studied polyhedra equations to represent reductions. The polyhedra model provides powerful mathematical foundations to simplify the algorithmic complexities of accumulation loops. Empirical data, obtained by augmenting our reduction detector extractor, show a total of 431 loop nests, out of which 57% are affine control loops, and an average of 2.4 per nest depth.

Much of this work [5, 9, 10, 6] supports reduction extraction for Fortran program, Liu and Stoller's work [3] implements reduction detection and simplification using the ALPHA language and the MMALPHA framework for transforming ALPHA program [1]. However, for programming languages such as Spad, Aldor, C++, and Java, those tools cannot correctly extract reductions of user defined functions—compiler does not have knowledge about the algebraic properties of some reduction operator over user defined types due to overloading. The assumption mechanism presented in this paper helps user convey operator properties to compiler tools.

8.2 Attributes

The original AXIOM system supports algebraic properties via *axioms* and *attributes* in type descriptions [4, Chapter 12]. However, axioms are just stated as comments, they do not affect the compiler in any way. *Attributes* are uninterpreted identifiers or tags. However, the attribute support is very limited and it is not possible to express that a function is a monoid structure with a specific neutral value.

8.3 Maple `assume` facility

The Maple computer algebra system has enjoyed an assume facility since the work of Weibel and Gonnet [13, 14]. It enables a powerful conditional rewriting tool for expressions simplification. That functionality is not directly supported by the work described in this paper. However, it would be interesting to explore how Maple's `assume` facility—which is mostly dynamic—could be combined with the type-directed approach of this paper to support correct and fast algebraic computations on modern computers.

9. CONCLUSIONS AND FUTURE WORK

This paper presented an empirical and quantitative study of reductions in the AXIOM algebra libraries. The experiments suggest rich parallelization opportunities exposed by the uses of a language extension called *assumptions*. Experimental data show that specifying operator properties directly in code, checkable by the compiler, is beneficial for parallelizable reductions. The core idea of this approach is not restricted to Spad or Aldor programs. It can be applied to programs written in higher-level languages such as C++ or Java.

There are several directions for future work that we would like to explore. Currently, a parallel library is being built in OpenAxiom aiming at providing support for higher-level parallel programming. It would be interesting to see how a compiler that utilizes the information provided by our reduction detector, could perform an effective implicit parallization. Another direction would be to develop more use cases of *assumption* to increase its benefit to symbolic computation. As inspired by recent work on axiom based verifications for Java program [12, 11], it would be also interesting to see how user assumption can help verify program transformations for computer algebra code.

10. REFERENCES

[1] Alpha. `http://www.irisa.fr/cosi/ALPHA`, IRISA, France, 2010.

[2] J. V. Z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2003.

[3] Gautam and S. Rajopadhye. Simplifying reductions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 30–41, New York, NY, USA, 2006. ACM.

[4] R. D. Jenks and R. S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, 1992.

[5] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 186–194, New York, NY, USA, 1989. ACM.

[6] Y. A. Liu, S. D. Stoller, N. Li, and T. Rothamel. Optimizing aggregate array computations in loops. *ACM Trans. Program. Lang. Syst.*, 27(1):91–125, 2005.

[7] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[8] OpenAxiom. `open-axiom.org`, 2010.

[9] S. S. Pinter and R. Y. Pinter. Program optimization and parallelization using idioms. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–92, New York, NY, USA, 1991. ACM.

[10] X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 132–145, London, UK, 1993. Springer-Verlag.

[11] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 389–402, New York, NY, USA, 2010. ACM.

[12] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, New York, NY, USA, 2009. ACM.

[13] T. Weibel and G. H. Gonnet. An algebra of properties. In *ISSAC '91: Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, pages 352–359, New York, NY, USA, 1991. ACM.

[14] T. Weibel and G. H. Gonnet. An assume facility for cas, with a sample implementation for maple. In *DISCO '92: Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 95–103, London, UK, 1993. Springer-Verlag.

Parallel Sparse Polynomial Division Using Heaps

Michael Monagan [*]
Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.
mmonagan@cecm.sfu.ca

Roman Pearce
Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.
rpearcea@cecm.sfu.ca

ABSTRACT

We present a parallel algorithm for exact division of sparse distributed polynomials on a multicore processor. This is a problem with significant data dependencies, so our solution requires fine-grained parallelism. Our algorithm manages to avoid waiting for each term of the quotient to be computed, and it achieves superlinear speedup over the fastest known sequential method. We present benchmarks comparing the performance of our C implementation of sparse polynomial division to the routines of other computer algebra systems.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algebraic Algorithms

General Terms: Algorithms, Design, Performance

Keywords: Parallel, Sparse, Polynomial, Division, Heaps

1. INTRODUCTION

Modern multicore processors let you write extremely fast parallel programs. The cores share a coherent cache with a latency of nanoseconds, where communication can occur at roughly the speed of the processor. The challenge now is to design fast parallel algorithms that execute largely in cache and write only their result to main memory.

In [11] we presented such a method for sparse polynomial multiplication. Given polynomials f and g with $\#f$ and $\#g$ terms, we construct $f \times g = \sum_{i=1}^{\#f} \sum_{j=1}^{\#g} f_i \cdot g_j$ by creating, sorting, and merging all the products in parallel, entirely in the cache. We based the algorithm on Johnson's method [7] which we found to be a fast sequential approach in [12, 13].

Johnson's algorithm computes $\sum_{i=1}^{\#f} f_i \cdot g$ using a binary heap to perform an $\#f$-ary merge. The products $f_i \cdot g_j$ are constructed on the fly so only $O(\#f)$ scratch space is used. It begins with $f_1 \cdot g_1$ in the heap, and after merging $f_i \cdot g_j$ it inserts $f_i \cdot g_{j+1}$. When $j = 1$ it also inserts $f_{i+1} \cdot g_1$. This assumes that f and g are sorted in a monomial ordering.

[*]We gratefully acknowledge the support of the MITACS NCE of Canada and NSERC of Canada

In our parallel algorithm each core multiplies a subset of the terms of f by all of g. Those subproblems were chosen because Johnson's algorithm is $O(\#f\#g \log \#f)$. The cores write their intermediate results to circular buffers in shared cache, and a global instance of Johnson's algorithm merges the buffers to produce the result. Superlinear speedup was obtained from the extra local cache in each core.

This paper obtains a similar result for sparse polynomial division. This is a considerably harder problem, because in multiplication the polynomials f and g are known up front. For division, we are given the dividend f and the divisor g, and we construct each new term of the quotient q from the largest term of $f - q \cdot g$. This produces a tight dependency among the terms of the quotient, and adds synchronization and contention to the multiplication of q and g.

We are not aware of a comparable attempt to parallelize sparse polynomial division. Our algorithm is asynchronous and does not wait between the computation of q_i and q_{i+1}. In [15], Wang suggests parallelizing the subtraction of $q_i \cdot g$ and synchronizing after each new term of the quotient. No data is provided to assess the effectiveness of this approach but we believe the waiting would be a problem. It appears the CABAL group [10, 14] has also tried this approach. For dense polynomials, Bini and Pan develop a parallel division algorithm based on the FFT in [1], and in [8], Li and Maza assess parallelization strategies for dense univariate division modulo a triangular set.

Our paper is organized as follows. In Section 2 we discuss the division algorithm and the challenges of parallelization. We describe our solutions and present the algorithm. Then in Section 3 we present benchmarks of our implementation. We compare its performance and speedup to the sequential routine of [13], the parallel multiplication codes of [11], and the division routines of other computer algebra systems.

2. SPARSE POLYNOMIAL DIVISION

Consider the problem of dividing two sparse multivariate polynomials $f \div g = q$ in $\mathbb{Z}[x_1, \ldots, x_n]$. In general there are two ways to proceed. In the *recursive* approach we consider them as polynomials in x_1 with coefficients in $\mathbb{Z}[x_2, \ldots, x_n]$. We divide recursively to obtain a quotient term q_i, then we subtract $f := f - q_i g$. The recursive coefficient operations could be performed in parallel as suggested by Wang in [15].

One problem with this method is the many intermediate pieces of storage required. Memory management is difficult to do in parallel while preserving locality and performance. For exact division the polynomial f is also reduced to zero, so the construction of $q \cdot g$ in memory is wasteful.

In the *distributed* approach we impose a monomial order on $\mathbb{Z}[x_1,\dots,x_n]$ to divide and cancel like terms. We divide the largest term of f by the largest term of g to construct the first term q_1 of the quotient, and repeat the process for $f - q_1 g$ to obtain q_2, and so on, until either $f - \sum q_i g = 0$ or the division fails. There may be very little work between the computation of q_i and q_{i+1}, which makes this approach difficult to parallelize.

But it has a critical advantage for division. Using a heap we can merge the terms of $q \cdot g$ in descending order without constructing large objects in memory. For example, when a new term q_i is computed we can insert $q_i \cdot g_2$ into the heap, and when this term is used we would replace it with $q_i \cdot g_3$. This is Johnson's "quotient heap" algorithm, where a heap of size $\#q$ is used to merge $\sum_{i=1}^{\#q} q_i \cdot (g - g_1)$. It uses $O(\#q)$ memory in total, far less than the $O(\#f + \#q\,\#g)$ memory used by the recursive approach.

One nice feature of the quotient heap algorithm is that a new term q_i completely determines a row $q_i \cdot g$ in the heap. If we could distribute the q_i to different processors it would be easy to parallelize division. However one problem is that a new term of the quotient could be computed at any time. We may also use $q_i \cdot g_2$ immediately to compute q_{i+1}. This suggests an alternative partition of the work.

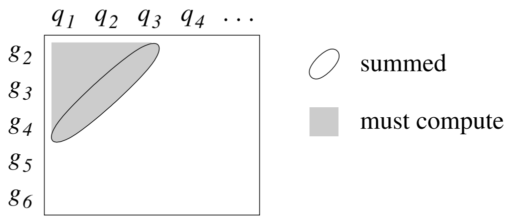
The "divisor heap" algorithm of Monagan and Pearce [12] computes $\sum_{i=2}^{\#g} g_i \cdot q$ instead. That is, elements of the heap walk down the quotient and multiply by some divisor term. Distributing terms of g to the threads solves two problems. First, we can divide the work in advance with good locality and suggest the number of threads. Second, $\{g_2, g_3, \dots, g_k\}$ may be merged by the processor computing quotient terms so that their products are known without delay. Our entire algorithm is designed to avoid waiting in a typical division, and this is one of two situations we address.

One may ask whether there is a loss of efficiency because the divisor heap algorithm performs $O(\#f + \#q\#g \log \#g)$ monomial comparisons. This is not optimal when $\#q < \#g$. In [13] we present a sequential division algorithm that does $O(\#f + \#q\#g \log \min(\#q, \#g))$ comparisons. However our divisor heap is run on subproblems with $\#g/p$ by $\#q$ terms where p is the number of threads, so the threshold becomes easier to meet as the number of threads increases.

2.1 Dependencies

We begin with an example that shows the main problem encountered in parallelizing sparse polynomial division. Let $g = x^5 + x^4 + x^3 + x^2 + x + 1$ and $f = g^2$. To divide f by g we will compute $q_1 = f_1/g_1 = x^5$, $q_2 = (f_2 - q_1 g_2)/g_1 = x^4$, $q_3 = (f_3 - q_1 g_3 - q_2 g_2)/g_1 = x^3$, and so on. Each new term of the quotient is used immediately to compute subsequent terms, so q_k depends on the triangle of products $g_i \cdot q_j$ with $i + j - 1 \le k$, as shown below for q_4.

Figure 1: Dependency in Dense Univariate Division.



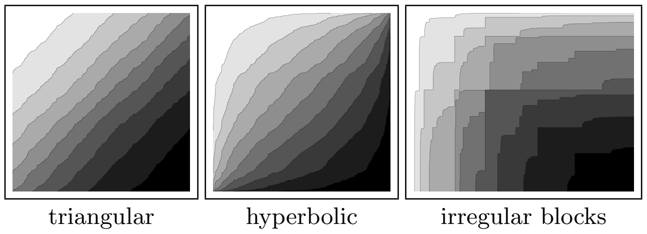
In parallel division the products $g_i \cdot q_j$ are merged using multiple threads. Our problem is to divide up the products in a way that mostly prevents having threads wait for data. For example, in the computation above we compute q_k and then immediately use $q_k g_2$ to compute q_{k+1}. It would make sense to do both operations in the same thread. Otherwise, one thread will compute q_k and stop to wait for $q_k g_2$ while the thread computing $q_k g_2$ waits for q_k and then carries out its task. Waiting serializes the algorithm because the round trip latency is longer than it takes to compute terms.

In the dense example (see Figure 1) we might be able to multiply $g_6 \cdot q$ in a separate thread without waiting for any of its terms, because after we compute q_k we need to merge $\{g_2 q_k, g_3 q_k, g_4 q_k, g_5 q_k\}$ before $g_6 q_k$ is used. We could merge other terms as well but those four have distinct monomials. The second thread may still have to wait for q_k if it doesn't have enough other work to do.

The structure of sparse polynomial multiplication is that $g_i q_j > g_{i+1} q_j$ and $g_i q_j > g_i q_{j+1}$ when the terms of q and g are sorted in a monomial ordering. In general this is called $X + Y$ sorting, see Harper et al. [6]. We are exploiting this structure to get parallelism in the multiplication of q and g. The approach is a recognized parallel programming pattern called *geometric decomposition.* For details see [9].

Our algorithm partitions the products $\{g_i q_j\}$ into regions that are merged by different threads. The $X + Y$ structure provides a lower bound on the amount of work that is done before a term from an adjacent region is needed. The work is used to conceal the latency of communication so that our threads can run independently and do not have to wait.

Figure 2: Common X+Y Sort Orders.



Whatever partition we choose will have to interact nicely with the construction of the quotient q, but there is no way to know the dependencies of q in advance. So we identified three common cases by experiment, see Figure 2. To create each graphic, we sorted the products $\{g_i q_j\}$ for $1 \le i \le \#g$ and $1 \le j \le \#q$ and shaded them from white to black. The image shows the order that terms are merged, and the first row shows when we construct each term of the quotient.

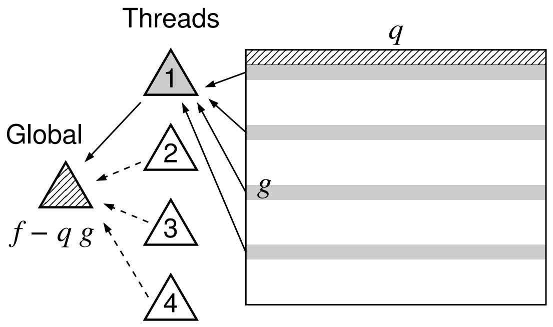
The triangular dependencies of dense univariate divisions (see Figure 1) are apparent in the first image, although the structure is found in sparse problems too. In this case $O(k)$ terms are merged between the computation of q_k and q_{k+1}. Merging and quotient computation both occur at the same regular rate, so this is the easiest case to parallelize. In the hyperbolic case the quotient is computed rapidly, with very little work between the computation of q_k and q_{k+1}. There we must avoid waiting for $\{g_2 q_k, g_3 q_k, \dots\}$ to be computed since those terms will be needed immediately. The last case is the hardest one to parallelize. Polynomials with algebraic substructure tend to produce blocks which must be merged in their entirety before any new quotient term is computed. In the next section we describe our solution.

2.2 Parallel Algorithm

Our parallel division algorithm borrows heavily from our multiplication algorithm in [11]. To each thread we assign a subset of the partial products $\{g_i \cdot q\}$. These are merged in a heap and the result is written to a buffer in shared cache. A global function is responsible for merging the contents of the buffers and computing new terms of the quotient. This function is protected by a lock.

Unlike in the parallel multiplication algorithm, the global function here is also assigned a strip of terms along the top $(g_1 + \cdots + g_s) \cdot q$. This allows it to compute some quotient terms and stay ahead of the threads. It uses g_1 to compute quotient terms and the terms $(g_2 + \cdots + g_s) \cdot q$ are merged. Then the strip $(g_{s+1} + \cdots + g_{2s}) \cdot q$ is assigned to thread 1, the next strip of s terms is assigned to thread 2, and so on, as in Figure 3 below. The strip height s is derived from the number of terms in g, refer to Section 2.3 for details.

Figure 3: Parallel Sparse Division Using Heaps.



The threads merge terms from left to right in the style of a divisor heap of Monagan and Pearce [13]. Each iteration of the main loop extracts all of the products $g_i \cdot q_j$ with the largest monomial, multiplies their coefficients to compute a sum of like terms, and inserts their successors $g_i \cdot q_{j+1}$ into the heap to set up the next iteration of the algorithm.

A major problem is that after $g_i \cdot q_j$ is extracted from the heap and merged, we may find that q_{j+1} does not yet exist. For example, towards the end of a division there will be no more quotient terms. The threads need some way to decide that it is safe to continue without $g_i \cdot q_{j+1}$ in the heap.

In the sequential division algorithm this is easy because $g_1 \cdot q_{j+1} > g_i \cdot q_{j+1}$ in the monomial order. This guarantees q_{j+1} is constructed (by dividing by g_1) before any products involving it need to be in the heap. We can safely drop the products missing q_{j+1} as long as they are reinserted before they could be merged. For example, in our algorithm in [13] we set bits to indicate which g_i have a product in the heap. When a new quotient term q_{j+1} is computed we check if g_2 has a product in the heap and insert $g_2 \cdot q_{j+1}$ if it does not, and when we insert $g_i \cdot q_j$ with $i < \#g$, we also insert the next product for g_{i+1} if it is not already in the heap.

In the parallel algorithm the computation of the quotient is decoupled from the merging of products, so this strategy does not work. It becomes difficult to maintain consistency in the algorithm and expensive synchronization is required. Eventually we made a compromise – if a thread encounters $g_i \cdot q_{j+1}$ and q_{j+1} is missing, the thread must wait for q_{j+1} to be computed or be relieved of the task of merging $g_i \cdot q$. The idea is to have the global function steal rows from the threads to allow them to proceed.

Figure 4: The Global Function Steals Rows.

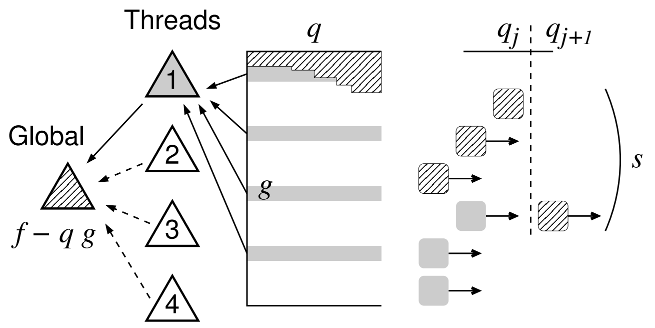


Figure 4 shows the global function in more detail. At the beginning of the computation it is assigned a strip of $s = 4$ terms. It uses g_1 to construct quotient terms and it merges $(g_2 + g_3 + g_4) \cdot q$ using a heap. After merging $g_2 \cdot q_j$, it sees that q_{j+1} has not been computed. It steals $g_5 \cdot (q_{j+1} + \cdots)$ by incrementing a global bound that is read by all threads. This bound is initially set to 4, and it will be updated to 5. When new quotient terms are computed, the current value of the bound is stored beside them for the threads to read.

Two possibilities can now occur in the Figure 4 example. If the thread merging $g_5 \cdot q$ reaches $g_5 \cdot q_{j+1}$ before q_{j+1} has been computed, it checks the global bound and the number of terms in the quotient. With no more quotient terms and a global bound greater than or equal to 5, it drops the row from its heap. Otherwise, if q_{j+1} is computed first, a bound of at least 5 is stored beside q_{j+1}. The thread sees this and again drops the row from its heap.

Stealing rows in the global function allows the threads to continue merging terms without any extra synchronization. If used aggressively it also eliminates waiting, at the cost of serializing more of the computation. This is a bad tradeoff. We prefer to steal as few rows as possible with a reasonable assurance that waiting will not occur.

2.3 Implementation

It is a non-trivial matter to sit down and implement this algorithm given the main idea. With sequential algorithms one expects the performance of implementations to vary by a constant factor. This is not the case for complex parallel algorithms since implementation details may determine the scalability. These details are a critical aspect of the design.

Our main challenge in designing an implementation is to minimize contention. This occurs when one core reads data that is being modified by another. In the division algorithm the quotient is a point of contention because we compute it as the algorithm runs and it is used by all of the threads.

We manage contention by using one structure to describe the global state of the algorithm. Shared variables, such as the current length of the quotient and the bound are stored on one cache line and updated together. Each thread reads these values once and then continues working for as long as possible before reading them again. This optimization may reduce contention by up to an order of magnitude.

We first used the trick of caching shared variables in the circular buffers of the parallel multiplication algorithm [11]. Those buffers are reused here. They reach 4.4 GB/s on our Intel Core i7 920 with this optimization, but only 1.2 GB/s without it. This shows just how high the cost of contention is for only two threads, and with more threads it is worse.

We now present the algorithm. The first function sets up the global state and creates the threads. When the threads terminate, it could be because the algorithm has completed or because the global function has stolen every row. In the latter case we continue to call the global function until the division is complete.

Just like our multiplication algorithm [11] we run at most one thread per core to avoid context switches. For X cores we compute $t = \sqrt[3]{\#g}$, create $p = min(t/2, X)$ threads, and give each thread strips of size $s = t^2/p$ terms. This value is a compromise between large strips which are fast and small strips which uniformly distribute the work.

The next function is the local merge that we run on each thread. It creates a heap and tries to add the first product. If the necessary quotient term does not exist yet, it tries to enter the global function and compute more quotient terms. It also discards any products stolen by the global function.

A product $g_i \times q_j$ has been stolen if q_j exists $(j < t)$ and $i \leq bound(q_j)$, or if q_j does not exist $(j \geq t)$ and $i \leq b$. The function will block in the case $j \geq t$ and $i > b$, i.e. when q_j does not exist and the row has not yet been stolen.

An important detail of the algorithm is that it must use memory barriers to ensure correctness. For example, as the algorithm runs, the global function computes new quotient terms and steals rows by incrementing a bound. Imagine if both were to happen in quick succession. A thread may see the bound modified first and discard a row before it merges all of the terms. Memory barriers enforce the correct order.

We use a simple rule: *'first written, last read'* to logically eliminate race conditions from our program. With this rule threads can read a volatile global state and act consistently as long as the variables are monotonic. Here the number of rows stolen and quotient terms computed only increase.

The global function is shown on the next page. It inserts terms from the buffers to update the global heap G, but at the start of the division there is no quotient and $mergeG$ is set to $false$. It performs a three way comparison to decide which of the dividend, local heap, and global heap have the largest monomial that must be merged. We write this step in a clear but inefficient way. Our implementation performs at most two ternary comparisons that return $<$, $>$, or $=$.

The global function then merges the maximal terms. The local heap case contains additional logic to add stolen rows. After merging $g_i \times q_j$, we check to see if g_{i+1} has a term in the heap. If not and $i+1 \leq bound(q_j)$ we insert the row for g_{i+1} starting at q_j. Otherwise $g_{i+1} \times q_j$ will be merged by a thread so we set $mergeG := true$ to start the global heap.

The global function can steal a row if $g_i \cdot q_j$ is merged by the local heap and q_{j+1} does not exist, or if terms from the global heap are merged when the local heap is empty. This second case is needed at the end of the division when there are no more quotient terms. The global function must keep stealing rows to allow the threads to progress.

The general idea is to maintain a gap of $s-1$ monomials between the global function and all the threads. When the global function merges the last term of row g_i, it steals row g_{i+s-1} if it has not already done so. This allows a thread to merge to the end of row g_{i+s}. Once all of its assigned terms have been merged, the global function steals a row for each distinct monomial it encounters. This allows the threads to continue merging terms without any extra synchronization, as long as they send zero terms to the global function to be merged.

Algorithm: Parallel Sparse Polynomial Division.

```
Input:    $f, g \in \mathbb{Z}[x_1, \ldots, x_n]$, number of threads $p$.
Output:   quotient $q = f/g$, boolean saying if division failed
Globals:  heap $F$, heap $G$, set $Q$, lock $L$, quotient $q$,
          booleans $terminate, failed, mergeG$,
          slack $S$, gap $s$, bound $b$.
  $F :=$ an empty heap ordered by $<$ with max element $F_1$
       for merging the top strip in the global function
  $G :=$ an empty heap ordered by $<$ with max element $G_1$
       for merging the results from all the threads
  $Q :=$ a set of $p$ empty buffers
       from which we insert terms into $G$
  $L :=$ an unheld lock to protect the global function
  $terminate := false$       // set to terminate threads
  $mergeG := false$          // set to merge terms from $G$
  $failed := false$          // set if exact division fails
  $q := 0$                   // the quotient $q = f/g$
  $b := p$                   // rows owned by global function
  $s := b$                   // initial height of the top strip
  $S := 0$                   // "slack" before a row is stolen
  for $i$ from 1 to $p$ do
    spawn $local\_merge(i, p)$
  wait for all threads to complete
  while not $terminate$ do
    $merge\_global()$
  return $(q, failed)$
```

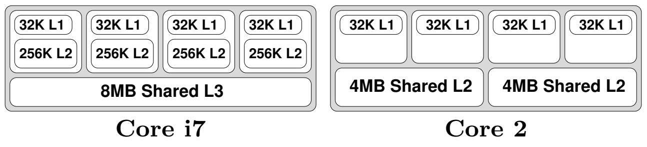
Subroutine: Local Merge.

```
Input:    thread number $r$, total number of threads $p$.
Output:   a subset of terms of $q \cdot g$ are written to $B$.
Locals:   heap $H$, set $E$, monomial $M$, coefficient $C$
          rows stolen $b1$, number of quotient terms $t1$.
Globals:  quotient $q$ and divisor $g$ in $\mathbb{Z}[x_1, \ldots, x_n]$,
          rows stolen $b$, number of quotient terms $t$,
          lock $L$, boolean $terminate$
  $H :=$ an empty heap ordered by $<$ with max element $H_1$
  $E := \{\}$      // terms extracted from $H$
  $t1 := 0$        // number of quotient terms
  $b1 := p$        // number of rows stolen
  // $\{g_1, \cdots, g_p\}$ owned by global function, we start at $g_{p+r}$
  $(i, j) := (p + r, 0)$       // try to insert $g_{p+r} \times q_1$
  goto check_term:
  while $|H| > 0$ do
    // merge all products with largest monomial $M$
    $M := mon(H_1)$;  $C := 0$;  $E := \{\}$;
    while $|H| > 0$ and $mon(H_1) = M$ do
      $(i, j, M) := extract\_max(H)$
      $C := C + cof(g_i) \cdot cof(q_j)$
      $E := E \cup \{(i, j)\}$
    insert term $(C, M)$ into the buffer $B$
    // for each extracted term insert next term into heap
    for all $(i, j) \in E$ do
      // insert first element of next row
      if $j = 1$ and $i + p \leq \#g$ and $bound(q_1) < i + p$ then
        insert $g_{i+p} \times q_1$ into $H$
      check_term:
      // loop until $g_i \times q_{j+1}$ can be inserted or discarded
      while $j = t1$ and $i > b1$ do
        if $trylock(L)$ then
          $global\_merge()$
          $release(L)$
        else
          sleep for 10 microseconds
        $b1 := b$       // update rows stolen
        $read\_barrier()$
        $t1 := t$       // update number of quotient terms
        if $terminate$ then return
      if $j < t1$ and $bound(q_{j+1}) < i$ then
        insert $g_i \times q_{j+1}$ into $H$
  $close(B)$
  return
```

```
Subroutine: Global Merge.
Output:  terms of the quotient are written to $q$.
Locals:  coefficient $C$, monomial $M$, buffer $B$,
         booleans $stealG$, $stealL$.
Globals: heaps $F$ and $G$, sets $P$ and $Q$, polynomials $f, g, q$,
         rows stolen $b$, number of quotient terms $t$,
         booleans $terminate, failed, mergeG$,
         index $k$ into $f$, initial height $s$, slack $S$.
  if $terminate$ then return
  if $mergeG$ then        // insert terms into global heap $G$
    for all $B$ in $Q$ do
      if $B$ is not empty then
        extract next term $(C, M)$ from buffer $B$
        insert $[\,B, C, M\,]$ into heap $G$
        $Q := Q \setminus \{B\}$
      else if not $is\_closed(B)$ then goto done:
  // 3-way comparison of dividend, local heap, global heap
  // $u, v, w$ is set to true or false to merge terms from each
  $C := 0$;  $u := (k \le \#f)$;  $v := (|F| > 0)$;  $w := (|G| > 0)$;
  $stealG := w$ and not $v$;  $stealL := false$;
  if $u$ and $v$  and $mon(f_k) < mon(F_1)$  then $u := false$
  if $u$ and $w$ and $mon(f_k) < mon(G_1)$ then $u := false$
  if $v$ and $u$  and $mon(F_1) < mon(f_k)$  then $v := false$
  if $v$ and $w$ and $mon(F_1) < mon(G_1)$ then $v := false$
  if $w$ and $u$ and $mon(G_1) < mon(f_k)$ then $w := false$
  if $w$ and $v$ and $mon(G_1) < mon(F_1)$ then $w := false$
  if not ($u$ or $v$ or $w$) then        // no terms to merge
    $terminate := true$                  // division complete
  if $u$ then        // merge a term from the dividend
    $C := C + cof(f_k)$
    $M := mon(f_k)$
    $k := k + 1$
  if $v$ then        // merge terms from local heap $F$
    $P := \{\}$
    $M := mon(F_1)$
    while $|F| > 0$ and $mon(F_1) = M$ do
      $(i, j, M) := extract\_max(F)$
      $C := C + cof(g_i) \cdot cof(q_j)$
      $P := P \cup \{(i, j)\}$
    for all $(i, j) \in P$ do
      if $j < \#q$ then
        insert $g_i \times q_{j+1}$ into $F$
      else $stealL := true$
      if $i < \#g$ and $g_{i+1}$ has no term in $F$ then
        if $i + 1 \le bound(q_j)$ then
          insert $g_{i+1} \times q_j$ into $F$
        else     // start merging global heap
          $mergeG := true$
  if $w$ then        // merge terms from global heap $G$
    $Q := \{\}$
    $M := mon(G_1)$
    while $|G| > 0$ and $mon(G_1) = M$ do
      $(B, K, M) := extract\_max(G)$
      $C := C - K$
      $Q := Q \cup \{B\}$
  if $C = 0$ then goto done:
  // compute a new quotient term
  if $LM(g) \mid M$ and $LC(g) \mid C$ then
    $q_{t+1} := (C/LC(g), M/LM(g))$
    $bound(q_{t+1}) := b$
    $write\_barrier()$     // commit term to memory
    $t := t + 1$           // make term visible
    $S := b - s$           // set slack
    if $\#g > 1$ and $g_2$ has no product in $G$ then
      insert $g_2 \times q_t$ into $G$
  else                     // division failed
    $terminate := true$
    $failed := true$
done:    // steal row if local heap empty or product dropped
  if ($stealG$ or $stealL$) and $b < \#g$ then
    if $S > 0$ then $S := S - 1$      // reduce slack
    else $b := b + 1$                 // steal a new row
  return
```

3. BENCHMARKS

We retained the benchmark setup of [11] to allow for easy comparison of the parallel sparse polynomial multiplication and division algorithms. We use two quad core processors: an Intel Core i7 920 2.66GHz and a Core 2 Q6600 2.4GHz. These processors are shown below. The Core i7 has 256KB of dedicated L2 cache per core. We get superlinear speedup by using more of the faster cache in the parallel algorithms. In all of the benchmarks our time for one thread denotes a sequential time for an algorithm from [13].



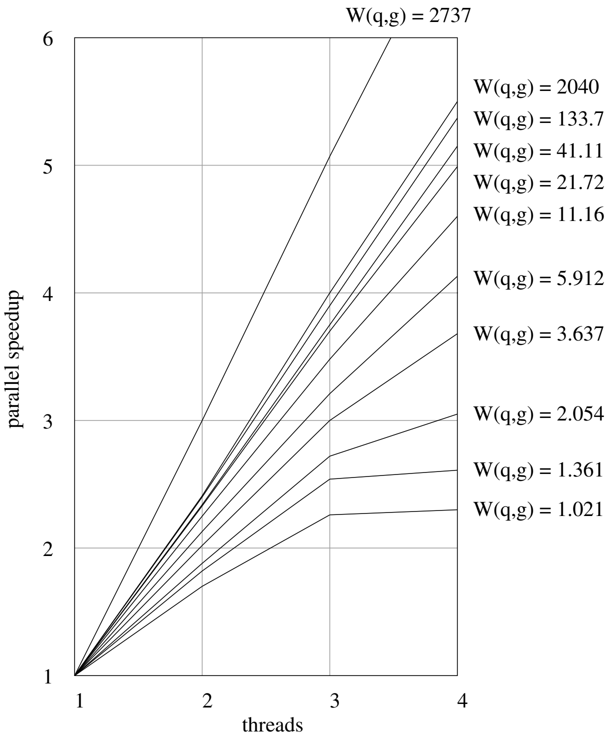
3.1 Sparsity and Speedup

We created random univariate polynomials with different sparsities and multiplied them modulo 32003 as in [11]. The polynomials have 8192 terms. We then divide their product by one of the polynomials modulo 32003. The graph shows the speedup obtained at different sparsities on the Core i7.

For division we measure sparsity as the *work per term* to multiply the quotient and the divisor. That is, for $f/g = q$ $W(q, g) = (\#q \cdot \#g)/\#(q \cdot g)$. This makes our graph below directly comparable to the one for multiplication in [11].

Figure 5: Sparsity vs. Parallel Speedup over $\mathbb{Z}_p$

(totally sparse) $1 \le W(q, g) \le 4096.25$ (totally dense)



The results in Figure 5 are generally good, but the curve for extremely sparse problems flattens out. We are not able to fully utilize all the cores to maintain parallel speedup as $W(q, g) \to 1$. Otherwise, our results here are comparable to those for parallel multiplication in [11]. We obtained linear speedup in the completely dense case.

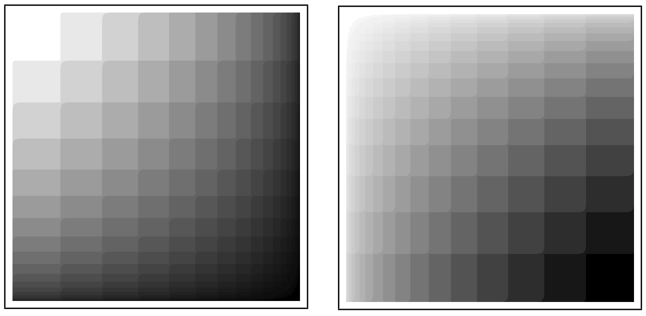
Our throughput here is limited by the dependencies of q, which are triangular in shape. The computation of quotient terms is thus tightly coupled to the merging in the threads, and our global function can not stay ahead. This forces the threads to wait for quotient terms.

3.2 Dense Benchmark

Let $g = (1 + x + y + z + t)^{30}$. We compute $f = g \cdot (g + 1)$ and divide f/g. The quotient and divisor have 46376 terms and 61 bit coefficients. The dividend has 635376 terms and 128 bit coefficients. This problem is due to Fateman [3].

Unlike [11], we also test graded lexicographical order with $x > y > z > t$. This choice of order produces the monomial structure below. The upper left block is 5456×5456 terms consisting of all the products of total degree 60. It must be merged in its entirety to compute the 5457th quotient term, which forces our global function to steal 5455 rows. Despite this difficulty, the performance of our algorithm was good.

Figure 6: Fateman Benchmark



In addition to our software sdmp, we timed Magma 2.16, Singular 3-1-0, and Pari 2.3.3. Magma now also uses heaps to do polynomial multiplication and division. Singular uses a divide-and-conquer algorithm to multiply and a recursive sparse algorithm to divide. Pari uses recursive dense and it supports division only in the univariate sense.

Table 1: Dense benchmark $\mathbb{Z}_{32003}$, $W(f,g) = 3332$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	11.68 s	5.87x	15.10 s	5.42x
	3	16.52 s	4.15x	21.94 s	3.73x
	2	27.83 s	2.46x	37.07 s	2.21x
	1	68.59 s		81.93 s	
sdmp (grlex)	4	11.20 s	6.12x	15.37 s	5.43x
	3	15.94 s	4.30x	21.22 s	3.93x
	2	27.56 s	2.49x	35.01 s	2.38x
	1	68.59 s		83.54 s	
Singular 3-1-0	1	152.65 s		105.26 s	
Magma 2.16-7	1	134.29 s		299.29 s	
Pari 2.3.3	1	795.22 s		438.62 s	
Core 2	**threads**	$f = q \cdot g$		$q = f/g$	
sdmp	4	13.86 s	4.25x	17.82 s	3.80x
	3	19.06 s	3.09x	23.93 s	2.83x
	2	29.82 s	1.97x	35.24 s	1.92x
	1	58.91 s		67.69 s	
sdmp (grlex)	4	13.93 s	4.34x	18.42 s	3.74x
	3	19.19 s	3.15x	23.97 s	2.87x
	2	27.58 s	2.19x	35.06 s	1.96x
	1	60.50 s		68.87 s	
Singular 3-1-0	1	273.05 s		150.36 s	
Magma 2.16-7	1	139.98 s		446.57 s	
Pari 2.3.3	1	942.78 s		520.15 s	

Table 2: Dense benchmark $\mathbb{Z}$, $W(f,g) = 3332$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	11.33 s	6.25x	15.18 s	5.78x
	3	16.30 s	4.34x	21.94 s	4.00x
	2	28.01 s	2.53x	37.03 s	2.37x
	1	70.81 s		87.68 s	
sdmp (grlex)	4	11.50 s	6.15x	15.57 s	5.72x
	3	16.33 s	4.33x	21.36 s	4.17x
	2	28.31 s	2.50x	35.34 s	2.52x
	1	70.75 s		89.11 s	
Singular 3-1-0	1	817.43 s		296.75 s	
Magma 2.16-7	1	359.98 s		441.43 s	
Pari 2.3.3	1	651.02 s		354.82 s	
Core 2	**threads**	$f = q \cdot g$		$q = f/g$	
sdmp	4	14.20 s	4.25x	17.88 s	4.28x
	3	19.48 s	3.10x	24.15 s	3.17x
	2	30.35 s	1.99x	35.29 s	2.17x
	1	60.38 s		76.59 s	
sdmp (grlex)	4	14.27 s	4.24x	18.59 s	4.20x
	3	19.69 s	3.07x	24.20 s	3.22x
	2	28.11 s	2.15x	35.39 s	2.20x
	1	60.50 s		78.09 s	
Singular 3-1-0	1	1163.49 s		349.06 s	
Magma 2.16-7	1	361.42 s		597.51 s	
Pari 2.3.3	1	692.59 s		382.74 s	

Tables 1 and 2 present times to multiply and divide with coefficients in $\mathbb{Z}/32003$ and $\mathbb{Z}$. The parallel heap algorithms generally achieve superlinear speedup on the Core i7 due to their use of extra L2 cache. On the Core 2 architecture the speedup is still fairly good. The sdmp times are similar for $\mathbb{Z}$ and $\mathbb{Z}_p$ because our integer arithmetic assumes word size coefficients. Magma and Singular use faster representations for $\mathbb{Z}_p$ when p is less than 24 or 31 bits.

3.3 Sparse Benchmark

Our last benchmark is a sparse problem with an irregular block pattern. Let $g = (1 + x + y + 2z^2 + 3t^3 + 5u^5)^{12}$ and $q = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{12}$. We compute $f = q \cdot g$ and divide f/g in lexicographical order $x > y > z > t > u$. The quotient q and the divisor g have 6188 terms and their coefficients are 37 bits. The dividend f has 5.8×10^6 terms and its coefficients are 75 bits.

Table 3: Sparse benchmark $\mathbb{Z}_{32003}$, $W(f,g) = 6.577$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.547 s	2.67x	0.589 s	3.40x
	3	0.658 s	2.22x	0.707 s	2.83x
	2	0.915 s	1.60x	1.004 s	1.99x
	1	1.462 s		2.006 s	
Singular 3-1-0	1	10.520 s		20.860 s	
Magma 2.16-7	1	4.710 s		66.540 s	
Pari 2.3.3	1	113.786 s		65.314 s	
Core 2	**threads**	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.663 s	2.67x	0.741 s	3.16x
	3	0.813 s	2.18x	0.858 s	2.73x
	2	1.081 s	1.64x	1.196 s	1.96x
	1	1.774 s		2.343 s	
Singular 3-1-0	1	16.940 s		26.140 s	
Magma 2.16-7	1	5.770 s		127.750 s	
Pari 2.3.3	1	132.388 s		74.991 s	

We were surprised that the speedup for division could be higher than for multiplication, but the sequential algorithm for division seems to have lower relative performance. This could be due to the extra work it performs to maintain low complexity. Unlike the parallel algorithm, the method from [13] is highly efficient if the quotient is small.

Table 4: Sparse benchmark $\mathbb{Z}$, $W(f,g) = 6.577$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.584 s	2.65x	0.675 s	3.23x
	3	0.738 s	2.10x	0.791 s	2.76x
	2	1.002 s	1.54x	1.102 s	1.75x
	1	1.548 s		2.182 s	
Singular 3-1-0	1	25.660 s		32.400 s	
Magma 2.16-7	1	7.780 s		80.200 s	
Pari 2.3.3	1	59.823 s		34.566 s	

Core 2	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.752 s	2.33x	0.817 s	3.02x
	3	0.903 s	1.95x	0.951 s	2.60x
	2	1.205 s	1.46x	1.289 s	1.91x
	1	1.759 s		2.468 s	
Singular 3-1-0	1	36.840 s		40.090 s	
Magma 2.16-7	1	9.930 s		137.460 s	
Pari 2.3.3	1	65.362 s		37.582 s	

4. CONCLUSION

We presented a fast new parallel algorithm for division of sparse polynomials on multicore processors. The algorithm was designed to achieve very high levels of performance and superlinear speedup on a problem that could be considered inherently sequential. Our benchmarks show that with few exceptions, this was achieved in practice. This has made us cautiously optimistic towards parallel computer algebra.

Our next task is to integrate the routines into the Maple computer algebra system. By parallelizing basic operations at a low level, we hope to obtain noticable parallel speedup for users and library code at the top level.

Acknowledgements

We gratefully acknowledge the MITACS NCE of Canada and NSERC of Canada for funding this work, and we thank the anonymous referees for their helpful comments.

5. REFERENCES

[1] D. Bini, V. Pan. Improved parallel polynomial division. *SIAM J. Comp.* **22** (3) 617–626, 1993.

[2] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symb. Comp.*, **24** (3-4) 235–265, 1997

[3] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *ACM SIGSAM Bulletin*, **37** (1) 4–15, 2003.

[4] M. Gastineau, J. Laskar. Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. *Proc. ICCS 2006*, Springer LNCS 3992, 446–453.

[5] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3.1.0 – A computer algebra system for polynomial computations, 2009. `http://www.singular.uni-kl.de`

[6] L.H. Harper, T.H. Payne, J.E. Savage, E. Straus. Sorting X+Y. Comm. ACM 18 (6), pp. 347–349, 1975.

[7] S.C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, **8** (3) 63–71, 1974.

[8] X. Li and M. Moreno Maza. Multithreaded parallel implementation of arithmetic operations modulo a triangular set. *Proc. of PASCO '07*, ACM Press, 53–59.

[9] T. Mattson, B. Sanders, B. Massingill. Patterns for Parallel Programming. Addison-Wesley, 2004.

[10] M. Matooane. Parallel Systems in Symbolic and Algebraic Computation. Ph.D Thesis, Cambridge, 2002.

[11] M. Monagan, R. Pearce. Parallel Sparse Polynomial Multiplication Using Heaps. *Proc. of ISSAC 2009*, 295–315.

[12] M. Monagan, R. Pearce. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. *Proc. of CASC 2007*, Springer LNCS 4770, 295–315.

[13] M. Monagan, R. Pearce. Sparse Polynomial Division Using a Heap. *submitted to J. Symb. Comp.*, October 2008.

[14] A. Norman, J. Fitch. CABAL: Polynomial and power series algebra on a parallel computer. *Proc. of PASCO '97*, ACM Press, pp. 196–203.

[15] P. Wang. Parallel Polynomial Operations on SMPs. *J. Symbolic. Comp.,* **21** 397–410, 1996.

A high-performance algorithm for calculating cyclotomic polynomials.

Andrew Arnold
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
ada26@sfu.ca.

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
mmonagan@cecm.sfu.ca.

ABSTRACT

The n_{th} cyclotomic polynomial, $\Phi_n(z)$, is the monic polynomial whose $\phi(n)$ distinct roots are the n_{th} primitive roots of unity. $\Phi_n(z)$ can be computed efficiently as a quotient of terms of the form $(1 - z^d)$ by way of a method the authors call the Sparse Power Series algorithm. We improve on this algorithm in three steps, ultimately deriving a fast, recursive algorithm to calculate $\Phi_n(z)$. The new algorithm, which we have implemented in C, allows us to compute $\Phi_n(z)$ for $n > 10^9$ in less than one minute.

Categories and Subject Descriptors:
G.0 [Mathematics of Computing]: General
General Terms: Algorithms
Keywords: Cyclotomic Polynomials

1. INTRODUCTION

The n_{th} cyclotomic polynomial, $\Phi_n(z)$, is the minimal polynomial over $\mathbb{Q}$ of the n_{th} primitive roots of unity.

$$\Phi_n(z) = \prod_{\substack{j=1 \\ \gcd(j,n)=1}}^{n} \left(z - e^{\frac{2\pi i}{n} j}\right). \tag{1.1}$$

We let the *index* of $\Phi_n(z)$ be n and the *order* of $\Phi_n(z)$ be the number of distinct odd prime divisors of n. The n_{th} inverse cyclotomic polynomial, $\Psi_n(z)$, is the monic polynomial whose roots are the n_{th} non-primitve roots of unity.

$$\Psi_n(z) = \prod_{\substack{j=1 \\ \gcd(j,n)>1}}^{n} \left(z - e^{\frac{2\pi i}{n} j}\right) = \frac{z^n - 1}{\Phi_n(z)}. \tag{1.2}$$

We denote by $A(n)$ the *height* of $\Phi_n(z)$, that is, the largest coefficient in magnitude of $\Phi_n(z)$. It is well known that for $n < 105$, $A(n) = 1$ but for $n = 105$, $A(n) = 2$. The smallest n with $A(n) > 2$ is $n = 385$ where $A(n) = 3$. Although the heights appear to grow very slowly, Paul Erdős proved in [2] that $A(n)$ is not bounded above by any polynomial in n, that is, for any constant $c > 0$, there exists n such that $A(n) > n^c$. Maier showed that the set of n for which $A(n) > n^c$ has positive lower density. A natural question to ask is, what is the first n for which $A(n) > n$?

In earlier work [1], we developed two asymptotically fast algorithms to compute $\Phi_n(z)$. The first algorithm, which we call the FFT algorithm, uses the Fast Fourier Transform to perform a sequence of polynomial exact divisions in $\mathbb{Z}[z]$ modulo a prime q. Using this algorithm we found the smallest n such that $A(n) > n$, namely for $n = 1,181,895$, the height $A(n) = 14,102,773$. Here $n = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29$. To find $\Phi_n(z)$ with larger height, we tried simply multiplying this n by additional primes. In this way we found an n with $A(n) > n^2$ and several $n > 10^9$ with $A(n) > n^4$, the latter requiring the use of a supercomputer with a lot of RAM.

The second algorithm, which we call the Sparse Power Series (SPS) algorithm, does a sequence of sparse series multiplications and divisions in $O(2^k \phi(n))$ integer arithmetic operations. Although not asymptotically faster than the FFT algorithm, it turns out that because the SPS algorithm only needs integer additions and subtractions, it is considerably faster (more than 20 times - see section 4) than the FFT algorithm. Using the SPS algorithm we found the smallest n with $A(n) > n^2$, $A(n) > n^3$ and $A(n) > n^4$, namely $n = 43,730,115$, $n = 416,690,995$, and $1,880,394,945$, respectively, as well as other new results. One of the difficulties when $n > 10^9$ is space. For such n, even storing $\Phi_n(z)$ requires many gigabytes of memory. The SPS algorithm has a substantial space advantage over the FFT algorithm. It has now been implemented in the Sage and Maple 13 computer algebra systems.

In this paper we present a fast recursive algorithm to calculate $\Phi_n(z)$ and $\Psi_n(z)$. It improves on the sparse power series (**SPS**) algorithm by approximately another factor of 10 (see section 4). To give one specific benchmark; Yoichi in [4, 5] found $A(n)$ for n the product of the first 7 odd primes but was unable to determine $A(n)$ for n the product of the first 8 primes. We used the FFT algorithm to find $A(n)$ for n the product of the first 9 odd primes in approx. 12 hours. The SPS algorithm takes 7 minutes and our new algorithm takes 50 seconds. A challenge problem given to us by Noe [6] is to compute $\Phi_n(z)$ for $n = 99,660,932,085$ which we expect will have a huge height. The main difficulty now is space; for the mentioned unsolved problem, the set of coefficients of $\Phi_n(z)$, stored as 320-bit integers (which we estimate will be sufficient) requires over 750 GB of space.

Our paper is organized as follows. Section 2 presents identities involving $z^n - 1$, $\Phi_n(z)$ and $\Psi_n(z)$ used in the algorithms, and the basic algorithm used for the FFT approach.

Section 3 details the Sparse Power Series algorithm for computing $\Phi_n(z)$ and introduces a similar algorithm for computing $\Psi_n(z)$, then develops improvements in three steps. The third step makes the algorithm recursive. Section 4 presents some timings comparing the FFT algorithm, the original SPS algorithm, and the three improvements.

2. USEFUL IDENTITIES OF CYCLOTOMIC POLYNOMIALS

Before describing the algorithms, we establish some basic identities of cyclotomic polynomials. First, as the roots of $\Phi_n(z)$ and $\Psi_n(z)$ consist of all n_{th} roots of unity, we have

$$\Phi_n(z)\Psi_n(z) = \prod_{j=0}^{n-1}(z - e^{\frac{2\pi j}{n}i}) = z^n - 1. \tag{2.1}$$

Every n_{th} root of unity is a d_{th} primitive root of unity for some unique $d|n$. Conversely, if $d|n$, every d_{th} primitive root of unity is trivially an n_{th} root of unity. As such,

$$\prod_{d|n}\Phi_d(z) = z^n - 1. \tag{2.2}$$

Applying the Möbius inversion formula to (2.2), we have

$$\Phi_n(z) = \prod_{d|n}(z^d - 1)^{\mu(\frac{n}{d})}, \tag{2.3}$$

where μ is the Möbius function. From (2.1) and (2.3) we obtain a similar identity for $\Psi_n(z)$.

$$\Psi_n(z) = \prod_{d|n,d<n}(z^d - 1)^{-\mu(\frac{n}{d})}, \tag{2.4}$$

and from (2.1) and (2.2), we have that

$$\Psi_n(z) = \prod_{d|n,d<n}\Phi_d(z). \tag{2.5}$$

Thus $\Psi_n(z)$ is a product cyclotomic polynomials.

Given $\Phi_1(z) = z - 1$ and $\Psi_1(z) = 1$, we can compute all cyclotomic polynomials using the following lemmas.

Lemma 1. *If p, q primes such that $p \nmid n$ and $q|n$ then*

$$\Phi_{np}(z) = \frac{\Phi_n(z^p)}{\Phi_n(z)}, \tag{2.6a}$$

$$\Phi_{nq}(z) = \Phi_n(z^q), \tag{2.6b}$$

$$\Psi_{np}(z) = \Psi_n(z^p)\Phi_n(z), \text{ and} \tag{2.6c}$$

$$\Psi_{nq}(z) = \Psi_n(z^q). \tag{2.6d}$$

Lemma 2. *If $n > 1$ is odd then*

$$\Phi_{2n}(z) = \Phi_n(-z) \text{ and} \tag{2.7a}$$

$$\Psi_{2n}(z) = -\Psi_n(-z)(z^n + 1). \tag{2.7b}$$

Lemmas 1 and 2 are well-known. One can prove these identities by equating roots of both sides of the respective equations. Given $\Phi_n(z)\Psi_n(z) = z^n - 1$, the identities for $\Psi_n(z)$ (2.6c), (2.6d) and (2.7b) can be easily derived from (2.6a), (2.6b) and (2.7a), their respective analogs for $\Phi_n(z)$.

These lemmas give us a means to calculate $\Phi_n(z)$. For example, for $n = 150 = 2 \cdot 3 \cdot 5^2$ we have

$$\begin{aligned}
\Phi_3(z) &= \tfrac{\Phi_1(z^3)}{\Phi_1(z)} = \tfrac{z^3-1}{z-1} = z^2 + z + 1, \text{ and}\\
\Phi_{15}(z) &= \tfrac{\Phi_3(z^5)}{\Phi_3(z)} = \tfrac{z^{10}+z^5+1}{z^2+z+1}\\
&= z^8 - z^7 + z^5 - z^4 + z^3 - z + 1, \text{ by (2.6a).}\\
\Phi_{75}(z) &= \Phi_{15}(z^5) \text{ by (2.6b),}\\
&= z^{40} - z^{35} + z^{25} - z^{20} + z^{15} - z^5 + 1.\\
\Phi_{150} &= \Phi_{75}(-z) \text{ by (2.7a),}\\
&= z^{40} + z^{35} - z^{25} - z^{20} + z^{15} + z^5 + 1
\end{aligned}$$

We formally describe this approach in algorithm 1.

Algorithm 1: Computing $\Phi_n(z)$ by repeated polynomial division

```
Input: $n = 2^{e_0}p_1^{e_1}p_2^{e_2}\cdots p_k^{e_k}$, where $2 < p_1 < \cdots < p_k$,
       $e_0 \geq 0$, and $e_i > 0$ for $1 \leq i \leq k$
Output: $\Phi_n(z)$
1  $m \longleftarrow 1$
2  $\Phi_m(z) \longleftarrow z - 1$
3  for $i = 1$ to $k$ do
4      $\Phi_{mp_i}(z) \longleftarrow \Phi_m(z^{p_i})/\Phi_m(z)$        // By (2.6a)
5      $m \longleftarrow m \cdot p_i$
6  if $e_0 > 0$ then
7      $\Phi_{2m}(z) \longleftarrow \Phi_m(-z)$                 // By (2.7a)
8      $m \longleftarrow 2m$
   // $m$ is the largest squarefree divisor of $n$ now
9  $s \longleftarrow n/m$
10 $\Phi_n(z) \longleftarrow \Phi_m(z^s)$                       // By (2.6b)
   return $\Phi_n(z)$
```

While algorithm 1 is beautifully simple, it is not nearly the fastest way to compute $\Phi_n(z)$, particularly if we use classical polynomial division to calculate the polynomial quotient $\Phi_m(z^{p_i})/\Phi_m(z)$ (line 4). For even though the numerator is sparse, the denominator and quotient are typically dense.

We implemented algorithm 1 using the discrete fast Fourier transform (FFT) to perform $\Phi_m(z^{p_i})/\Phi_m(z)$ fast. This is done modulo suitably chosen primes q_j. The cost of computing one image of $\Phi_n(z)$ modulo a prime q is $\mathcal{O}(\phi(n)\log\phi(n))$ arithmetic operations in $\mathbb{Z}_q$. With each iteration of the loop on line 3, the degree of the resulting polynomial grows by a factor. As such the cost of this approach is dominated by the last division. For a description of the discrete FFT, we refer the reader to [3].

We compute images of $\Phi_n(z)$ modulo sufficiently many primes and recover the integer coefficients of $\Phi_n(z)$ using Chinese remaindering. In order to apply the FFT modulo q, we need a prime q with $2^k|q-1$ and $2^k > \phi(n)$, the degree of the output $\Phi_n(z)$. Since for $n > 10^9$ there are no such 32 bit primes, we used used 42-bit primes with arithmetic modulo q coded using 64-bit machine integer arithmetic.

It follows from lemma 1 that for primes $p|n$, the set of nonzero coefficients of $\Phi_{np}(z)$ and $\Phi_n(z)$ are the same. Similarly, by lemma 2 we have $A(n) = A(2n)$ for odd n. Thus $\Phi_n(z)$ for even or nonsquarefree n, for our purposes, are uninteresting. Moreover, if $\bar{n}$ is the largest odd squarefree divisor of n, then it is easy to obtain $\Phi_n(z)$ from $\Phi_{\bar{n}}(z)$.

For the remainder of this paper, we only consider $\Phi_n(z)$ for squarefree, odd n.

3. HIGH-PERFORMANCE ALGORITHMS FOR COMPUTING $\Phi_N(Z)$

Our C implementation of the FFT-based approach proved to be faster than methods available via computer algebra systems at the time. Using this method we were able to compute examples of $\Phi_n(z)$ of degree in the billions and height well beyond that. However, the FFT approach was eclipsed by a faster algorithm.

For $n > 1$, the number of squarefree divisors of n is even. As such we can rewrite (2.3) as

$$\Phi_n(z) = \prod_{d|n}(1 - z^d)^{\mu(\frac{n}{d})}. \tag{3.1}$$

As $\Phi_n(z)\Psi_n(z) = z^n - 1$ we also have, for $n > 1$,

$$\Psi_n(z) = -\prod_{d|n, d<n}(1 - z^d)^{-\mu(\frac{n}{d})}. \tag{3.2}$$

In our implementation of every algorithm presented in section 3, we compute $\Phi_n(z)$ as the product of terms $(1-z^d)^{\pm 1}$ appearing in (3.1); however, in the identities we present in this section it is often less cumbersome to express $\Phi_n(z)$ in terms of $(z^d-1)^{\pm 1}$. We refer to the $(1-z^d)^{\pm 1}$ (alternatively $(z^d-1)^{\pm 1}$) comprising $\Phi_n(z)$ as the *subterms* of $\Phi_n(z)$.

Given that the power series expansion of $(1 - z^d)^{-1}$ is $(1 + z^d + z^{2d} + z^{3d} + \dots)$, it becomes equally easy to either multiply or divide by $(1 - z^d)$. $\Phi_n(z)$ can thus be computed as the truncated power series of

$$\prod_{\mu(\frac{n}{d})=1}(1 - z^d)\cdot \prod_{\mu(\frac{n}{d})=-1}(1 + z^d + z^{2d} + \dots), \tag{3.3}$$

as described in procedure SPS.

Procedure SPS(n), computing $\Phi_n(z)$ as a quotient of sparse power series

The Sparse Power Series (SPS) Algorithm

```
Input: $n$ a squarefree, odd integer
Output: $a(0), \dots, a(\frac{\phi(n)}{2})$, the first half of the
        coefficients of $\Phi_n(z)$
// we compute terms up to degree $D$
1 $D \longleftarrow \frac{\phi(n)}{2}$, $a(0) \longleftarrow 1$
2 for $1 \le i \le M$ do $a(i) \longleftarrow 0$
3 for $d|n$ such that $d < n$ do
4     if $\mu(\frac{n}{d}) = 1$ then            // multiply by $1 - z^d$
5         for $i = D$ down to $d$ by $-1$ do
6             $a(i) \longleftarrow a(i) - a(i-d)$
7     else                                      // divide by $1 - z^d$
8         for $i = d$ to $D$ do
9             $a(i) \longleftarrow a(i) + a(i-d)$
return $a(0), a(1), \dots a(D)$
```

The brunt of the work of the SPS algorithm takes place on lines 6 and 9, where we multiply by $(1 - z^d)$ and $(1 - z^d)^{-1}$ respectively. In either case, computing this product, truncated to degree $D = \phi(n)/2$, takes $\mathcal{O}(D - d) \in \mathcal{O}(\phi(n))$ arithmetic operations in $\mathbb{Z}$. As n, a product of k distinct primes, has 2^k positive divisors, the SPS method requires some $\mathcal{O}(2^k \cdot \phi(n))$ operations to compute $\Phi_n(z)$ of order k. Note that while $1 - z^n$ appears in (3.1), we do not multiply by $1 - z^n$ is algorithm SPS, as it does not affect our result. This is because $1 - z^n \equiv 1 \pmod{z^D}$ for $D = \phi(n)/2$.

Using the analog identity for $\Psi_n(z)$, (3.2), we derive a very similar method for $\Psi_n(z)$, described by procedure SPS-Psi.

Procedure SPS-Psi(n), computing $\Psi_n(z)$ as a quotient of sparse power series

A Sparse Power Series Algorithm for $\Psi_n(z)$

```
Input: $n$ a squarefree, odd integer
Output: $b(0), \dots, b(\lfloor\frac{n-\phi(n)}{2}\rfloor)$, the first half of the
        coefficients of $\Psi_n(z)$
1 $D \longleftarrow \lfloor\frac{n-\phi(n)}{2}\rfloor$, $b(0) \longleftarrow 1$
2 for $1 \le i \le D$ do $b(i) \longleftarrow 0$
3 for $d|n$ such that $d < n$ do
4     if $\mu(\frac{n}{d}) = -1$ then           // multiply by $1 - z^d$
5         for $i = D$ down to $d$ by $-1$ do
6             $b(i) \longleftarrow b(i) - b(i-d)$
7     else                                      // divide by $1 - z^d$
8         for $i = d$ to $D$ do
9             $b(i) \longleftarrow b(i) + b(i-d)$
return $b(0), b(1), \dots, b(D)$
```

By a similar analysis as for SPS, we see that procedure SPS-Psi requires $\mathcal{O}(2^k(n - \phi(n)) \in \mathcal{O}(2^k \cdot n)$ arithmetic operations.

3.1 The palindromic property of cyclotomic coefficients

In the SPS and SPS-Psi methods we truncate to half the degree of $\Phi_n(z)$ and $\Psi_n(z)$ respectfully. This is because it is trivial to obtain the ter,s of higher degree. For $n > 1$ the coefficients of $\Phi_n(z)$ are *palindromic* and those of $\Psi_n(z)$ are *antipalindromic*. That is, given

$$\Phi_n(z) = \sum_{i=0}^{\phi(n)} a(i)z^i \qquad \text{and} \qquad \Psi_n(z) = \sum_{i=0}^{n-\phi(n)} b(i)z^i,$$

we have that $a(i) = a(\phi(n) - i)$ and $b(i) = -b(n - \phi(n) - i)$. We prove a related result, which will bode useful in subsequent algorithms.

Lemma 3. *Let*

$$f(z) = \Phi_{n_1}(z)\cdot\Phi_{n_2}(z)\cdots\Phi_{n_s}(z) = \sum_{i=0}^{D} c(i)z^i \tag{3.4}$$

be a product of cyclotomic polynomials such that n_j is odd for $1 \le j \le s$. Then $c(i) = (-1)^D c(D - i)$ for $0 \le i < D$. In other words, if D is odd $f(z)$ is antipalindromic, and if D is even $f(z)$ is palindromic.

Proof. Clearly $f(z)$ is monic. If ω is a root of f, then ω is an $(n_j)_{th}$ primitive root of unity for some j such that $1 \le j \le s$. In which case, ω^{-1} is also an $(n_j)_{th}$ primitive root of unity and hence is also a root of $f(z)$. Set

$$g(z) = z^D f(z^{-1}) = \sum_{i=0}^{D} c(D - i)(z). \tag{3.5}$$

$g(z)$ is a polynomial of degree D with leading coefficient $c(0)$ whose roots are the roots of f. Thus $f(z)$ and $g(z)$ only differ by the constant factor $c(0)/c(D) = c(0)$. We need only resolve $c(0)$. To that end, we observe that $\phi(n)$ is even for odd $n > 1$, and $\phi(1) = 1$. Thus $r \equiv D \pmod 2$, where r is the cardinality of

$$\{j : 1 \le j \le s \text{ and } n_j = 1\}. \tag{3.6}$$

The constant term of f, $c(0)$, is the product of the constant terms of the $\Phi_{n_j}(z)$ in (3.4). Since the constant term of $\Phi_1(z) = z - 1$ is -1, and by (3.1), the constant term of $\Phi_n(z)$ is 1 for $n > 1$, we have that $c(0) = (-1)^r = (-1)^D$, completing the proof. □

We note that lemma 3 does not hold if we relax the restriction that n_j must be odd in (3.4). Consider the trivial counterexample $\Phi_2(z) = z+1$. By (2.5), we have that $\Psi_n(z)$ is a product of cyclotomic polynomials, and so lemma 3 applies to $\Psi_n(z)$ for odd n, or any product of the form

$$\Psi_{n_1}(z) \cdot \Psi_{n_2}(z) \cdots \Psi_{n_s}(z), \tag{3.7}$$

where $n_1, n_2, \ldots, n_s$ are all odd.

3.2 Improving the sparse power series method by further truncating degree

The sparse power series algorithm slows appreciably as we calculate $\Phi_n(z)$ for n with increasingly many factors. The slowdown in computing $\Phi_{np}(z)$ compared to $\Phi_n(z)$ is twofold. By introducing a new prime factor p we double the number of subterms $(1-z^d)^{\pm 1}$ in our product (3.1). In addition, the degree of $\Phi_{np}(z)$ is $p-1$ times that of $\Phi_n(z)$, thus increasing the cost of multiplying one subterm $(1 - z^d)^{\pm 1}$ by a factor. For $\Phi_n(z)$ of larger degree the algorithm also exhibits poorer locality.

In procedure SPS, we effectively compute 2^k distinct power series, each a product of subterms $(1-z^d)^{\pm 1}$, each truncated to degree $\phi(n)/2$. We can improve the SPS algorithm if we truncate any intermediate power series to as minimal degree necessary, thereby reducing the number of arithmetic operations and leveraging locality where possible. We let the *degree bound* refer to the degree we must truncate to at some stage in the computation of $\Phi_n(z)$ using the SPS algorithm or a variant thereof.

Depending on the order in which we multiply the subterms of $\Phi_n(z)$, some of the intermediate products of subterms we compute may be polynomials as well. If, at some point of our computation of $\Phi_n(z)$, we have a product of subterms that is a polynomial $f(z)$ of degree D, then $f(z)$ is a product of cyclotomic polynomials satisfying lemma 3 (provided n is odd and squarefree), and we need only truncate to degree at most $\lfloor D/2 \rfloor$ at previous stages of the computation.

Once we have computed $f(z)$, our degree bound may increase. In which case we can generate higher-degree terms of $f(z)$ as necessary using lemma 3.

More generally, if we have some product of subterms of $\Phi_n(z)$ and we know polynomials $f_1(z), f_2(z), \ldots f_s(z)$ of degrees $D_1, D_2, \ldots, D_s$ will occur as products of subterms at later stages of our computation, then we can truncate to $\lfloor D/2 \rfloor$, where $D = \min_{1 \le j \le s} D_s$. Our aim is to order the subterms in an intelligent manner which minimizes the growth of the degree bound over the computation of $\Phi_n(z)$.

To further our aim, we let $n = mp$, where p is the largest prime divisor of n and $m > 1$. In which case

$$\begin{aligned}\Phi_{mp}(z) =& \frac{\Phi_m(z^p)}{\Phi_m(z)} \text{ by lemma 1,} \\ =& \Psi_m(z) \cdot \Phi_m(z^p) \cdot (z^m - 1)^{-1}.\end{aligned} \tag{3.8}$$

By (3.1) and (3.2), we can break $\Psi_m(z)$ and $\Phi_m(z)$ into respective products of subterms.

$$\Phi_n(z) = \left(\prod_{d|m, d<m} (z^d - 1)^{-\mu(\frac{m}{d})}\right)\left(\prod_{d|m} (z^{dp} - 1)^{\mu(\frac{m}{d})}\right)(z^m - 1)^{-1}. \tag{3.9}$$

Thus to compute $\Phi_n(z)$, we can first compute $\Psi_m(z)$, the leftmost product of (3.9) to degree $\frac{m-\phi(m)}{2}$, use the antipalindromic property of $\Psi_m(z)$ to reconstruct its remaining coefficients, and then multiply the remaining subterms as we would in algorithm SPS. Algorithm SPS2 describes the method.

Procedure SPS2(n) : First revision of algorithm **SPS**

Algorithm SPS2: Improved Sparse Power Series

```
Input: $n = mp$, a squarefree, odd integer with greatest
       prime divisor $p$
Output: $a(0), \ldots, a(\frac{\phi(n)}{2})$, the first half of the
        coefficients of $\Phi_n(z)$
// Compute first half of $\Psi_m(z)$
$a(0), a(1), \ldots, a(\lfloor \frac{n-\phi(m)}{2} \rfloor) \longleftarrow$ SPS-Psi($m$)

// Construct other half of $\Psi_m(z)$ using lemma 3
$D \longleftarrow m - \phi(m)$
for $i = \lceil \frac{m-\phi(m)}{2} \rceil$ to $D$ do $a(i) \longleftarrow -a(m - \phi(m) - i)$

// Multiply by $\Phi_m(z^p)$
$D \longleftarrow \frac{\phi(n)}{2}$
$a(m - \phi(m) + 1), a(m - \phi(m) + 2), \ldots, a(D) \longleftarrow 0$
for $d|m$ do
    if $\mu(\frac{n}{d}) = 1$ then
        for $i = D$ down to $d$ by $-1$ do
            $a(i) \longleftarrow a(i) - a(i - dp)$
    else
        for $i = d$ to $D$ do
            $a(i) \longleftarrow a(i) + a(i - dp)$

// Divide by $z^m - 1 = (-1 - z^m - z^{2m} - \ldots)$
for $i = m$ to $D$ do $a(i) \longleftarrow -a(i) - a(i - m)$
return $a(0), a(1), \ldots a(D)$
```

For $n = mp$ with k distinct prime divisors, $\Psi_m(z)$ comprises $2^{k-1} - 1$ of the 2^k subterms of $\Phi_n(z)$. For each of these subterms appearing in $\Psi_m(z)$ we truncate to degree $(m - \phi(m))/2$. The asymptotic operation cost of SPS2 is no different than that of SPS; however, in practise this method cuts the running time in half (see table 1 for timings).

We note that the speed-up is not as substantial for m with very few prime factors. In the event that n is prime, we have $m = 1$ and $\Psi_m(z) = 1$. In such case the execution of SPS and SPS2 are effectively the same. For $n = qp$, a product of two primes with $q < p$, $\Phi_n(z)$ has only four subterms and we only get gains on the single subterm appearing in $\Psi_q(z) = z - 1$. For $\Phi_n(z)$ of low order, the proportion of subterms of $\Phi_n(z)$ appearing in $\Psi_m(z)$ is further from 1/2 compared to $\Phi_n(z)$ for highly composite n (i.e. n for which k is larger). That said, however, $\Phi_n(z)$ is already easy to compute by the original SPS method for $\Phi_n(z)$ of low order, as these cyclotomic polynomials have very few subterms.

3.3 Calculating $\Phi_n(z)$ by way of a product of inverse cyclotomic polynomials

In algorithm SPS2 we bound to a smaller degree than in SPS when multiplying the first $2^{k-1} - 1$ subterms of $\Phi_n(z)$. We are able to lower the degree bound for many of the remaining $2^{k-1}+1$ subterms of $\Phi_n(z)$. To that end we establish the next identity.

Let $n = p_1 p_2 \cdots p_k$, a product of k distinct odd primes. For $1 \le i \le k$, let $m_i = p_1 p_2 \cdots p_{i-1}$ and $e_i = p_{i+1} \cdots p_k$. We set $m_1 = e_k = 1$, and let $e_0 = n$. Note that $n = e_i p_i n_i$ for $1 \le i \le k$. In addition, $e_{i-1} = p_i e_i$ and $m_{i+1} = m_i p_i$. We restate (3.8), which was key to SPS2, as

$$\Phi_n(z) = \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \Phi_{m_k}(z^{e_{k-1}}). \tag{3.10}$$

By repeated application of lemma 1, we have

$$\begin{aligned}\Phi_n(z) &= \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \frac{\Psi_{m_{k-1}}(z^{e_{k-1}})}{(z^{n/p_{k-1}} - 1)} \Phi_{m_{k-1}}(z^{e_{k-2}}), \\ &\cdots \\ &= \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \cdots \frac{\Psi_{m_2}(z^{e_2})}{(z^{n/p_2} - 1)} \frac{\Psi_{m_1}(z^{e_1})}{(z^{n/p_1} - 1)} \Phi_{m_1}(z^{e_0}), \\ &= \left(\prod_{j=1}^{k} \frac{\Psi_{m_j}(z^{e_j})}{(z^{n/p_j} - 1)} \right) \cdot \Phi_{m_1}(z^{e_0}).\end{aligned} \tag{3.11}$$

As $\Psi_{m_1}(z^{e_1}) = \Psi_1(z^{e_1}) = -1$, and $\Phi_1(z^{e_0}) = \Phi_1(z^n) = z^n - 1$, this simplifies to

$$\Phi_n(z) = \prod_{j=2}^{k} \Psi_{m_j}(z^{e_j}) \cdot \prod_{j=1}^{k} (z^{n/p_j} - 1)^{-1} \cdot (z^n - 1) \tag{3.12}$$

For example, for $n = 105 = 3 \cdot 5 \cdot 7$,

$$\Phi_{105}(z) =$$
$$\Psi_{15}(z)\Psi_3(z^7)(z^{15} - 1)^{-1}(z^{21} - 1)^{-1}(z^{35} - 1)^{-1}(z^{105} - 1)$$

As with algorithm SPS2, we first calculate half the terms of $\Psi_{m_k}(z^{e_k}) = \Phi_{p_1 p_2 \cdots p_{k-1}}(z)$, those with degree at most $\lfloor \frac{\phi(m_k)}{2} \rfloor$. We then iteratively compute the product

$$\Psi_{m_k}(z^{e_k}) \cdots \Psi_{m_2}(z^{e_2}) \tag{3.13}$$

from left to right. When calculating the degree of $\Psi_{m_j}(z^{e_j})$ we truncate to degree at most

$$\left\lfloor \frac{1}{2} \prod_{i=j}^{k} (m_i - \phi(m_i)) e_i \right\rfloor, \tag{3.14}$$

half the degree of the product in (3.13). As our intermediate product grows larger we have to truncate to larger degree. The term $\Psi_{m_i}(z^{e_i})$, comprises $2^{i-1} - 1$ subterms of $\Phi_n(z)$. We compute $\Psi_{m_k}(z^{e_k})$ first because that contains 2^{k-1} subterms, nearly half of the 2^k we must multiply by to compute $\Phi_n(z)$, so it is best that we multiply these subterms first before the degree bound swells.

We leverage lemma 3 again when computing the product (3.13). Suppose we have half the terms of

$$f(z) = \prod_{i=j+1}^{k} \Psi_{m_i}(z^{e_i}),$$

for some $j \ge 2$ and we want to compute

$$g(z) = f(z) \cdot \Psi_{m_j}(z^{e_j}),$$

towards the aim of obtaining $\Phi_n(z)$. As both $f(z)$ and $g(z)$ have the (anti)palindromic property of lemma 3, when computing $g(z)$ we need to truncate to degree at most $\lfloor D/2 \rfloor$, where D is the lesser of

$$D_g = \prod_{i=j-1}^{k} (m_i - \phi(m_i)) e_i \qquad \text{and} \qquad \phi(n),$$

the former of which is the degree of $g(z)$, the latter being the degree of $\Phi_n(z)$. Thus we apply lemma 3 to generate the higher-degree terms of $f(z)$ up to degree D. Once we have the product (3.13) we then apply the palindromic property again to generate the coefficients up to degree $\phi(n)/2$, provided we do not have them already. We then divide by the subterms $(1 - z^{n/p_j})$ for $1 \le j \le k$, truncating, again, to degree $\phi(n)/2$. We describe this approach in procedure SPS3. We assume the e_i and m_i were precomputed.

For n a product of one or two primes, SPS3 executes the same as in SPS2, and we see no gains. We only begin to see improved performance for n a product of three primes. In practise, we see the biggest improvement in performance when computing $\Phi_n(z)$ with many distinct prime factors. These are the cyclotomic polynomials which are most difficult to compute.

We do not have an intelligible analysis of the asymptotic operation cost of algorithm SPS3. We try to answer, however, for what subterms of $\Phi_n(z)$ do we truncate to lower degree using SPS3 versus SPS2? For the $2^{k-1} - 1$ subterms appearing in $\Psi_{m_k}(z^{e_k})$ we truncate to the same degree as in SPS2. These are exactly the subterms for which SPS2 had gains over SPS. For the k subterms of the form $(1 - z^{n/p})$, we truncate to degree $\phi(n)/2$ in SPS3. Moreover, the degree of the product in (3.13) is, by (3.12),

$$\phi(n) - n + \sum_{p|n} n/p. \tag{3.15}$$

Thus (3.13) potentially has degree greater than that of $\Phi_n(z)$, provided

$$1/p_1 + 1/p_2 + \cdots + 1/p_k > 1. \tag{3.16}$$

So, for some n there may exist additional subterms for which we do not have gains. For $n = p_1 p_2 \cdots p_k$ for which $\Phi_n(z)$ is presently feasible to compute, however, it is seldom the case that (3.16) holds. The smallest odd, squarefree n for which (3.16) holds is $n = 3,234,846,615$, the product of the first nine odd primes. Thus for n a product of $k < 9$ distinct primes we have gains for all the remaining subterms. In any case, we always truncate to a lower degree than in procedure SPS2 when calculating $\Psi_{m_i}(z^{e_i})$ for $k-8 < i < k$. As $\Psi_{m_k}(z^{e_k}) \cdots \Psi_{m_{k-7}}(z^{e_{k-7}})$ comprise $2^{k-1} - 2^{k-8} - 8$, or close to half of the 2^k subterms.

Quantifying these gains is more difficult. Timings suggest, however, that for n with 6 or more factors, computing $\Phi_n(z)$

Procedure SPS3(n) : Second revision of algorithm **SPS**

Algorithm SPS3: Iterative Sparse Power Series

```
Input: $n = p_1p_2 \dots p_k$, a squarefree product of $k$ primes
Output: $a(0), \dots, a(\frac{\phi(n)}{2})$, the first half of the
        coefficients of $\Phi_n(z)$
1  $a(0), a(1), a(2), \dots, a(\phi(n)/2) \longleftarrow 1, 0, 0, \dots, 0$
2  $D_f \longleftarrow 0$, $D_g \longleftarrow m_k - \phi(m_k)$, $D \longleftarrow \min(D_g, \phi(n))$
3  for $j = k$ down to 2 do
       // $\times$ by $\Psi_{m_j}(z^{e_j})$; truncate to degree $\lfloor D/2 \rfloor$
4      for $d|m_j$ such that $d < m_j$ do
5          if $\mu(\frac{n}{d}) = -1$ then
6              for $i = D$ down to $d$ by $-1$ do
7                  $a(i) \longleftarrow a(i) - a(i-d)$
8          else
9              for $i = d$ to $D$ do
10                 $a(i) \longleftarrow a(i) + a(i-d)$
11     $D_f \longleftarrow D_g$
12     if $j > 2$ then
13         $D_g \longleftarrow D_g + (m_{j+1} - \phi(m_{j+1}))e_{j+1}$
14         $D \longleftarrow \min(D_g, \phi(n))$
15     else $D \longleftarrow \phi(n)$
       // Use lemma 3 to get higher-degree terms
16     for $i \longleftarrow \lfloor D_f/2 \rfloor + 1$ to $\lfloor D/2 \rfloor$ do
17         $a(i) \longleftarrow (-1)^{D_f} a(D_f - i)$

   // $\div$ by $(1 - z^{n/p_j})$; truncate to degree $\phi(n)/2$
18 for $j = 1$ to $k$ do
19     for $i = n/p_j$ to $\phi(n)/2$ do
20         $a(i) \longleftarrow a(i) + a(i - n/p_j)$
   return $a(0), a(1), \dots, a(\phi(n)/2)$
```

using SPS3 is between 3 and 5 times faster than SPS2 (see section 4). The speed-up is typically larger for n with more prime factors.

3.4 Calculating $\Phi_n(z)$ and $\Psi_n(z)$ recursively.

Algorithm SPS3 depended on the identity (3.12), which describes $\Phi_n(z)$ in terms of a product of inverse cyclotomic polynomials of decreasing order and index. We derive a similar expression for $\Psi_n(z)$. Let m_i and e_i be as defined in section 3.3, and again let $n = p_1p_2 \cdots p_k$ be a product of k distinct odd primes where $p_1 < p_2 < \dots p_k$. Again by repeated application of lemma 1,

$$\begin{aligned}\Psi_n(z) &= \Phi_{m_k}(z^{e_k})\Psi_{m_k}(z^{e_{k-1}}),\\ &= \Phi_{m_k}(z^{e_k})\Phi_{m_{k-1}}(z^{e_{k-1}})\Psi_{m_{k-1}}(z^{e_{k-2}}),\\ &\dots\\ &= \Phi_{m_k}(z^{e_k})\cdots\Phi_{m_1}(z^{e_1})\Psi_{m_1}(z^{e_1}).\end{aligned} \tag{3.17}$$

As $m_1 = 1$ and $\Psi_1(z) = 1$, we thus have that

$$\Psi_n(z) = \prod_{j=1}^{k} \Phi_{m_j}(z^{e_j}). \tag{3.18}$$

(3.12) and (3.18) suggest a recursive method of computing $\Phi_n(z)$. Consider the example of $\Phi_n(z)$, for $n = 1155 = 3 \cdot 5 \cdot 7 \cdot 11$. To obtain the coefficients of $\Phi_{1105}(z)$, procedure SPS3 constructs the product

$$\begin{aligned}&\Psi_{105}(z)\Psi_{15}(z^{11})\Psi_3(z^{77})(1 - z^{385})^{-1}\cdot\\ &\cdot (1 - z^{231})^{-1}(1 - z^{165})^{-1}(1 - z^{105})^{-1}(1 - z^{1155})\end{aligned} \tag{3.19}$$

from left to right. However, in light of (3.18), we know this method computes $\Psi_{105}(z)$ in a wasteful manner. We can treat $\Psi_{105}(z)$ as a product of cyclotomic polynomials of smaller index:

$$\Psi_{105}(z) = \Phi_{15}(z)\Phi_5(z^7)\Phi_1(z^{35}).$$

One could apply (3.12) yet again, now to $\Phi_{15}(z)$, giving us

$$\Phi_{15}(z) = \Psi_5(z)(1 - z^5)^{-1}(1 - z^3)^{-1}(1 - z^{15}).$$

Upon computing $\Psi_{105}(z)$, we can break the next term of (3.19), $\Psi_{15}(z^{11})$ into smaller products in a similar fashion. We effectively compute $\Phi_n(z)$ by recursion into the factors of n. We call this approach the recursive sparse power series method, and we describe our implemetation in procedure SPS4.

SPS4 effectively takes a product of cyclotomic polynomials $f(z)$, and multiplies by either $\Phi_m(z^e)$ (or $\Psi_m(z^e)$), by recursion described above. If we are to multiply by $\Psi_m(z^e)$, upon completion of our last recursive call, we are finished (line 8 of SPS4). This is because $\Psi_m(z^e)$ is exactly a product of cyclotomic polynomials by (3.18). If, however, we are to multiply by $\Phi_m(z^e)$, once we have completed our last recursive call, we need to divide and multiply by some additional subterms (lines 10 and 13), as is necessary by the identity (3.13).

Obtaining the degree bound in the recursive SPS method is not as immediate as in the previous SPS algorithms. In the iterative SPS our algorithm produces a sequence of intermediate polynomials. With the possible exception that the output polynomial $\Phi_n(z)$, these polynomials are in order of increasing degree. In the recursive SPS, however, we no longer have this monotonic property.

The difference between the degree bound in the iterative SPS and the recursive SPS, is that in the former we truncate to the least degree of two polynomials, whereas in the recursive sparse power series case, we may bound by the least degree of many polynomials. Moreover, we need to know what degree to bound to at each level of recursion. Procedure SPS4 has an additional parameter, D, which serves as a bound on the degree.

As before, let $f(z)$ be a product of cyclotomic polynomials. Let D_f be the degree of $f(z)$ and suppose, while we are in some intermediate step of the computation of $\Phi_n(z)$ or $\Psi_n(z)$, that we have the first $\lfloor D_f/2 \rfloor + 1$ terms of $f(z)$, and we want next to compute the terms of

$$g(z) = f(z) \cdot \Phi_m(z^e) \quad (\text{or } f(z) \cdot \Phi_m(z^e)), \tag{3.20}$$

up to degree $\lfloor D/2 \rfloor$, for some $D \in \mathbb{N}$. D is effectively the degree of some product of cyclotomic polynomials we will eventually obtain later at some previous level of recursion. If we let D_g be the degree of $g(z)$, then when computing $g(z)$ from $f(z)$ we need only compute terms up to $\lfloor D^*/2 \rfloor$, where $D^* = \min(D, D_g)$ (line 3). Thus when we recurse in SPS4, if $D_g < D$ we lower the degree bound from D to D_g.

To guarantee that we can obtain higher-degree terms whenever necessary we impose the following rule: If SPS4 is given $f(z)$ and is to output $g(z)$, we require that $f(z)$ is truncated to degree $\lfloor D'/2 \rfloor$ on input, where $D' = \min(D, D_f)$, and

Procedure SPS4(m, e, λ, D_f, D, a) : Multiply a product of cylotomic polynomials by $\Phi_m(z^e)$ or $\Psi_m(z^e)$

SPS4: A Recursive Sparse Power Series Algorithm.
Input:

- m, a positive, squarefree odd integer; λ, a boolean; $D \in \mathbb{Z}$, a bound on the degree
- D_f, the degree of $f(z)$, a product of cyclotomic polynomials partially stored in array a. D_f is passed by value.
- An array of integers $a = [a(0), a(1), \ldots]$, for which, given $f(z)$, $a(0)$,$a(1)$, $\ldots$, $a(\lfloor D'/2 \rfloor)$ are the first $\lfloor D'/2 \rfloor + 1$ coefficients of f, where $D' = min(D_f, D)$. a is passed by reference.

Result:
If λ is true, we compute $g(z) = f(z)\Phi_m(z^e)$, otherwise, we compute $g(z) = f(z)\Psi_m(z^e)$. In either case we truncate the result to degree $\lfloor D^*/2 \rfloor$, where $D^* = min(D, D_g)$. We write the coefficients of g to array a, and return the degree of g, D_g.

```
if $\lambda$ then $D_f \longleftarrow D + \phi(m)e$
else $D_f \longleftarrow D + (m - \phi(m))e$

$D^* \longleftarrow \min(D_g, D)$   // $D^*$ is our new degree bound
$e^* \longleftarrow e$, $m^* \longleftarrow m$, $D^* \longleftarrow D$

while $m^* > 1$ do
    $p \longleftarrow$ (largest prime divisor of $m^*$), $m^* \longleftarrow m/p$
    $D_f \longleftarrow$ SPS4($m^*$, $e^*$, not $\lambda$, $D_f$, $D^*$, $a$), $e^* \longleftarrow e^* p$
if not $\lambda$ then      // We have multiplied by $\Psi_m(z^e)$
    return $D_g$

// Get higher degree terms as needed
for $\lfloor D_f/2 \rfloor + 1$ to $\lfloor D^*/2 \rfloor$ do $a(i) \longleftarrow (-1)^{D_f} a(D_f - i)$

// Divide by $(1 - z^{me/p})$ for $p|m$
for each prime $p|m$ do
    for $i = (me/p)$ to $\lfloor D^*/2 \rfloor$ do
        $a(i) \longleftarrow a(i) + a(i - me/p)$

// multiply by $1 - z^{me}$
for $i = \lfloor D^*/2 \rfloor$ down to $d$ do
    $a(i) \longleftarrow a(i) - a(i - me)$
return $D_g$
```

that $g(z)$ is truncated to degree $\lfloor D^*/2 \rfloor$ on output. Note that the degree bound on $g(z)$ is always at least the bound on $f(z)$; it will only increase over the computation of $\Phi_n(z)$.

To calculate the first half of the coefficients of $\Phi_n(z)$, one would merely set

$$(a(0), a(1), a(2), \ldots, a(\phi(n)/2) = (1, 0, 0, \ldots, 0)$$

and call SPS4(n,1,true,0,$\phi(n)$, a)). Similarly, to calculate the first half of $\Psi_n(z)$ we would call SPS4(n,1,false,0,$n - \phi(n)$,a).

3.4.1 Implementing the recursive SPS algorithm

In procedure SPS4 we often need the prime divisors of input m. It is obviously wasteful to factor m every time we recurse. To compute $\Phi_n(z)$ or $\Psi_n(z)$ for squarefree n, we first precompute the factorization of n and store it in a global array $P = [p_1, p_2, \ldots, p_k]$. Upon calling SPS4, every subsequent recursive call will multiply by some (inverse) cyclotomic polynomial of index $m|n$. Our implementation of the recursive sparse power series algorithm has an additional argument, $B = [b_1, b_2, \ldots, b_k]$, a series of bits, that, given P, gives us the factorization of m. We set b_i to 1 if p_i divides m, and zero otherwise. For all tractable cases, B can fit in two bytes and in most practical cases, one byte.

Thus, in the while loop on line 5 in SPS4, we take a copy of B, call it B^*, and scan it for nonzero bits. Each time we find a nonzero bit we set that bit to zero, and pass B^* by value to the recursive call occuring on line 7 of procedure SPS4. We continue in this fashion until all the bits of B^* are set to zero. We similarly scan the bits of B again to later obtain the prime divisors of n, as is needed on line 10 of procedure SPS4.

We find that the recursive SPS is slightly faster than the iterative SPS; however, this improvement is not nearly as substantial as was the iterative SPS over prior versions. While the degree bound computing $\Phi_n(z)$ with the recursive SPS is always less than or equal to that using the iterative SPS, the recursive structure of the program results in additional overhead. We could program the recursive SPS iteratively; however, we would effectively have to create our own stack to mimic recursion.

4. PERFORMANCE AND TIMINGS

We first provide a visual comparison of the SPS algorithms computing explicit examples of $\Phi_n(z)$. Figures 1 and 2 show how the degree bound grows in algorithms SPS1-4 over the computation of $\Phi_n(z)$ for

$$n = 43730115 = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \text{ and}$$
$$n = 3234846615 = 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29.$$

respectively. In both figures, the horizontal axis represents how many subterms we have in our intermediate product. As $n = 43730115$ has 7 unique prime divisors, there are some $2^7 - 1 = 127$ intermediate products of subterms produced over the computation, excluding the final result $\Phi_n(z)$. As the computation of $\Phi_n(z)$ progresses we traverse from left to right in figures 1 and 2, and the degree bound increases.

We should note that in SPS2-4, the degree bound in each is at most the degree bound of its predecessor. In figure 1, we associate the darkest green region with SPS4; the two darkest green regions with SPS3; the three darkest green regions with SPS2; and all four green areas represent the degree bound for SPS. The height of the regions associated with a version of the SPS algorithm represents its degree bound at that stage of the computation. Figure 2, in red, should be interpreted similarly. In the case of SPS the degree bound is always the constant $\phi(n)/2$.

We think of the area of the regions in figure 1 associated with a version of the SPS algorithm as a heuristic measure of its time cost. One could think of the savings of SPS4 over SPS3, for instance, as the area of the second darkest green region. The area of the three darker green regions is slightly over half the area of all four. As such, we expect that SPS2 would take roughly half as much time as SPS. Moreover, by this measure we expect that SPS3 should be considerably faster than SPS2, and SPS4 should be marginally faster than SPS3. This is comparable with our timings in table 1. The

Figure 1: Growth of the degree bound over the computation of $\Phi_{43730115}(z)$ using SPS1-4

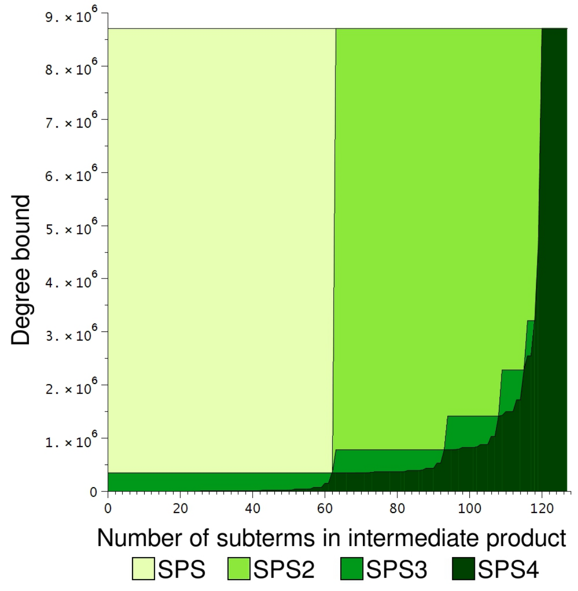
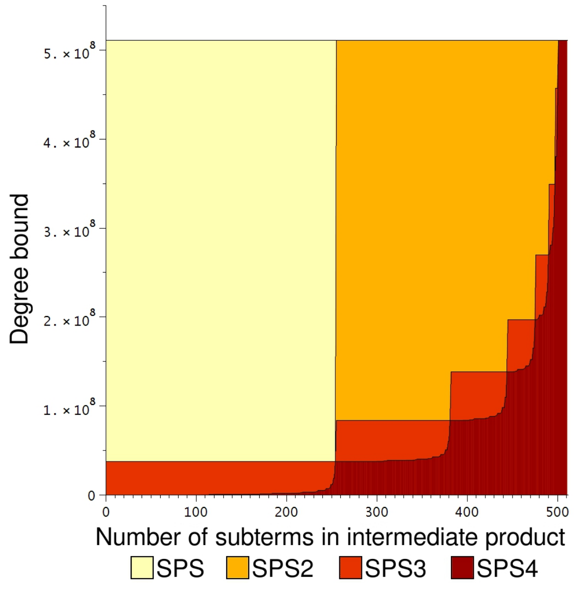


Figure 2: Growth of the degree bound over the computation of $\Phi_{3234846615}(z)$ using SPS1-4



degree bounds in figure 2 show a similar, albeit more clearly defined shape.

We timed our implementations on a system with a 2.67GHz Intel Core i7 quad-core processor and 6 GB of memory. All of our aglorithms are implemented in C and are single-threaded. Here we time our 64-bit precision implementations of procedures SPS1-4, each of which check for integer overflow using inline assembly. Our implementation of algorithm 1 calculates $\Phi_n(z)$ modulo two 32-bit primes and reconstructs $\Phi_n(z)$ by Chinese remaindering.

Table 1: Time to calculate $\Phi_n(z)$ (in seconds*)

	algorithm				
n	FFT	SPS	SPS2	SPS3	SPS4
255255	0.40	0.00	0.00	0.00	0.00
1181895	1.76	0.01	0.00	0.00	0.00
4849845	7.74	0.12	0.06	0.02	0.01
37182145	142.37	1.75	0.95	0.23	0.19
43730115	140.62	1.69	0.93	0.23	0.19
111546435	295.19	6.94	3.88	1.45	0.94
1078282205	-	105.61	58.25	12.34	9.29
3234846615	-	432.28	244.44	81.32	49.18

*times are rounded to the nearest hundredth of a second

As the number of distinct prime factors of n plays a significant role in the cost of computing $\Phi_n(z)$, we list the factors of n (table 2) and $A(n)$ (table 3) for n appearing in table 1.

For the SPS and SPS4 algorithms, we have implemented, in addition to the 64-bit version, 8-bit, 32-bit, and 128-bit precision versions. We do not use GMP multiprecision integer arithmetic. It was easy to implement multiprecision arithmetic for our specific purpose as we only add and subtract coefficients in the SPS algorithms. We also have a version of SPS and SPS4 which calculates images of $\Phi_n(z)$ modulo 32-bit primes, writes the images to the harddisk, and then reconstruct $\Phi_n(z)$ from the images by way of Chinese remaindering. This implementation is most useful for $\Phi_n(z)$ which we cannot otherwise fit in main memory.

Table 2: Factorization of n, for n from table 1

n	factorization of n
255255	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17$
1181895	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29$
4849845	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$
37182145	$5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$
43730115	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37$
111546435	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$
1078282205	$5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
3234846615	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$

Table 3: $A(n)$ for n from table 1

n	height $A(n)$
255255	532
1181895	14102773
4849845	669606
37182145	2286541988726
43730115	862550638890874931
111546435	1558645698271916
1078282205	8161018310
3234846615	2888582082500892851

5. CURRENT WORK

We have implemented the algorithms in this paper to create a library of data on the heights and lengths of cyclotomic polynomials. This data is available at

`http://www.cecm.sfu.ca/~ada26/cyclotomic/`

A 64-bit implementation of the SPS4 algorithm, written in C but without overflow check, is also made available at the website.

We aim to compute the coefficients of $\Phi_n(z)$, for

$$n = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 43 \cdot 53 = 99660932085.$$

We expect that this cyclotomic polynomial will have very large height. We have previously verified that

$$A(\tfrac{n}{53}) = 645409970360109115668264461815238889715 63 \text{ and}$$
$$A(\tfrac{n}{43}) = 67075962666923019823602030663153118803367$$

are the smallest two examples of k such that $A(k) > k^4$. Both $A(\frac{n}{53})$ and $A(\frac{n}{43})$ are greater than 2^{135}.

We previously attempted to compute $\Phi_n(z)$ using an implementation of the SPS algorithm. We computed images of $\Phi_n(z)$ modulo 32-bit primes. Storing half of $\Phi_n(z)$ to 32-bit precision takes roughly 76 GB of space. We do not have enough RAM to store these images in main memory, so we read and wrote intermediate results to the hard disk. This proved to be slow, as each image required us to make $2^9 = 512$ passes over the hard disk. We computed four images of $\Phi_n(z)$, after which the hard disk crashed.

In light of the development of the new variants of the SPS algorithms, we have a new approach to compute $\Phi_n(z)$. We want to minimize hard disk reads and writes. This is because performing the computation on the harddisk is appreciably slower and potentially more error-prone than in memory. We are limited to 16 GB of RAM. We expect that $A(n) < 2^{320}$; that is, 320-bit precision will be sufficient to construct $\Phi_n(z)$. Towards our aim, let

$$f(z) = \Psi_{m_9}(z)\Psi_{m_8}(z^{53}) \tag{5.1}$$

where $m_9 = \frac{n}{53} = 1,880,394,945$ and
$m_8 = \frac{n}{43\cdot 53} = 43,730,115$. By (3.11), we have

$$\Phi_n(z) = f(z)(1 - z^{n/53})^{-1}(1 - z^{n/43})^{-1}\Phi_{m_8}(z^{43\cdot 53}). \tag{5.2}$$

$f(z)$ has degree less than $2.55 \cdot 10^9$. We can compute images of $f(z)$ modulo 64-bit primes using roughly 10 GB of RAM, then extract $f(z)$ from its images by way of Chinese remaindering. After which we will compute the coefficients of the truncated power series

$$\begin{aligned} g(z) &= \sum_{i=0}^{\phi(n)/2} c(i)z^i, \\ &= f(z)(1 - z^{n/53})^{-1}(1 - z^{n/43})^{-1} \bmod z^{\phi(n)/2+1}. \end{aligned} \tag{5.3}$$

This will entail two passes over the hard disk, one per division by $1 - z^{n/53}$ or $1 - z^{n/43}$. We produce the coefficients of $g(z)$ in order of ascending degree during the second pass of the harddisk. Storing $g(z)$ or $\Phi_n(z)$ at this precision up to degree $\phi(n)/2$ requires more than 750 GB of storage. We can reorganize the terms of $g(z)$ in a manner which allows us to compute the coefficients of $\Phi_n(z)$ in memory. For $0 \leq j < 43 \cdot 53 = 2279$, let

$$g_j(z) = \sum_{0 \leq i\cdot 2279 + j \leq \phi(n)/2} c(i)z^i \tag{5.4}$$

We can construct the $g_j(z)$ as we sequentially produce the terms of $g(z)$. We have that

$$g(z) = \sum_{j=0}^{2278} z^j \cdot g_j(z^{2279}),$$

and thus by (5.2),

$$\Phi_n(z) \equiv \sum_{j=0}^{2278} z^j \cdot g_j(z^{2279})\Phi_{m_8}(z^{2279}) \pmod{z^{\phi(n)/2+1}}.$$

Thus to produce the first half of the coefficients of $\Phi_n(z)$, it suffices to compute $g_j(z) \cdot \Phi_{m_8}(z)$, for $0 \leq j < 2279$. Each polynomial has degree less than $2.6 \cdot 10^6$, and can be computed to 320-bit precision with less than a GB of memory.

6. REFERENCES

[1] A. Arnold and M. Monagan. Calculating cyclotomic polynomials. Submitted to *Mathematics of Computation*, available at `http://www.cecm.sfu.ca/~ada26/cyclotomic/`.

[2] P. Erdős. On the coefficients of the cyclotomic polynomial. *Bull. Amer. Math. Soc.*, 52:179–184, 1946.

[3] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Boston, 1992.

[4] Y. Koshiba. On the calculations of the coefficients of the cyclotomic polynomials. *Rep. Fac. Sci. Kagoshima Univ.*, (31):31–44, 1998.

[5] Y. Koshiba. On the calculations of the coefficients of the cyclotomic polynomials. II. *Rep. Fac. Sci. Kagoshima Univ.*, (33):55–59, 2000.

[6] T.D. Noe. Personal communication.

Accuracy Versus Time: A Case Study with Summation Algorithms[*]

Philippe Langlois
Equipe de Recherche DALI
Université de Perpignan
52 Avenue Paul Alduy
66860 Perpignan, France
langlois@univ-perp.fr

Matthieu Martel
Equipe de Recherche DALI
Université de Perpignan
52 Avenue Paul Alduy
66860 Perpignan, France
matthieu.martel@univ-perp.fr

Laurent Thévenoux
Equipe de Recherche DALI
Université de Perpignan
52 Avenue Paul Alduy
66860 Perpignan, France
laurent.thevenoux@univ-perp.fr

ABSTRACT

In this article, we focus on numerical algorithms for which, in practice, parallelism and accuracy do not cohabit well. In order to increase parallelism, expressions are reparsed, implicitly using mathematical laws like associativity, and this reduces the accuracy. Our approach consists in focusing on summation algorithms and in performing an exhaustive study: we generate all the algorithms equivalent to the original one and compatible with our relaxed time constraint. Next we compute the worst errors which may arise during their evaluation, for several relevant sets of data. Our main conclusion is that relaxing very slightly the time constraints by choosing algorithms whose critical paths are a bit longer than the optimal makes it possible to strongly optimize the accuracy.

We extend these results to the case of bounded parallelism and to accurate sum algorithms that use compensation techniques.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Optimization; G.1.0 [**Mathematics of Computing**]: Numerical Analysis, Computer Arithmetic; I.2.2 [**Automatic Programming**]: Program transformation.

General Terms

Algorithms, Design, Experimentation, Performance, Reliability.

Keywords

Parallelism, Summation, Floating-point numbers, Precision.

[*]This work was partly supported by the project "Compil'HD" of Région Languedoc-Roussillon ("Chercheur d'Avenir" programme) and by the ANR project "EvaFlo".

1. INTRODUCTION

Symbolic-numeric algorithms have to manage the *a priori* conflicting numerical accuracy and computing time. Performances and accuracy of basic numerical algorithms for scientific computing have been widely studied, as for example the central problem of summing floating point values – see the numerous references in [5] or more recently in [20, 14, 19]. Instruction level parallelism is commonly used to speed-up these implementations during compilation steps. One could also expect that compilers improve accuracy

However as already noticed by J. Demmel [2], in practice parallelism and accuracy do not cohabit well. To exploit the parallelism within an expression, this one is reparsed implicitly using mathematical laws like associativity. The new expression is then more balanced to benefit for as much parallelism as possible. In our scope, such re-writing should yield algorithms that sum n numbers in a logarithmic time $O(\log n)$. The point is that the numerical accuracy of some algorithms is strongly sensitive to reparsing. In IEEE754 floating-point arithmetic, additions are not associative and, in general, most algebraic laws like associativity and distributivity do not hold any longer. As a consequence, while increasing the parallelism of some expression, its numerical accuracy may decrease and, conversely, improving the accuracy of some computation may reduce its parallelism. Moreover, in architectures, instruction level parallelism is bounded and it may possible to execute an algorithm less parallel than the optimum in the same (or very similar) execution time.

In this article, we address the following question: *How can we improve the accuracy of numerical summation algorithms if we relax slightly the performance constraints?* More precisely, we examine how accurate can be algorithms which are k times less efficient than the optimal one or with a constant overhead with respect to the optimal one, e.g. for the summation of n values, in $k \times \lfloor \log(n) \rfloor$ or $k + \lfloor \log(n) \rfloor$ for a constant parameter k.

For example, let us consider the sum

$$s = \sum_{i=1}^{N} a_i, \text{ with } a_i = \frac{1}{2^i},\ 1 \le i \le N \tag{1}$$

Two extreme algorithms compute s

$$s_1 := \big(((a_1 + a_2) + a_3) + \ldots a_{N-1} \big) + a_N \tag{2}$$

and, assuming $N = 2^k$,

$$s_2 := \Big(\big((a_1+a_2)+(a_3+a_4)\big)+\ldots+(a_{\frac{N}{2}-1}+a_{\frac{N}{2}})\Big)+$$
$$\Big(\big((a_{\frac{N}{2}+1}+a_{\frac{N}{2}+2})+(a_{\frac{N}{2}+3}+a_{\frac{N}{2}+4})\big)+\ldots+(a_{N-1}+a_N)\Big) \quad (3)$$

Clearly, the sum s_1 is computed sequentially while s_2 corresponds to a reduction which can be computed in logarithmic time. However, in double precision, we have, for $N = 10$:

$$s = 0.9990234375 \quad s_1 = 0.9990234375 \quad s_2 = 0.99609375$$

and it happens that s_1 is far more precise than s_2.

Our approach consists in performing an exhaustive study. First we generate all the algorithms equivalent to the original one and compatible with our relaxed time constraint. Then we compute the worst errors which may arise during their evaluation for several relevant sets of data. Our main conclusion is that relaxing very slightly the time constraints by choosing algorithms whose critical paths are a bit longer than the optimal one makes it possible to strongly optimize the accuracy. This matter of fact is illustrated using various datasets, most of them being ill-conditioned. We extend these results to the case of bounded parallelism and to compensated algorithms. For bounded parallelism we show that more accurate algorithms whose critical path is not optimal can be executed in as many cycles as optimal algorithms, e.g. on VLIW [7] architectures. Concerning compensation, we show that elaborated summation algorithms can be discovered automatically by inserting systematically compensations and then reparsing the resulting expression.

This article is organized as follows. Section 2 gives an overview of summation algorithms. It also introduce our technique to bound error terms. Section 3 presents our main results concerning the time versus precision compromise. Section 4 describes how we generate exhaustively the summation algorithms of interest and Section 5 introduces further examples involving larger sums, accuracy versus bounded parallelism and compensated sums. Finally, some perspectives and concluding remarks are given in Section 6.

2. BACKGROUND

In floating-point arithmetic accuracy is a critical matter, for scientific computing as well as for critical embedded systems [11, 12, 3, 7, 6, 4]. Famous examples alas illustrate that bad accuracy can cause human damages [13] and money loses [10]. If accuracy is critical so is parallelism but usually these two domains are considered separately. While focusing on summation, this section compares the most well-known algorithms with respect to their accuracy and parallelism characteristics.

In Subsection 2.1 we recall background material on summation algorithms [5, 14] and we explain how we measure the error terms in Subsection 2.2.

2.1 Summation Algorithms

Summation in floating-point arithmetic is a very rich research domain. There are various algorithms that improve accuracy of a sum of two or more terms and similarly, there are many parallel summation algorithms.

2.1.1 Two Extreme Algorithms for Parallelism

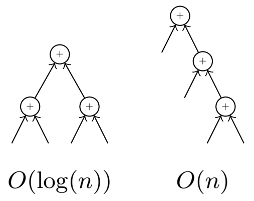


Figure 1: Graphical representation of Algorithm 1 and Algorithm 2 by dataflow graphs.

Basically, there are two extreme algorithms with respect to parallelism properties to compute the sum of $(n+1)$ terms. The first following algorithm is fully sequential whereas the second one benefits from the maximum degree of parallelism.

- **Algorithm 1** is the extreme sequential algorithm. It computes a sum in $O(n)$ operations successively summing the $n+1$ floating-point numbers (see Equation (2)).
- Pairwise summation **Algorithm 2** is the most parallel algorithm. It computes a sum in $O(\log(n))$ successive stages (see Equation (3)).

These algorithms are represented by dataflow graphs in Figure 1.

Algorithm 1 Sum: Summation of $n+1$ Floating-Point Numbers

```
Input: $p$ is (a vector of) $n+1$ floating-point numbers
Output: $s_n$ is the sum of $p$
  $s_0 \leftarrow p_0$
  for $i = 1$ to $n$ do
    $s_i \leftarrow s_{i-1} \oplus p_i$
  end for
```

Algorithm 2 SumPara: Parallel Summation of $n+1$ Floating-Point Numbers

```
Input: $p[l:r]$ is (a vector of) $n+1$ floating-point numbers
Output: the sum of $p$
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $l = r$ then
    return $p_l$
  else
    return SumPara($p[l:m]$) $\oplus$ SumPara($p[m+1:r]$)
  end if
```

Mixing Algorithm 1 and Algorithm 2 yields many algorithms of parallelism degrees between those two extreme ones.

2.1.2 Merging Parallelism and Accuracy

It is well known that these two extreme algorithms do not verify the same worst case error bounds [5]. Nevertheless to improve the accuracy of one computed sum, it is usual to sort the terms according to some of their characteristics (increasingly, decreasingly, negative or positive sort, etc.).

Summation accuracy varies with the order of the inputs. Increase or decrease orders of the absolute values of the operands are the first two choices for the simplest Algorithm

1. If the inputs are both negative and positive, the decrease order is better, otherwise other orders are equivalent. If all the inputs are of the same sign, the increase order is more interesting than others [5]. More dynamic inserting methods consist in sorting the inputs (in a given order), in summing the first two numbers and in inserting the result within the inputs conserving the initial order. Such sorting is more difficult to implement while conserving the parallelism level of Algorithm 2.

2.1.3 *More Accuracy with Compensation*

A well known and efficient techniques to improve accuracy is compensation which uses some of the following error-free transformations [14].

Algorithm 3 computes the sum of two floating-point number $x = a \oplus b$ and the absolute error y due to the IEEE754 arithmetic [1].

Algorithm 3 TwoSum, Result and Absolute Error in Summation of Two Floating-Point Numbers (Introduced by Knuth [8])

```
Input: $a$ and $b$, two floating-point numbers
Output: $x = a \oplus b$ and $y$ the absolute error on $x$
  $x \leftarrow a \oplus b$
  $z \leftarrow x \ominus a$
  $y \leftarrow (a \ominus (x \ominus z)) \oplus (b \ominus z)$
```

When $|a| \geq |b|$ Algorithm 4 is faster than Algorithm 3. Obviously it will be necessary to check this condition to apply it. The overcost of such practice on modern computing environments is not so clear [19, 9]. In both cases the key point is the error-free transformation $x + y = a + b$.

Algorithm 4 FastTwoSum, Result and Absolute Error in Summation of Two Floating-Point Numbers

```
Input: $a$ and $b$ two floating-point numbers such that $|a| \geq$
  $|b|$
Output: $x = a \oplus b$ and $y$ the absolute error on $x$
  $x \leftarrow a \oplus b$
  $y \leftarrow (a \ominus x) \oplus b$
```

To improve the accuracy of Algorithm 1, VecSum Algorithm applies this error-free transformation. Algorithm 6 uses this error-free vector transformation and yields a twice more accurate summation algorithm [14]. Hence Sum2 computes every rounding error y and adds them together before compensating the classic Sum computed result. In other words, Sum Algorithm applies twice, once to the $n+1$ summand and then once to the n error terms, the compensated summation being the last addition between these two values.

Algorithm 5 VecSum, Error-Free Vector Transformation of $n+1$ Floating-Point Numbers [14]

```
Input: $p$ is (a vector of) $n+1$ floating-point numbers
Output: $p_n$ is the approximate sum of $p$, $p[0 : n-1]$ is (a
  vector of) the generated errors
  for $i = 1$ to $n$ do
    $[p_i, p_{i-1}] \leftarrow TwoSum(p_i, p_{i-1})$
  end for
```

Algorithm 6 Sum2, Compensated Summation of $n+1$ Floating-Point Numbers

```
Input: $p$ is (a vector of) $n+1$ floating-point numbers
Output: $s$ the sum of $p$
  $p \leftarrow VecSum(p)$
  $e \leftarrow Sum(i = 0, n-1, p[i])$
  $s \leftarrow p_n \oplus e$
```

These error-free transformations have been used differently within several other accurate summation algorithm. Previous Sum2 was also considered by [15]. A slight variation is the famous Kahan compensated summation: in Algorithm 7, every rounding error e is added to the next summand (the compensating step) before adding it to the previous partial sum.

It exists many other algorithms for accurate summation that use these error-free transformations, as for example Priest double-compensated summation [16] or the recursive SumK algorithms of [14] or also the very fast AccSum and PrecSum of [19]. We do not detail these any longer.

Algorithm 7 SumComp, Compensated Summation of n Floating-Point Numbers (Kahan [5])

```
Input: $p$ is (a vector of) $n+1$ floating-point numbers
Output: $s$ the sum of input numbers
  $s \leftarrow p_0$
  $s \leftarrow 0$
  for $i = 1$ to $n$ do
    $tmp \leftarrow s$
    $y \leftarrow p_i \oplus e$
    $s \leftarrow tmp \oplus y$
    $e \leftarrow (tmp \ominus s) \oplus y$
  end for
```

2.2 Measuring the Error Terms

Let x and y be two real numbers approximated by floating-point numbers $\hat{x}$ and $\hat{y}$ such that $x = \hat{x} + \epsilon_x$ and $y = \hat{y} + \epsilon_y$ for some error terms $\epsilon_x \in \mathbb{R}$ and $\epsilon_y \in \mathbb{R}$. Let us consider the sum $S = x + y$. In floating-point arithmetic this sum is approximated by

$$\hat{S} = \hat{x} \oplus \hat{y}$$

where $\oplus$ denotes the floating-point addition. We write the difference ϵ_S between S and $\hat{S}$ as in [21],

$$\epsilon_S = S - \hat{S} = \epsilon_x + \epsilon_y + \epsilon_+, \tag{4}$$

where ϵ_+ denotes the round-off error introduced by the operation $\hat{x} \oplus \hat{y}$ itself.

In the rest of this article, we use intervals $\mathbf{x}$, $\mathbf{y}, \dots$ instead of floating-point numbers $\hat{x}$, $\hat{y}, \dots$ as well as for the error terms ϵ_x, $\epsilon_y, \dots$ for the next two different reasons.

(i) Our long-term objective is to perform program transformations at compile-time [12] to improve the numerical accuracy of mathematical expressions. It comes out that our transformations have to improve the accuracy of any dataset or, at least, of a wide range of datasets. So we consider inputs belonging to intervals.

(ii) The error terms are real numbers, not necessarily representable by floating-point numbers as ϵ_S in Equation

(4). We approximate them by intervals, using rounding modes towards outside. Clearly, the towards outside rouding mode correspond, in this case, at the rounding mode towards $-\infty$ for the lower bound of the interval and towards $+\infty$ for the upper bound.

An interval $\mathbf{x}$ with related interval error $\epsilon_{\mathbf{x}}$ denotes all the floating-point numbers $\hat{x} \in \mathbf{x}$ with a related error $\epsilon_x \in \epsilon_{\mathbf{x}}$. This means that the pair $(\mathbf{x}, \epsilon_{\mathbf{x}})$ represents the set X of exact results:

$$X = \{x \in \mathbb{R} \ : \ x = \hat{x} + \epsilon_x, \ \hat{x} \in \mathbf{x}, \ \epsilon_x \in \epsilon_{\mathbf{x}}\}.$$

Let $\mathbf{x}$ and $\mathbf{y}$ be two sets of floating-point numbers with error terms belonging to the intervals $\epsilon_{\mathbf{x}} \subseteq \mathbb{R}$ and $\epsilon_{\mathbf{y}} \subseteq \mathbb{R}$. We have

$$\mathbf{S} = \mathbf{x} \oplus_I \mathbf{y} \tag{5}$$

where $\oplus_I$ is the sum of intervals with the same rounding mode than $\oplus$ (generally to the nearest) and

$$\epsilon_{\mathbf{S}} = \epsilon_{\mathbf{x}} \oplus_O \epsilon_{\mathbf{y}} \oplus_O \epsilon_+ \tag{6}$$

where $\oplus_O$ denotes the sum of intervals with rounding mode towards outside. Per example:

$$\big([\underline{x}, \overline{x}]; [\underline{\epsilon_x}, \overline{\epsilon_x}]\big) + \big([\underline{y}, \overline{y}]; [\underline{\epsilon_y}, \overline{\epsilon_y}]\big) =$$

$$\big([\underline{x} +_{-\infty} \underline{y}, \overline{x} +_{+\infty} \overline{y}]; [\underline{\epsilon_x} +_{-\infty} \underline{\epsilon_y}, \overline{\epsilon_x} +_{+\infty} \overline{\epsilon_y}]\big)$$

In addition, ϵ_+ denotes the round-off error introduced by the operation $\hat{x} \oplus_I \hat{y}$. Let $ulp(x)$ denote the function which computes the unit in the last place of x [5], i.e. the weight of the least significant digit of x and let $S = [\underline{S}, \overline{S}]$. We bound ϵ_+ by the interval $[-u, u]$ by:

$$u = \frac{1}{2} \max(ulp(|\underline{S}|), ulp(|\overline{S}|)).$$

Using the notations of equations (4), (5) and (6), it follows that for all $\hat{x} \in \mathbf{x}$, $\epsilon_x \in \epsilon_{\mathbf{x}}$, $\hat{y} \in \mathbf{y}$, $\epsilon_y \in \epsilon_{\mathbf{y}}$

$$S \in \mathbf{S} \text{ and } \epsilon_S \in \epsilon_{\mathbf{S}}.$$

3. NUMERICAL ACCURACY OF NON-TIME-OPTIMAL ALGORITHMS

The aim of this section is to show how we can improve accuracy while relaxing the time constraints. In Subsection 3.1, we illustrate our approach using as an example a sum of random values. We generalize our results to some significant sets of data in Subsection 3.2.

3.1 The General Approach

In order to evaluate the algorithms to compute one sum expression, associativity and distributivity are only needed hereafter. Basically, while in exact arithmetic all the algorithms are numerically equivalent, in floating-point arithmetic the story is not the same. Indeed, many things may arise like absorption, rounding errors, overflow, etc. and then floating-point algorithms return various different results.

One mathematical expression yields a huge amount of evaluation schemes. We propose to analyse this huge set of algorithms with respect to accuracy and parallelism. First we search the most accurate algorithms among all levels of parallelism, and then we search among them the ones with the best degrees of parallelism. We aim at finding the more interesting ratio between accuracy and parallelism.

In this section, we use random data (generated using an uniform random distribution) defined as interval $[\underline{a}, \overline{a}]$. We measure the interval that represents the maximum error bound $[\overline{e}, \underline{e}]$ applying the previously described error model. Let $\mathbf{a}_i = [\underline{a_i}, \overline{a_i}], 1 \leq i \leq n$. This means that for all $a_1 \in \mathbf{a}_i, \ldots, a_n \in \mathbf{a}_n$, the error on $\Sigma_1^n a_i$ belongs to $[\underline{e}, \overline{e}]$. We focus the maximum error which is defined as $max(|\underline{e}|, |\overline{e}|)$. Algorithms which have the smaller maximum error are called optimal algorithms. This maximum error is a pertinant optimization criteria in the compilation domain. With this criteria we want to guarantee the maximum error which can arise during any execution of a program.

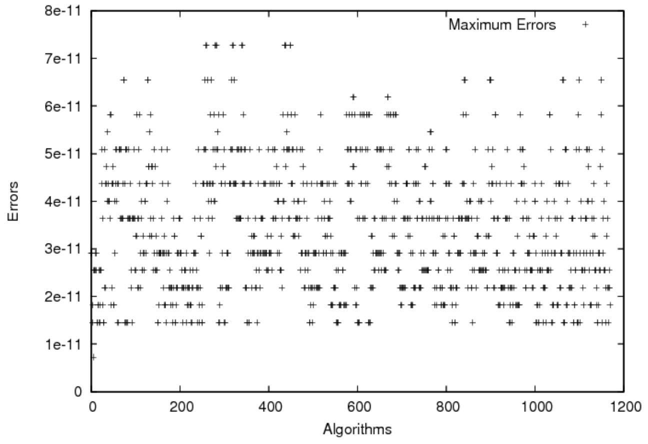


Figure 2: Maximum errors for each algorithms for a six terms summation reparsings.

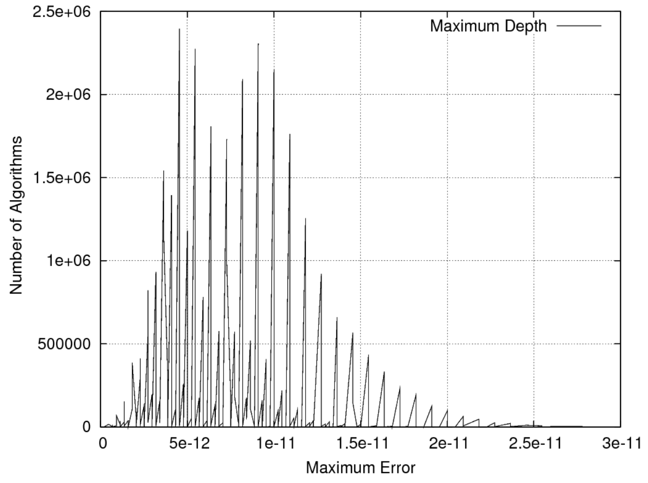


Figure 3: Error repartition when summing ten terms.

Each dot of Figure 2 shows the absolute error of every algorithms, i.e. every parsing of the summing expression with six terms. X-axis represents the algorithms numbered from 0 to 1,170 and Y-axis represents the maximal absolute error which can be encounter during the algorithm evaluation. It is not a surprise that errors are not uniformly distributed and that the errors belong to a small number of stages. Figure 3 shows the distribution of the errors for the different stages of a ten terms summation. The proportion of algorithms with very few small or very large errors is small. Most of the algorithms present an average accuracy between small and large errors. We guess that it will be difficult to find the best accurate algorithms (as well as the worst one), most having an average accuracy.

It exists 46,607,400 different algorithms for an expression of ten terms. Among this huge set, many of them are sequential or almost sequential. So we propose to restrict the search to a certain level of parallelism. Let n be the number of additions and k a constant chosen arbitrarily e.g. here $k = 2$. In the following of this article, if it is not precisely defined, we sum ten terms and k is equal to two. We restrict our search of accurate algorithms within three included sets: algorithms having a computing tree of height smaller or equal to $\lfloor \log(n) \rfloor + 1$, $\lfloor \log(n) \rfloor + k$ and $k \times \lfloor \log(n) \rfloor$. Using these restrictions, there are 27,102,600 algorithms of level $k \times \lfloor \log(n) \rfloor$, 13,041,000 algorithms of level $\lfloor \log(n) \rfloor + k$ and 2,268,000 algorithms of level $\lfloor \log(n) \rfloor + 1$.

Results are given in Figure 4 and in Table 1. We observe that the highest level of parallelism, the level $\lfloor \log(n) \rfloor + 1$, does not allow us to compute the most accurate results. Nevertheless if we use a less high but still reasonable level of parallelism, e.g. levels $O(\lfloor \log(n) \rfloor + k)$ or $O(k \times \lfloor \log(n) \rfloor)$, we can compute accurate results.

The more the level of parallelism is, the harder it is to find the more accurate algorithms among all of them. In tables 2 and 3 we observe that the level $\lfloor \log(n) \rfloor + k$ presents a better proportion of accurate algorithms (stages with small numbers) than the higher parallelism level $k \times \lfloor \log(n) \rfloor$. Moreover the most accurate algorithms within the first set are less accurate than the ones of the second set — see Figure 4.

Parallelism	Error of Optimal Algorithm	Percent
no parallelism	$2.273e^{-13}$	0.006
$\lfloor \log(n) \rfloor + 1$	$4.547e^{-13}$	0.007
$\lfloor \log(n) \rfloor + k$	$2.273e^{-13}$	0.006
$k \times \lfloor \log(n) \rfloor$	$2.273e^{-13}$	0.007

Table 1: Optimal error value and percentage of algorithms reaching this precision.

3.2 Larger Experiments

We study a more representative sets of data using various kinds of values chosen as well-known error-prone problems, i.e. ill-conditioned set of summands. The condition number for computing $s = \sum_{i=1}^{N} x_i$, is defined as following,

$$cond(s) = \frac{\sum_{i=1}^{N} |(x_i)|}{|s|}.$$

The larger this number is, the more ill-conditioned the summations are, the less the result is accurate.

Summation suffers from the two following problems:

- Absorption arises when adding a small and a large values. The smallest values are absorbed by the largest ones. In our context (IEEE-754 double precision): $10^{16} \oplus 10^{-16} = 10^{16}$. In general absorption is not so dangerous while adding values of the same sign: its condition number equals roughly one. Nevertheless a large amount of small errors accumulates in large summations — this was the case in the well known Patriot Missile failure [13].

- Cancellation arises when absorption appears within data with different sign. In this case, the condition number can be arbitrarily large. We will call such case as summation with ill-conditioned data. In our context an example is : $(10^{16} \oplus 10^{-16}) \ominus 10^{16} = 0$.

Stage	Example of expression	%
1	$(i+(f+g))+((c+d)+((h+j)+(e+(a+b))))$	0.006
2	$(i+(f+g))+(j+((c+d)+((e+h)+(a+b))))$	0.024
3	$(i+(f+g))+(j+((e+(a+h))+(b+(c+d))))$	0.001
⋮	⋮	⋮
141	$(j+((c+g)+(b+h)))+(e+(a+(d+(f+i))))$	0.001
142	$(j+(h+(g+(c+e))))+(b+(a+(d+(f+i))))$	0.005
143	$(j+(h+(e+(c+g))))+(b+(a+(d+(f+i))))$	0.002

Table 2: Repartition of the algorithms according to their precision at the parallelism level $O(\lfloor \log(n) \rfloor + k)$ on ten terms summation (stages with small numbers are the smallest errors).

Stage	Example of expression	%
1	$(i+(f+g))+((c+d)+((h+j)+(e+(a+b))))$	0.008
2	$(i+(f+g))+(j+((c+d)+(h+(e+(a+b)))))$	0.039
3	$(i+(f+g))+(j+((e+(a+h))+(b+(c+d))))$	0.004
⋮	⋮	⋮
171	$(j+(g+(b+h)))+(e+(c+(a+(d+(f+i)))))$	0.007
172	$(j+(h+(e+g)))+(c+(b+(a+(d+(f+i)))))$	0.015
173	$(j+(h+(c+g)))+(e+(b+(a+(d+(f+i)))))$	0.001

Table 3: Repartition of the algorithms according to their precision at the parallelism level $O(k \times \lfloor \log(n) \rfloor)$ on ten terms summation (stages with small numbers are the smallest errors).

We introduce 9 datasets to generate different types of absorptions and cancellations. These two problems are clear with scalar values. So we first use intervals with small variations around such scalar values. Every dataset is composed of ten samples that share the same numerical characteristics. We recall that these experiments are limited to ten summands. In the following, we say that a floating-point value is a small, medium or large when it is, respectively, of the order of 10^{-16}, 1 and 10^{16}. This is justified in double

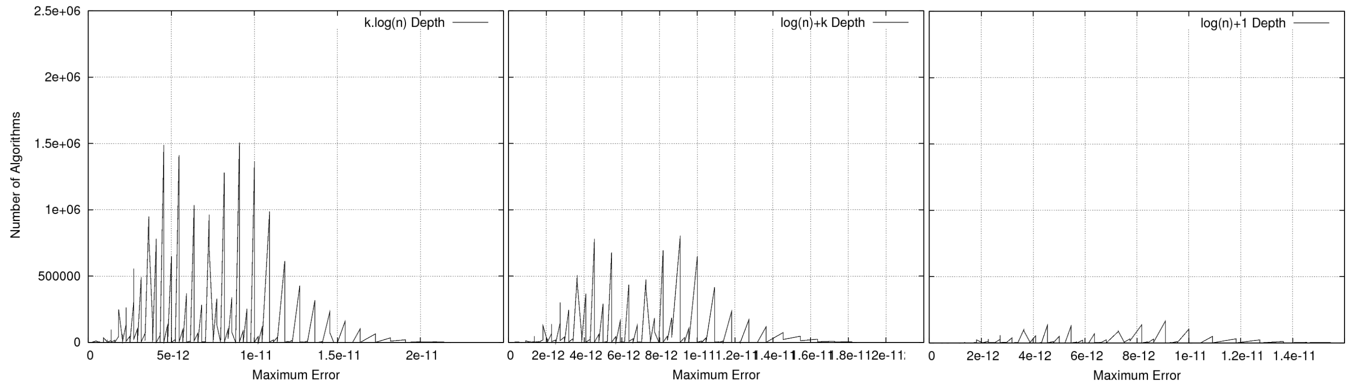


Figure 4: Error repartition with three different degrees of parallelism for ten terms summation.

precision IEEE-754 arithmetic.

- **Dataset 1**. Positive sign, 20% of large values among small values. There are absorptions and accurate algorithms should first sum the smallest terms (increasing order).
- **Dataset 2**. Negative sign, 20% of large values among small values. Results should be the same as in Dataset 1.
- **Dataset 3**. Positive sign, 20% of large values among small and medium values. The best results should be obtained with algorithms which sum in increase order.
- **Dataset 4**. Negative sign, 20% of large values among small and medium values. Results should be equivalent to the results of Dataset 3.
- **Dataset 5**. Both signs, 20% of large values that cancel, among small values. The most accurate algorithms should sum the two largest values first. In a more general case, the best algorithms should sum in decrease order of absolute values. It is a classic ill-conditioned summation.
- **Dataset 6**. Both signs, few small values and same proportion of large and medium values. Only large values cancel. The best algorithms should sum in decrease order of absolute values.
- **Dataset 7**. Both signs, few small values and same proportion of large and medium values. Large and medium values are ill-conditioned. Results should be the same than in Dataset 6.
- **Dataset 8**. Both signs, few small values and same proportion of large and medium values. Only medium values cancel. Results should be the same than in Dataset 6.
- **Dataset 9**. In order to simulate data encounted in embedded systems, this dataset is composed of intervals defined by $[0.4, 1.6]$. This is representative of values sent by a a sensor to an accumulator. This dataset is well-conditioned.

Example of data generated for Dataset 1:
$a = [2.667032062476577e^{16}, 3.332967937523422e^{16}]$
$b = [1.778021374984385e^{-16}, 2.221978625015614e^{-16}]$
$c = \ldots$ etc.

Figure 5 shows the proportion of optimal algorithms, i.e. the ones which return the smallest error with each dataset for the corresponding level of parallelism. Each proportion is the average value for the ten samples within each dataset. Parallelism degrees are $O(\lfloor \log(n) \rfloor + 1)$, $O(\lfloor \log(n) \rfloor + k)$, $O(k \times \lfloor \log(n) \rfloor)$, and $O(n)$ which describe all the algorithms of all levels of parallelism, as defined in Subsection 3.1.

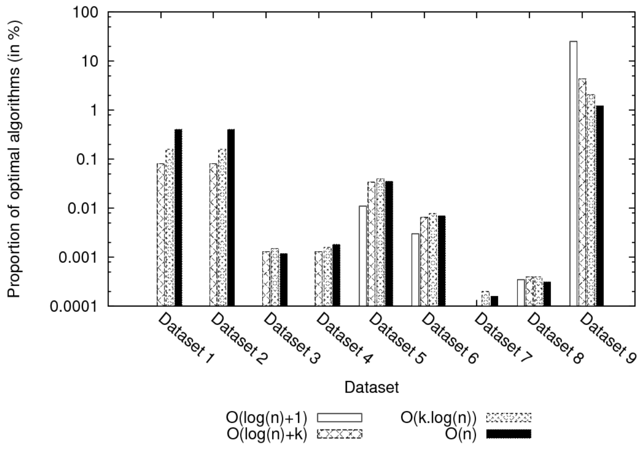


Figure 5: Proportion of the optimal algorithm on ten terms summation (average on 10 datasets).

First, we can observe that the proportion of optimal algorithms is tiny: the average of optimal algorithms with respect to the best accuracy is less than one percent except for the well-conditioned Dataset 9. Results in Table 1 match those displayed in Figure 5. In most cases, among all the levels of parallelism, the highest degree in $O(\lfloor \log(n) \rfloor + 1)$ is not able to keep the most accurate algorithms, particularly when there is absorption (percentage equals zero and no bar is plotted). We observe that the more the level of parallelism is, the harder it is to find a good algorithm. But if we relax the time constraint, i.e. the parallelism, it is easier to get an optimal algorithm.

For example, results of Dataset 1 show that if we limit the algorithms to all the algorithms of complexity $O(\lfloor \log(n) \rfloor + 1)$ there are no algorithm with the best error. If the level of parallelism is not so good, for example $O(\lfloor \log(n) \rfloor + k)$ or $O(k \times \lfloor \log(n) \rfloor)$ there are algorithms with the best errors.

Results in Figure 5 show that for Dataset 9, the proportion of optimal algorithms with the highest degree of parallelism is larger than the ones with less parallelism. In this case of well-conditioned summation, it reflects that whereas there are less algorithms of this parallelism level, these ones do not particularly suffer from inaccuracy. For well-conditioned summation, it seems that it is easier and easier to find an optimal algorithm as parallelism increases.

4. GENERATION OF THE ALGORITHMS

In this section, we describe how our tool generates all the algorithms. Our program, written in C++, builds all the reparsing of an expression. In the case of summation, the combinatory is huge, so it is very important to reduce the reparsing to the minimum.

The combinatory of summation is important, this was often studied but no general solution exists. For example, see CGPE [17] which computes equivalent polynomial expressions.

Intuitively, to generate all the expressions for a sum of n terms we process as follows.

- Step 1 : Generate all the parsings using the associativity of summation $((a+b)+c = a+(b+c))$. The number of parsings is given by the Catalan Number C_n [18]:

$$C_n = \frac{(2n)!}{n!\,(n+1)!}$$

- Step 2 : Generate all the permutations for all the expressions found in Step 1 using the commutativity of summation $(a+b = b+a)$. There is $n!$ ways to permute n terms in a sum.

So, the total number of equivalent expressions for a n terms summation is

$$C_n \cdot n! \tag{7}$$

Figure 6 shows this first combinatory result.

Our tool finds all the equivalent expressions of an original expression but it only generates the different equivalent expressions. For example, $a+(b+c)$ is equivalent to $a+(c+b)$ but it is not different because it corresponds to the same algorithm: these expressions correspond to the same sequence of operation. In Subsection 4.1, we present how we generate the structurally different trees and in Subsection 4.2, how we generate the permutations.

Table 4 and Figure 6 represent the number of algorithms generated for n terms as n grows.

The summation is a complex case, for example CGPE [17] generate equivalent polynomial expressions using heuristics to find a result as fast as possible. We want to do a study on exhaustive expressions reparsings, so because the combinatory is huge, we use ten terms during this work.

Terms	All expressions	Different expressions
5	1680	120
10	$1.76432e^{+10}$	$4.66074e^{+07}$
15	$3.4973e^{+18}$	$3.16028e^{+14}$
20	$4.29958e^{+27}$	$1.37333e^{+22}$

Table 4: Number of terms and expressions.

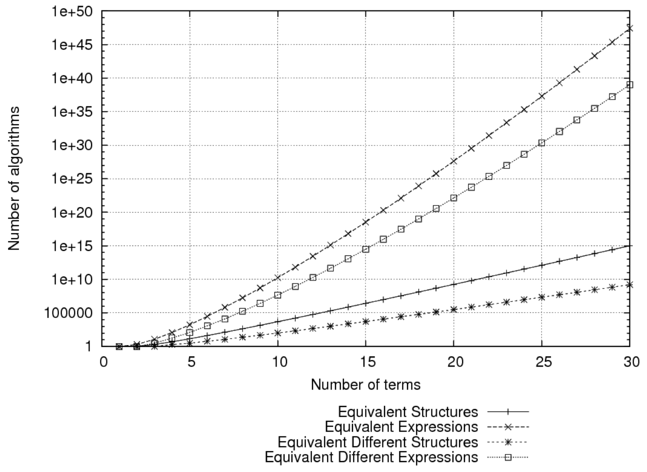


Figure 6: Number of trees when summing n terms.

4.1 Exhaustive Generation of Structurally Different Trees

We represent one algorithm with one binary tree. Nodes are sum operators and leaves are values. We describe how to generate all the structurally different trees. It is a recursive method defined as follows.

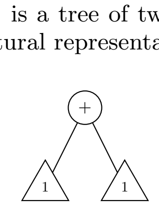
- We know that the number of terms is $n \geq 1$. An expression is composed of one term at least.

- A leaf x has only one representation, it is a tree of one term represented like this: $\triangle_1$.

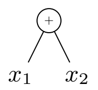
  x

  Then the number of structures for one term trivially reduces to one.

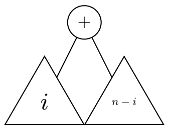
- Expression $x_1 + x_2$ is a tree of two terms $\triangle_2$. It has the following structural representation.



  With two terms we can create only one tree. So again the number of structures for two terms equals 1.



- Recursively, we apply the same rules. For a tree of n terms, we generate all the different structural trees for all the possible combinations of sub-trees, i.e. for all $i \in [1, n-1]$, two sub-trees with, respectively, i and $(n-i)$ terms. Because summation is commutative, it is sufficient to generate these $(i; n-i)$-sub-trees for all $i \in [1, \lfloor \frac{n}{2} \rfloor]$. This is represented as follows:

$\forall i \in [1, \lfloor \frac{n}{2} \rfloor],$ 

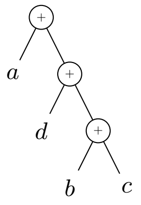
- So, for n terms, we generate the following numbers of structurally different trees,

$$S_{truct}(1) = S_{truct}(2) = 1, \quad (8)$$

$$S_{truct}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} S_{truct}(n-i) \cdot S_{truct}(i). \quad (9)$$

4.2 Generation of Permutations

To generate only different permutations, the leaves are related to the tree structure. For example, we do not wish to have the following two permutations, $a + (d + (b + c))$ and $a + ((c + b) + d)$.



Indeed, these expressions have the same accuracy and the same degree of parallelism.

In order to generate all the permutations, we use a similar algorithm as in the previous subsection.

- Firstly, we know that for an expression of one term, we may generate only one permutation. $P_{erm}(1) = 1$.
- Using our permutation restriction, it is sufficient to generate one permutation for an expression of two terms; so, again, $P_{erm}(2) = 1$.
- Permutations are related to the tree structures and we count them with the following recursive relation,

$$P_{erm}(1) = P_{erm}(2) = 1, \quad (10)$$

$$P_{erm}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \binom{i}{n} \cdot P_{erm}(n-i) \cdot P_{erm}(i). \quad (11)$$

Equations $S_{truct}(n)$ and $P_{erm}(n)$ are asymptotically exponential.

5. FURTHER EXAMPLES

In this section, we present results for larger or more sophisticated examples. Subsection 5.1 introduces a sum of twenty terms, Subsection 5.2 focuses on compensation and we discuss about bounded parallelism in Subsection 5.3

5.1 An Example Over More Terms

We now consider a sum of 20 terms. We chose a dataset where all the values belong to the interval $[0.4, 1.6]$. Again this is representative, for example, of what may happen in an embedded system when accumulating values provided by a sensor, like a sinusoidal signal.

Critical path	**Average of optimal algorithms (%)**	
$\lfloor \log(n) \rfloor + 1$	54.08	
	$k = 2$	$k = 3$
$\lfloor \log(n) \rfloor + k$	19.75	12.41
$k \times \lfloor \log(n) \rfloor$	5.26	4.15
n	4.13	

Table 5: Proportion of optimal algorithms.

We can see that the results in Table 5 are similar to the results of Dataset 9. We obtain the same repartition of optimal algorithms with ten or twenty terms. This confirms that the sum length does not govern the accuracy – at least while overflow does not appear.

In this case, we show that for a sum of identical intervals, the more parallelism, the easier to find an algorithm which preserves the maximum accuracy.

5.2 Compensated Summation

Now we present an example to illustrate one of the core motivation of this work. The question is the following: Starting from the simplest sum algorithm, are we able to automatically generate a compensated summation algorithm that improves the evaluation ? Here we describe how to introduce one level of compensation as in the algorithms presented in Section 2 (TwoSum, Sum2, SumComp).

To improve the accuracy of expression E, we compute an expression E_{cmp}.

For values X and Y, we introduce the function $C(X, Y)$ which computes the compensation of $X \oplus Y$ (like in Algorithm TwoSum, see section 2.1).

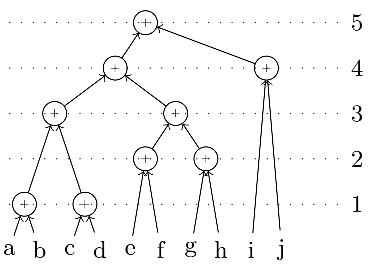
For example, for three terms we have:
$E = (X \oplus Y) \oplus Z$
$E_{cmp} = \left[((X \oplus Y) \oplus C(X, Y)) \oplus Z\right] \oplus C(X + Y, Z)$

E_{cmp} is the expression we obtain automatically by systematically compensating the original sums. It could be generated by a compiler. To illustrate this, we present one example with a summation of five terms $((((a + b) + c) + d) + e)$. Terms are defined as follows,

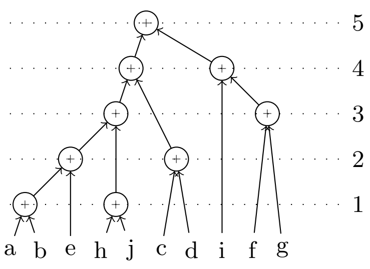
$$\begin{aligned} a &= -9.5212224350e^{-18} \\ b &= -2.4091577979e^{-17} \\ c &= 3.6620086288e^{+03} \\ d &= -4.9241247828e^{+16} \\ e &= 1.4245601293e^{+04} \end{aligned}$$

As before we can identify the two followings cases. The maximal accuracy which can be obtained, among all the reparsing of this five terms expression, is given by the following algorithm: $(((a+b)+c)+e)+d$. It generates the absolute error $\Delta = 4.0000000000020472513$. We observe that this algorithm is Algorithm 1 at Section 2 with increase order.

The maximal accuracy given by the maximal level of parallelism is obtained by the algorithm $((a + c) + (b + e)) + d$.

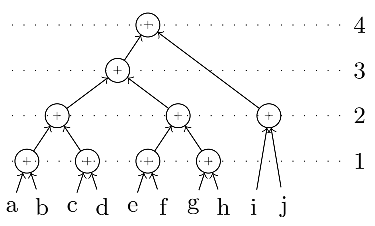


a) Fully Parallel and Non-Fully Accurate Algorithm.

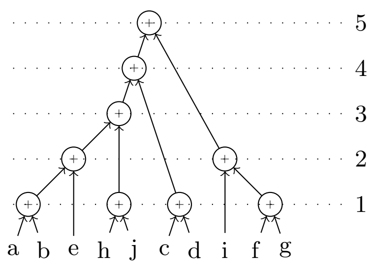


b) Fully Accurate and Non-Fully Parallel Algorithm.

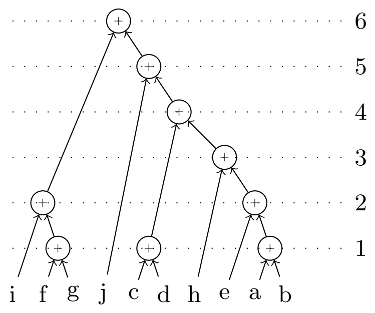
Figure 7: Dataflow Graphs of Algorithms in Bounded Parallelism on 2sums/cycle architecture.



a) Fully Parallel and Non-Fully Accurate Algorithm.



b) Fully Accurate and $\log(n) + k$ Parallelism Degree Algorithm.



c) Fully Accurate and $k \times \log(n)$ Parallelism Degree Algorithm.

Figure 8: Dataflow Graphs of Algorithms in Bounded Parallelism on 4sums/cycle architecture.

In this case, the absolute error is

$$\delta_{nocomp} = 4.0000000000029558578.$$

When applying compensation on this algorithm, we obtain the following algorithm :

$$(f + (g + (h + i))) + (d + ((b + e) + (a + c))),$$

with :

$$\begin{gathered} f = C(a,c) = -9.5212224350000e^{-18} \\ g = C(b,e) = -2.4091577978999e^{-17} \\ h = C(f,g) = -1.8189894035458e^{-12} \\ i = C(h,d) = 3.6099218000017 \end{gathered}$$

Now, we measure the improved absolute error $\delta_{comp} =$ 4.0000000000000008881. It happens that this algorithm found with the application of compensation is actually the Sum2 algorithm –Algorithm 6 at Section 2. This results illustrates that we can automatically find algorithms existing in the bibliography and that the transformation improves the accuracy.

5.3 Bounded Instruction Level Parallelism

Section 3 showed that in the case of maximum parallelism, maximum accuracy is not possible (or very difficult) to have. The fastest algorithms ($O(\lfloor\log(n)\rfloor + 1)$ are rarely the most accurate but by relaxing the time constraint, it becomes possible to find an optimally accurate algorithm.

This subsection is motivated by the following fact. In processor architectures, parallelism is bounded. So it is possible to execute an algorithm less parallel in the same execution time, or in a very closed time, as the fastest parallel one. We show here two examples to illustrate this. Firstly we use a processor which executes two sums per cycle and secondly one which executes four sums per cycle.

For an expression of ten terms :

- **2 sums/cycle**:

  The execution of the fastest algorithm ($\lfloor\log(n)\rfloor + 1$) for the expression does not provide the maximum accuracy. It takes five cycles to compute the expression as shown in a), in Figure 7. Now we take another algorithm, with less parallelism but that provides the maximum accuracy (See Line 1, Table 2, Subsection 3.1). In bounded parallelism this algorithm takes the same time than the more parallel one as shown in b), in Figure 7.

- **4 sums/cycle**:

  Again, execution of the fastest algorithm ($\lfloor\log(n)\rfloor+1$) of this expression, do not have the maximum accuracy. It takes four cycles to compute the expression (See a), in Figure 8). We take two other algorithms, both with less parallelism but with the maximum accuracy. The first algorithm is described at Line 1, Table 2, Subsection 3.1. It takes one more cycle than the most parallel one (See b), in Figure 8). The second algorithm is in $k \times \lfloor\log(n)\rfloor$; it corresponds to Line 2, Table 3, Subsection 3.1. This one takes two more cycles than the most parallel one (See c), in Figure 8).

This confirms our claim that in current architectures, we can improve accuracy without lowering too much the execution.

6. CONCLUSION AND PERSPECTIVES

We have presented our first steps towards the development of a tool that aims at automatically improving the accuracy of numerical expressions evaluated in floating-point arithmetic. Since we target to embed such tool within compiler, introducing more accuracy should not jeopardize the improvement of running-time performances provided by the optimization steps. This motivates to study the simultaneous improvement of accuracy and timing. Of course we exhibit that a trade-off is necessary to generate optimal transformed algorithms. We validated the presented tool with summation algorithms; these are simple but significant problems in our application scope. We have shown that this trade-off can be automatically reached, and the corresponding algorithm generated, for data belonging to intervals – and not only scalars. These intervals included ill-conditioned summations. In the last section, we have shown that we can automatically generates more accurate algorithms that use compensation techniques. Compared to the fastest algorithms, the overcost of these automatically generated more accurate algorithms may be reasonable in practice. Our main conclusion is that relaxing very slightly the time constraints by choosing algorithms whose critical paths are a bit longer than the optimal makes it possible to strongly optimize the accuracy.

Next step needs to increase the complexity of the case study both performing more operations and different ones. One of the main problem to tackle is the combinatory of the possible transformations. Brute force transformation should be replaced using heuristics or more sophisticated transformations as, e.g. the error-free ones we introduced to recover the compensated algorithms. Another point to explore is how to develop significant datasets corresponding to any data intervals provided by the user of the expression to transform. A further step will be to transform any expression up to a prescribed accuracy and to formally certified it. Such facility is for example necessary to apply such tool for symbolic-numeric algorithms. In this scope, this project plans to use static analysis and abstract interpretation as in [12].

7. REFERENCES

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, revision of std 754-1985 edition, 2008.

[2] James Demmel. Trading off parallelism and numerical stability, 1992.

[3] Stef Graillat and Philippe Langlois. Real and complex pseudozero sets for polynomials with applications. *Theoretical Informatics and Applications*, 41(1):45–56, 2007.

[4] Nicholas J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14:783–799, 1993.

[5] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[6] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Faster floating-point square root for integer processors. In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, July 2007.

[7] Claude-Pierre Jeannerod and Guillaume Revy. Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores. In *Proceedings of the 43rd Asilomar Conference on Signals, Systems, and Computers (Asilomar'09)*, Pacific Grove, CA, USA, november 2009.

[8] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] Philippe Langlois. Compensated algorithms in floating point arithmetic. In *12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics, Duisburg, Germany*, September 2006. (Invited plenary speaker).

[10] Jean Louis Lions, Remy Hergott (CNES), Bernard Humbert (Aerospatiale), and Eric Lefort (ESA). Ariane 5, flight 501 failure, report by the inquiry board. european space agency, 1996.

[11] Matthieu Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19:7–30, 2006.

[12] Matthieu Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Journal of Formal Methods in System Design*, 35:265–278, 2009.

[13] United States General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Report GAO/IMTEC-92-26, Information Management and Technology Division, Washington, D.C., February 1992.

[14] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26, 2005.

[15] Michèle Pichat. Correction d'une somme en arithmétique à virgule flottante. *Applied Numerical Mathematics*, 19:400–406, 1972.

[16] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David W. Matula, editors, *Proc. 10th IEEE Symposium on Computer Arithmetic*, pages 132–143. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[17] Guillaume Revy. *Implementation of binary floating-point arithmeric on embedded integer processors.* PhD thesis, École Normale Supérieure de Lyon, 2009.

[18] Fred S. Roberts. *Applied combinatorics.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[19] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, 2009.

[20] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation –part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[21] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, New York, second edition, 1993.

Polynomial Homotopies on Multicore Workstations[*]

Jan Verschelde
Dept of Math, Stat, and CS
University of Illinois at Chicago
851 South Morgan (M/C 249)
Chicago, IL 60607-7045, USA
jan@math.uic.edu

Genady Yoffe
Dept of Math, Stat, and CS
University of Illinois at Chicago
851 South Morgan (M/C 249)
Chicago, IL 60607-7045, USA.
gyoffe2@uic.edu

ABSTRACT

Homotopy continuation methods to solve polynomial systems scale very well on parallel machines. We examine its parallel implementation on multiprocessor multicore workstations using threads. With more cores we speed up pleasingly parallel path tracking jobs. In addition, we compute solutions more accurately in about the same amount of time with threads, and thus achieve quality up. Focusing on polynomial evaluation and linear system solving (key ingredients of Newton's method) we can double the accuracy of the results with the quad doubles of `QD-2.3.9` in less than double the time, if all available eight cores are used.

Categories and Subject Descriptors

G.1.5 [**Roots of Nonlinear Equations**]: Continuation (homotopy) methods

General Terms

Experimentation, Performance

Keywords

homotopy, path following, polynomial system, thread, task.

1. INTRODUCTION

PHCpack is an open source software package for homotopy continuation methods to solve polynomial systems. For surveys and general introductions on homotopy continuation methods, see [32], [35], and [43].

The first public release of PHCpack is archived in [45]. Parallel implementations of various homotopy algorithms in PHCpack have been developed jointly with Yusong Wang [46, 48], Yan Zhuang [31, 47, 50], Yun Guan [23], and Anton Leykin [29, 30, 31]. All parallel homotopy algorithms in [23, 29, 30, 31, 46, 47, 48, 50] use the message passing library MPI [41] for interprocess communication on clusters of Linux nodes, developed on personal clusters and tested on the NCSA supercomputers. Other continuation software for polynomial systems (Bertini [8], HOM4PS-2.0para [34], PHoMpara [24], and POLSYS_GLP [44]) use MPI.

PHCpack is developed with the aid of the GNU-Ada compiler which maps tasks [13] to kernel threads, The multitasking implementation of homotopies could complement the existing parallel versions in a multi-tiered approach: each node could run a more efficient multithreaded finer-grained homotopy algorithm, while the communication between the nodes happens with message passing in a coarser grain. Granularity issues in homotopy algorithms are discussed in [3].

We first describe multithreaded versions of the most commonly used homotopy continuations in PHCpack. As expected from pleasingly parallel computations, the speedup of the multithreaded code is close to optimal. The second part of this paper is devoted to a "quality up" (defined in [2]) using multiprecision arithmetic instead of the standard hardware floats. Because of locks placed on memory allocations of the existing multiprecision of PHCpack, we have started to use the quad doubles as available in the software `QD-2.3.9` [25]. A quad double is an unevaluated sum of four doubles and quadruples the working precision from $\epsilon = 2.204 \times 10^{-16}$ to $\epsilon^4 = 2.430 \times 10^{-63}$. Extending the precision in this manner was introduced in [17], see also [38] and [40].

Performing arithmetic with operations defined by software instead of hardware multiplies the cost with a certain factor. Experimentally we report in this paper that the multiplication factors in raising the level of precision from double to double double, and from double to quad double are close to the number eight, which is the number of available cores on our computer. Using all available cores properly, we can compensate for the extra cost of working with extended precision and thus obtain a quality up.

Our computations happened on a on a Mac OS X Pro with 2 Quad-Core Intel Xeons at 3.2 Ghz, bus speed at 1.6 Ghz, 12 Mb L2 cache per processor and 8Gb memory. We used the GNAT GPL 2009 edition of the GNU-Ada compiler. The code is available in version 2.3.56 of PHCpack. We also use `QD-2.3.9` via its C interface, compiled with the `g++` compiler (gcc version 4.0.1), including the `pthread` library.

Acknowledgements. Preliminary versions of this work were presented by the first author at a session of ACA 2009 and at a minisymposium of the SIAM PP10 conference. The first author is grateful to the organizers of these meetings. We thank the reviewers for their comments.

[*]Date: 17 June 2010. This material is based upon work supported by the National Science Foundation under Grant No. 0713018.

2. SPEEDING UP PATH TRACKERS

In this section we describe the use of threads to speedup three different types of homotopies: cheater [33] [36], polyhedral [26], and monodromy breakup [42]. First we outline the manager/worker paradigm applied to multithreading.

2.1 Job Scheduling Algorithms

When we use the MPI library, we typically apply the manager/worker paradigm. One manager node (or process) maintains a job queue. Worker nodes (or processes) send messages to request jobs. If the received job is not the termination signal, then after computing the job, the results of the job are sent back to the manager. This paradigm scales very well for thousands of nodes and for systems of millions of solutions.

On smaller multiprocessor multicore computers, one could of course also run MPI, but in that case one may lose one entire process to the manager. Because of the irregularities of the path tracking jobs – it is hard to predict in advance how much work one job will take – the manager must always be available to serve the worker processes with a new job. Therefore one of the available cores must be assigned to the manager and is therefore unavailable for computational jobs. On a system with 8 cores, this leaves only 7/8 or at most 87.5% of the total capacity available for computation.

The multithreaded version of the manager/worker paradigm uses one main thread to launch the worker threads. The main thread initializes the job queue but leaves the management of the pointer to the current job to the worker threads. If a worker needs a new job, it will request a lock (or semaphore) and adjust the pointer to the current job in a critical section. After adjusting the pointer, the lock is released [13]. Semaphores are supported in the GNU-Ada compiler [1].

Starting worker tasks in Ada is relatively simple and illustrated with code in Figure 1. The procedure `Workers` calls the `Job` procedure, which executes code based on the `id` number.

```
procedure Workers ( n : in natural ) is
   task type Worker ( id,n : natural );
   task body Worker is
   begin
     Job(id,n);
   end Worker;
   procedure Launch_Workers ( i,n : in natural ) is
     w : Worker(i,n);
   begin
     if i < n
      then Launch_Workers(i+1,n);
     end if;
   end Launch_Workers;
begin
  Launch_Workers(1,n);
end Workers;
```

Figure 1: Launching `n` workers. Each worker is an Ada task and calls the procedure `Job` with its identification number `id`.

The GNU-Ada compiler maps the Ada tasks to kernel threads. The implementation of the task scheduler is described in [39]. While our development platform is a Mac Pro with 2 Quad-Core processors, our multicore implementations have also been tested on dual core computers running Linux and Windows (win32 thread model). Also there – using the same source code – we observed good speedups.

2.2 Multitasking Polynomial Continuation

Table 1 lists preliminary timings on tracking all 924 paths in a cheater homotopy for the cyclic 7-roots problem [4]. Although the system overhead increases as we go from one to 8 threads, it constitutes only about 1% of the total wall clock time. The timings were obtained using version 2.3.45 of PHCpack, typing at the command prompt `$`

```
$ time phc -p -t8 < /tmp/input8
```

where the 8 was replaced subsequently by 4, 2, and 1.

#workers	real	user	sys	speedup
1	15.478s	15.457s	0.010s	1
2	7.790s	15.483s	0.010s	1.987
4	3.926s	15.445s	0.011s	3.942
8	1.992s	15.424s	0.015s	7.770

Table 1: Real, user, and system time in seconds (s) for tracking 924 paths of the cyclic 7-roots system using 1, 2, 4, and 8 worker threads. The speedup is the real time for one worker thread divided by the real time for 2, 4, and 8 workers.

Since version 2.3.46 of PHCpack, the blackbox solver called as `phc -b -t8` will use 8 tasks to run the polyhedral homotopies [26]. To run MPI programs, the environment of the user must be configured for `mpirun` or `mpiexec`. In contrast, using threads is done just by adding the `-t8` option to the `phc` executable.

2.3 Multitasking Polyhedral Continuation

To approximate all isolated solutions of a given system $f(\mathbf{x}) = \mathbf{0}$ with as many equations as unknowns, using polyhedral homotopies [26] we distinguish three stages:

1. Compute the mixed volume MV (aka the BKK bound) of the Newton polytopes spanned by the supports A of f via a regular mixed-cell configuration Δ_ω.
2. Given Δ_ω, solve a generic system $g(\mathbf{x}) = \mathbf{0}$, using polyhedral homotopies [26]. Every cell $C \in \Delta_\omega$ defines one homotopy
$$h_C(\mathbf{x}, s) = \sum_{\mathbf{a}\in C} c_{\mathbf{a}}\mathbf{x}^{\mathbf{a}} + \sum_{\mathbf{a}\in A\setminus C} c_{\mathbf{a}}\mathbf{x}^{\mathbf{a}} s^{\nu_{\mathbf{a}}}, \quad \nu_{\mathbf{a}} > 0, \quad (1)$$
tracking as many paths as the mixed volume of the cell C, as s goes from 0 to 1.
3. Use $(1-t)g(\mathbf{x}) + tf(\mathbf{x}) = \mathbf{0}$ to solve $f(\mathbf{x}) = \mathbf{0}$.

Stages 2 and 3 are computationally most intensive. For example, for cyclic 10-roots (MV = 35940): stage 1 takes 21 seconds (using the version of MixedVol [22] as integrated in PHCpack) whereas stage 2 lasts 39 minutes.

We point out that PHoMpara [24] provides a parallel computation of the mixed volume.

A static distribution of the workload (as used in `mpi2cell_s` developed with Yan Zhuang [47]) is shown in Figure 2.

manager	worker 1	worker 2	worker 3
$\mathrm{V}(c_1) = 5$	$\#\mathrm{p}(c_1) : 5$		
$\mathrm{V}(c_2) = 4$	$\#\mathrm{p}(c_2) : 4$		
$\mathrm{V}(c_3) = 4$	$\#\mathrm{p}(c_3) : 4$		
$\mathrm{V}(c_4) = 6$	$\#\mathrm{p}(c_4) : 1$	$\#\mathrm{p}(c_4) : 5$	
$\mathrm{V}(c_5) = 7$		$\#\mathrm{p}(c_5) : 7$	
$\mathrm{V}(c_6) = 3$		$\#\mathrm{p}(c_6) : 2$	$\#\mathrm{p}(c_6) : 1$
$\mathrm{V}(c_7) = 4$			$\#\mathrm{p}(c_7) : 4$
$\mathrm{V}(c_8) = 8$			$\#\mathrm{p}(c_8) : 8$
#p : 41	#p : 14	#p : 14	#p : 13

Figure 2: Static workload distribution of polyhedral continuation, for 8 cells $c_1, c_2, \ldots, c_8$, where V() is the mixed-volume function and #p stands for the number of paths.

In a static workload distribution, we assume that every path takes about the same amount of work. Because polyhedral homotopies solve a generic system $g(\mathbf{x}) = \mathbf{0}$, this assumption may hold, although we experienced better performance with a dynamic load balancing [47]. Table 2 shows timings for running polyhedral homotopies on a random coefficient system, distributing mixed cells, for the cyclic n-roots problems.

		#tasks, times in seconds			
n	MV	1	2	4	8
7	924	12	6	3	2
8	2560	58	29	15	8
9	11016	417	209	104	52
10	35940	2156	1068	534	270

Table 2: Running polyhedral homotopies on cyclic n-roots, with 1, 2, 4, and 8 tasks, tracking a total of MV (mixed volume) many paths.

On the same random coefficient system $g(\mathbf{x}) = \mathbf{0}$ used to solve the cyclic 10-roots problem, we compared the MPI implementation with the new multitasked code, on our 8-core Mac Pro:

- `mpirun -n 9 mpi2cell_d`: total wall time = 270.5 seconds,
- `phc -m -t8`: elapsed wall clock time is 233 seconds.

Both implementations use the same tolerances and the same numerical settings for the parameters.

In defense of the implementation using MPI, one must point out the differences in managing the list of solutions. Because the MPI implementation was set up to work for millions of solutions, the complete list of solutions is not kept in memory: start solutions are read from a file or computed when needed and after tracking a path, the end solution is also directly written to a file. This jumpstarting mechanism is described in greater detail in [31].

Rather than advocating for the exclusive use of either MPI or threads, MPI and threads should be combined in a multi-tiered implementation on supercomputers with multicore nodes.

2.4 Multitasking Monodromy Breakup

Many polynomial system arising in practical applications have positive dimensional solution sets. Keeping to our running examples, the cyclic 8-roots problem [11] has a curve of degree 144. This curve factors into 16 irreducible components, 8 factors have degree 16 and the other 8 are quadrics.

The application of monodromy to factor polynomials first appeared in [6], mainly to derive a complexity result. Actual computations were described in [15, 16] and [42]. See [14] for a nice introduction, [20], [21], [37] for recent developments, and to [28] for an application.

In Table 3 we show the outcome of running the real, user, and system timing on running ten monodromy loops. The fluctuations in the times are due to taking different random slices between the runs.

#workers	real	user	sys	speedup
1	61.203s	61.137s	0.046s	1
2	30.179s	59.888s	0.044s	2.028
4	15.472s	60.551s	0.050s	3.958
8	7.874s	59.469s	0.070s	7.773

Table 3: Real, user, and system time in seconds (s) for running 10 random loops, tracking a total of 20×144 paths to factor the solution curve of the cyclic 8-roots system, using 1, 2, 4, and 8 worker threads. The speedup is the real time for one worker thread divided by the real time for 2, 4, and 8 workers.

Using a basic parallel implementation as in [29], the experiments in Table 3 show a very good speedup. For larger number of workers, the overhead caused by scheduling and managing the irreducible decomposition leads to performance losses, remedied by the scheduling algorithms in [30].

Although we have no formal complexity bounds on the number of loops in the monodromy breakup algorithm seems to scale very well. One limiting factor — relevant for the second part of this paper — is that factoring polynomials of high degrees requires multiprecision arithmetic.

2.5 Granularity Issues

All parallel computations described in this section are coarse grained. We have many more paths to track than the available cores. Every task is in charge of multiple paths and every path can be tracked independently.

So far we obtained nice speedups with relatively little effort on multicore workstations. In addition, because the number of cores is limited and the jobs remain pleasingly parallel, there is no significant job scheduling overhead. If we can compute benchmark problems faster, then we can of course claim that we can solve more problems, but can we improve the quality of the solutions as well? Questions like these are addressed in [2] which defines "quality up".

Shifting our attention from speedup to quality up, what if we have to track, instead of many solution paths, only a few, or even only one path? Often it occurs that there are a couple of solution paths that require extra care, for which the use multiprecision arithmetic is necessary [7]. The paper [5] surveys applications of high-precision computations. As software driven arithmetic is more expensive than hardware arithmetic, we want to offset this expense using multiple cores.

3. QUALITY UP

In this section we examine quality up: given 8 cores and roughly the same amount of time, can we increase the quality of our computations? Applied to polynomial system solving, we interpret quality as accuracy. We investigate the cost of obtaining more accurate solutions using Newton's method in quad double complex arithmetic. Experiments in this section were done with an Ada translation of `QD-2.3.9` [25], available in PHCpack since version 2.3.55.

3.1 Cost Overhead of Arithmetic

Complex arithmetic is standard practice in all homotopy solvers for polynomial systems and the overhead compared to real arithmetic is taken for granted. Since Ada 95, complex arithmetic is part of the language. However, PHCpack has its own packages for complex arithmetic and does *not* use the arithmetic provided by the language. The complex arithmetic in PHCpack happens via a generic package (generic is the equivalent to template in C++), suitable to work over any real number field. The experiment below applies the same code for all complex operations, over hardware doubles, double doubles, and quad doubles.

Fully optimized code on one core of a 3.2 Ghz Intel Xeon, performed one thousand times the following steps:

1. generate a 100-by-100 random matrix A and corresponding right hand side random vector $\mathbf{b}$;
2. solve $A\mathbf{x} = \mathbf{b}$ via LU factorization and substitutions $L\mathbf{y} = \mathbf{b}$, $U\mathbf{x} = \mathbf{y}$;
3. print $||\mathbf{b} - A\mathbf{x}||_\infty$ with 3 decimal places.

User CPU times are recorded in Table 4.

type of arithmetic	user CPU seconds
`double real`	2.026s
`double complex`	16.042s
`double double real`	20.192s
`double double complex`	140.352s
`quad double real`	173.769s
`quad double complex`	1281.934s

Table 4: CPU user time for solving a random linear system one thousand times, using double real and complex, double double real and complex, quad double real and complex arithmetic on one core.

Even as we take complex arithmetic for granted, we first measure the overhead factor as `16.042/2.026 = 7.918`, `140.352/20.192 = 6.951`, and `1281.934/173.769 = 7.377`. The three factors average to 7.415.

Going from double complex to double double complex gives a factor of `140.352/16.042 = 8.749` and the multiplication factor in the cost of quad double complex over double double complex arithmetic is `1281.934/140.352 = 9.134`.

Although fluctuations in these computational experiments happen, we observe that the multiplication factors in the cost of using double double and quad double are of the same magnitude as the multiplication factor in the cost of complex arithmetic. In addition, as we have 8 cores at our disposal, we could offset the extra cost of more accurate arithmetic mapping threads of execution to all available 8 cores.

3.2 Newton's Method with QD

To illustrate quality up, we consider a pleasingly parallel computation. Given a polynomial system and a list of isolated solution, we apply Newton's method till either a specified tolerance on the size of the residual is achieved, or till the maximum number of iterations is exhausted.

As an example system, we take the cyclic 10-roots system. Because of symmetry, we refine only the generating solutions, starting at 1747 solutions accurate up to double precision. With double double complex arithmetic we set the tolerance to `1.0E-30` and we allow at most 3 iterations of Newton's method. With quad double complex arithmetic, the tolerance is set to `1.0E-60` and we allow at most 5 iterations. The results are shown in Table 5.

	`double double complex`			
#workers	real	user	sys	speedup
1	4.818s	4.790s	0.015s	1
2	2.493s	4.781s	0.013s	1.933
4	1.338s	4.783s	0.015s	3.601
8	0.764s	4.785s	0.016s	6.306
	`quad double complex`			
#workers	real	user	sys	speedup
1	58.593s	58.542s	0.037s	1
2	29.709s	58.548s	0.054s	1.972
4	15.249s	58.508s	0.053s	3.842
8	8.076s	58.557s	0.058s	7.255

Table 5: Real, user, and system time in seconds (s) for refining 1747 solutions with Newton's method in double double and quad double complex arithmetic, using 1, 2, 4, and 8 threads.

Entries in speedup columns of Table 5 are the real time on the same row divided by the real time for one worker. We observe an improved speedup as we move from double double to quad double.

Comparing the cost of quad double over double double arithmetic, we compute the factor `58.593/4.818 = 12.161`. As `12.161` is larger than the multiplication factor of `9.134` (based on the data in Table 4) we point out that to achieve an accuracy of `1.0E-60`, typically one extra iteration of Newton's method will be required, explaining the higher multiplication factor.

Considering quality up, we compare the first real time of Table 5 with the last real time: `4.818s` with `8.076s`. We conclude: if we can afford to wait twice as long, we can double the accuracy of our solutions from double double to quad double precision. As the cost overhead of using quad double complex arithmetic is compensated by the 8 cores, we have achieved quality up.

3.3 Granularity and Memory

Although threads were used in the previous section, the granularity is coarse, as the 1747 solutions were mapped to the threads (8 at most). Because the layout of quad doubles is restricted in size, one quad double is regarded as four local variables of type double and there is no memory allocation and deallocation of arrays of variable size like in general types of multiprecision. Memory (de)allocations impose locks on threads, preventing speedups.

4. TRACKING ONE PATH FASTER AND MORE ACCURATELY

In this section we focus on tracking one path with threads using double double or quad double complex arithmetic.

4.1 Multitasking Newton's Method

The computational effort to execute one step of Newton's method can be broken up into two main parts:

1. evaluate the system and the Jacobian matrix;
2. solve a linear system to update the solution.

There is more to path tracking than Newton's method, although Newton's method is the most computationally intensive ingredient. Complexity questions often reduce to estimating bounds on the domains of quadratic convergence [12]. A report on practical work of certified homotopies appears in [9]. Parallel algorithms for linear algebra are explained in [10] and [18].

Sparse and low degree polynomials can be evaluated fast and the cost of solving the linear system will dominate the total cost of Newton's method. The situation may be reversed for large degree polynomials. From the perspective of achieving good speedups, we better assume we are dealing with either dense and/or large degree polynomial systems.

A detailed study of the efficient evaluation of polynomials for homotopy methods appeared in [27].

4.2 Polynomial System Evaluation

If we store all monomials of a polynomial system in one vector and its coefficients in a matrix (with proper correspondence to the monomial vector), then polynomial system evaluation is turned into a matrix-vector product. While effective for dense polynomials (all monomials up to a certain degree appear with nonzero coefficient), sparse polynomials will generate sparse coefficient matrices. Our running example starts with 30 polynomials, each with 30 monomials with nonzero random complex coefficients in 30 variables. Along with the Jacobian matrix, this leads to a system of 930 polynomials with a total of 11540 distinct monomials.

We represent a sparse polynomial

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in A} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}}, \quad c_{\mathbf{a}} \in \mathbb{C} \setminus \{0\}, \ \mathbf{x}^{\mathbf{a}} = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}, \quad (2)$$

collecting the exponents in the support A in a matrix E, as

$$F(\mathbf{x}) = \sum_{i=1}^{m} c_i \mathbf{x}^{E[k_i,:]}, \quad c_i = c_{\mathbf{a}}, \ \mathbf{a} = E[k_i,:] \quad (3)$$

where k is an m-vector linking exponents to rows in E: $E[k_i,:]$ denotes all elements on the k_ith row of E. Storing all values of the monomials in a vector V, evaluating F (and f) is equivalent to making an inner product:

$$F(\mathbf{x}) = \sum_{i=1}^{m} c_i V_{k_i}, \quad V = \mathbf{x}^E. \quad (4)$$

Because we consider also all derivatives of all polynomials, we could exploit relations between the monomials and put evaluated powers in cache. In our first parallel evaluation algorithm, all monomials are evaluated independently, potentially by different tasks. In our running example, evaluating 11540 monomials of degree 30 requires about 346200 multiplications, more than ten times the inner products of the 930 coefficient vectors with the corresponding 30 values of the monomials. Since the evaluation of the monomials dominates the entire calculation, in our first parallel evaluation algorithm, we do not interlace the computation of the inner products with the evaluation of the monomials.

If p threads (or tasks) are labeled as $0, 1, \ldots, p-1$, then the ith entry in the monomial vector is computed by the thread t for which $i \mod p = t$.

In Table 6, we summarize the computational results.

#tasks	real	user	sys	speedup
	`double double complex`			
1	`1m  9.536s`	`1m  9.359s`	`0.252s`	1
2	`0m 37.691s`	`1m 10.126s`	`0.417s`	1.845
4	`0m 21.634s`	`1m 10.466s`	`0.753s`	3.214
8	`0m 14.930s`	`1m 12.120s`	`1.711s`	4.657
	`quad double complex`			
#tasks	real	user	sys	speedup
1	`9m 19.085s`	`9m 18.552s`	`0.563s`	1
2	`4m 43.005s`	`9m 19.402s`	`0.679s`	1.976
4	`2m 24.669s`	`9m 20.635s`	`1.023s`	3.865
8	`1m 21.220s`	`9m 26.120s`	`2.809s`	6.884

Table 6: Real, user, and system time in minutes (m) and seconds (s) for evaluating a random system of 930 polynomials in 30 variables (at most 30 monomials of degree at most 30 in each polynomial), with double double and quad double complex arithmetic, at one thousand randomly generated points, using 1, 2, 4, and 8 threads.

Our first observation in Table 6 is that the system time grows as the number of tasks increase. Threads were created anew and destroyed twice for every evaluation as the evaluation of the monomial vectors was done separately from the computation of the inner products. While for double double complex arithmetic, the speedup is not so good, it is acceptable for quad double arithmetic.

Concerning quality up, we compare the sequential time of `69.536` seconds (first real time in Table 6) for double doubles with the `81.220` seconds (last real time) for quad double computations using 8 cores. Without parallel algorithms, the doubling of the accuracy takes an eightfold increase of computing time, as `559.085/69.536 = 8.040`. With 8 cores, the cost of extra accuracy is reduced to an increase of 17% of the real time, as `81.220/69.536 = 1.168`.

The system times of Table 6 are cut in half if we do not destroy and create threads between the evaluation of the monomial vector and the multiplication with the coefficient vectors. Rather than giving updated tables of timings, we describe how we avoid thread destruction and creation.

To synchronize jobs performed by p threads we maintain p flags b_i of boolean values, for $i = 0, 1, \ldots, p-1$. The ith flag b_i is true if and only if the ith thread is busy with a computational job. Before starting the next job, threads must wait till all threads are finished with their current job. The first thread manages the job counter k. When thread i finishes its job k, it sets its flag b_i to false and then waits to continue to the next job till k is increased. The first thread will increase k only when no jobs are busy. After increasing k, the first thread sets all busy flags to true.

4.3 Multithreaded Linear Algebra with real Quad Doubles

In this section we consider *real* quad doubles and work with the original QD library. Runs are done on the same computer, but the software environment is different. We use `QD-2.3.9` via its C interface, compiled with the `g++` compiler (gcc version 4.0.1), including the `pthread` library. Optimization flags were left off.

The operations we consider are (1) matrix-vector product; (2) solving a triangular linear system (back substitution); and (3) Gaussian elimination.

The setup for the matrix-vector product is as follows. We launch a number of threads and then divide the work to do *one* matrix-vector product among the threads. Because doing only one product goes too fast, we run multiple instances of multiple threaded matrix-vector products. As the matrix-vector products simulate polynomial evaluations, during the tracking of one solution path in our simulation we assume that one thousand evaluations of the system and all its partial derivatives are performed.

#threads	real	user	sys	speedup
1	28.990s	28.444s	0.022s	1
2	14.598s	28.205s	0.021s	1.986
4	7.537s	28.197s	0.017s	3.846
8	4.231s	28.394s	0.026s	6.852

Table 7: Real, user, and system time in seconds (s) to multiply a 700-by-200 matrix of quad doubles with a vector 1000 times, using 1, 2, 4, and 8 threads.

In Table 7 we report on timings with a multithreaded implementation of a matrix-vector product. The threads are created only once, when performing multiple jobs by many threads, but they wait on each other before moving on to the next job. While not optimal, the speedup attained in Table 7 is still acceptable. In Table 8, we consider matrices of smaller sizes. We observe decreasing speedups as the dimensions of the matrices decrease.

The sequel to Table 7 in our simulated path tracking with multiple threads is the solution of a triangular linear system, with times reported in Table 9. As the linear system is already in triangular form, it can be solved via back substitutions. Our parallel implementation of the algorithm for solving a triangular system is inspired by [49, §5.3.4].

Let a lower triangular n-by-n matrix L with entries $\ell_{i,j}$ and an n-vector $\mathbf{b}$ define the system $L\mathbf{x} = \mathbf{b}$. The solution vector $\mathbf{x}$ is computed via the formulas

$$x_i := \frac{1}{\ell_{i,i}} \left(b_i - \sum_{j=1}^{i-1} \ell_{i,j} x_j \right), \quad i = 1, 2, \ldots, n. \qquad (5)$$

The calculation of x_i needs the values for all previous components x_j, for $j = 1, 2, \ldots, i-1$.

Labeling p threads by $0, 1, \ldots, p-1$, the ith thread computes first the value x_i and then computes successively all values of x_{i+jp} for all j: $i + jp \leq n-1$. Starting to calculate x_{i+jp}, the ith thread does not wait until all values of x_k, for all k: $k < i + jp$ are computed. At first, without waiting, the ith thread computes the partial sum

$$\sum_{j=1}^{m-1} \ell_{i+jp,j} x_j \quad \text{of} \quad \sum_{j=1}^{i+jp-1} \ell_{i+jp,j} x_j, \qquad (6)$$

100-by-100 matrix

#threads	real	user	sys	speedup
1	2.031s	2.028s	0.003s	1
2	1.020s	2.030s	0.004s	1.991
4	0.514s	2.031s	0.004s	3.951
8	0.272s	2.129s	0.005s	7.467

50-by-50 matrix

#threads	real	user	sys	speedup
1	0.510s	0.508s	0.002s	1
2	0.257s	0.508s	0.003s	1.984
4	0.137s	0.532s	0.003s	3.723
8	0.078s	0.590s	0.004s	6.538

20-by-20 matrix

#threads	real	user	sys	speedup
1	0.085s	0.083s	0.002s	1
2	0.045s	0.085s	0.002s	1.889
4	0.025s	0.086s	0.002s	3.340
8	0.016s	0.104s	0.003s	5.313

Table 8: Real, user, and system time in seconds (s) for 1000 matrix-vector multiplications with matrices of quad doubles, using 1, 2, 4, and 8 threads.

#threads	real	user	sys	speedup
1	41.877s	41.770s	0.018s	1
2	21.365s	42.631s	0.026s	1.960
4	10.986s	43.376s	0.026s	3.812
8	5.984s	47.097s	0.030s	6.998

Table 9: Real, user, and system time in seconds (s) for 10,000 on solving a 200-by-200 triangular matrix of quad doubles, using 1, 2, 4, and 8 threads.

where m is the number of x_k values computed by the time it starts calculation of x_{i+jp}. Then the thread merely proceeds with the computing the remaining part of the sum for x_{i+jp} as long as each next x_k appearing in it is computed.

To synchronize the calculations, we keep an array of status flags associated to the variables. The status flag of a variable is updated by the processor which computes that particular variable after the calculation of the variable is complete. Other threads which need the value of the variable must wait till the status flag of the variable has been updated.

Adjusting the algorithm described above to solve upper triangular linear systems happens by reversing the indices of $\mathbf{x}$, i.e.: $x_i = x_{n-i}$, for $i = 1, 2, \ldots, n$.

Larger values of n/p lead to larger speedups, since then for large indexes $i+jp$ by the time the ith thread starts computing x_{i+jp} almost all x_k with $k < i+jp$ are already computed, thus the computations of x_{i+jp} with large indexes are done almost uninterruptedly with relatively very few short breaks for the ith thread to wait until several preceding values of x_k to x_{i+jp} are calculated.

The analogue to Table 8 for back substitutions is Table 10. In Table 10 we display times for back substitutions of various dimensions. As before, we observe decreasing speedups in Table 10 as the dimensions of the matrices decrease. However, it seems that the threshold dimension for achieving

150-by-150 matrix

#threads	real	user	sys	speedup
1	23.661s	23.650s	0.009s	1
2	12.167s	24.311s	0.014s	1.945
4	6.370s	24.974s	0.014s	3.714
8	3.530s	27.590s	0.022s	6.703

100-by-100 matrix

#threads	real	user	sys	speedup
1	10.752s	10.743s	0.009s	1
2	5.588s	11.069s	0.007s	1.924
4	2.868s	11.451s	0.007s	3.749
8	1.866s	14.624s	0.012s	5.762

50-by-50 matrix

#threads	real	user	sys	speedup
1	2.802s	2.799s	0.003s	1
2	1.487s	2.963s	0.006s	1.884
4	0.859s	3.420s	0.004s	3.262
8	0.640s	5.088s	0.006s	4.378

20-by-20 matrix

#threads	real	user	sys	speedup
1	0.510s	0.509s	0.002s	1
2	0.310s	0.614s	0.002s	1.645
4	0.220s	0.866s	0.003s	2.318
8	0.264s	2.066s	0.005s	1.932

Table 10: Real, user, and system time in seconds (s) for solving 10,000 triangular systems of quad doubles, using 1, 2, 4, and 8 threads.

good speedups lies higher with multithreaded back substitution than with matrix-vector products. Note that path tracking typically uses complex arithmetic, whereas the computations with quad double are fully real: the threshold dimension for good speedups is lower for complex arithmetic.

To solve a linear system $A\mathbf{x} = \mathbf{b}$, we apply row reduction on the augmented matrix $[A\ \mathbf{b}]$. Denoting the entries of A by $a_{i,j}$, formulas using pivot row i, for i ranging from 1 to $n-1$ are

$$a_{j,k} := a_{j,k} - \frac{a_{j,i}}{a_{i,i}} a_{i,k}, \quad k = j, j+1, \ldots, n, \tag{7}$$

and on $\mathbf{b}$: $b_j := b_j - (a_{j,i}/a_{i,i})b_i$, for j ranging from i to n. Note that these formulas do not perform row interchanges (partial pivoting) to increase the numerical stability. Partial pivoting is done by the algorithms in the next section.

To achieve an equal workload among p threads, we assign rows to threads as follows: the first thread will work on rows $1, p+1, 2p+1, \ldots$, the second thread will work on rows $2, p+2, 2p+2, \ldots$, in general: the ith thread works on rows $i+pj$ for all natural values of j starting at 0 and increasing as long as $i+pj \leq n$. As the pivot row increases, the difference between workloads among the threads is never more than one.

For correctness, threads are synchronized so no threads starts updating rows for the next pivot until all threads have finished updating their rows for the current pivot row. This synchronization is the same as described at the end of section 4.2.

Timings on our multithreaded code for Gaussian elimi-

100-by-100 matrix

#threads	real	user	sys	speedup
1	71.464s	71.436s	0.023s	1
2	37.129s	74.222s	0.026s	1.925
4	18.903s	75.557s	0.031s	3.781
8	10.721s	84.655s	0.116s	6.666

50-by-50 matrix

#threads	real	user	sys	speedup
1	9.354s	9.345s	0.009s	1
2	5.049s	10.086s	0.007s	1.853
4	2.583s	10.314s	0.008s	3.621
8	1.522s	12.001s	0.026s	6.146

20-by-20 matrix

#threads	real	user	sys	speedup
1	0.655s	0.653s	0.002s	1
2	0.368s	0.731s	0.003s	1.780
4	0.219s	0.864s	0.003s	2.991
8	0.166s	1.290s	0.006s	3.946

Table 11: Real, user, and system time in seconds (s) for performing 1000 times Gaussian elimination, using 1, 2, 4, and 8 threads.

nation are reported in Table 11. We observe a very good speedup for dimensions 50 and 100. Even with a small number of variables such as 20, the speedup is still acceptable. Compared to the timings for the back substitutions in Table 10 (where we had to do ten thousand runs), it is definitely worthwhile to use multiple threads for the row reduction stage of Newton's method.

4.4 Multithreaded Linear Algebra with Complex Quad Doubles

The routines to solve linear systems in PHCpack are based on LINPACK [19], and in particular on `ZGEFA` and `ZGESL` to solve linear systems via an LU factorization.

In this section we report on experiments with a basic multithreaded version of our LU factorization routines. Following the same synchronization mechanism as described with polynomial evaluation, the first thread is in charge of updating the column counter. The first thread also takes care of the pivoting, i.e.: the selection of the largest element in the current column used as the denominator in formulas (7). The computational results are summarized in Table 12.

In Table 12 we observe a significant increase in the user and system time for 8 tasks. The increase could be due to threads spending a significant amount of time in busy waiting loops, waiting for the pivoting.

Looking at speedups, we see that for double double complex arithmetic, results start to deteriorate once we go past four threads. The speedups are better with quad double complex arithmetic. Comparing the first real time with the last one in Table 12, we see that the expense of doubling the precision from double double to quad double is compensated by the 8 threads.

4.5 Newton's Method with Threads

Comparing Tables 6 and 12 we see that a LU factorization of an 80-by-80 matrix takes only slightly more time than the evaluation of a sparse system in 30 variables with all

double double complex				
#tasks	real	user	sys	speedup
1	1m 8.173s	1m 8.074s	0.131s	1
2	0m 36.712s	1m 13.061s	0.249s	1.857
4	0m 21.565s	1m 25.035s	0.455s	3.161
8	0m 20.986s	1m 42.156s	2.270s	3.248
quad double complex				
#tasks	real	user	sys	speedup
1	10m 12.216s	10m 11.900s	0.311s	1
2	5m 12.753s	10m 24.774s	0.477s	1.958
4	2m 42.653s	10m 48.795s	0.699s	3.764
8	1m 33.234s	12m 17.653s	1.930s	6.566

Table 12: Real, user, and system time in minutes (m) and seconds (s) for computing one thousand times the LU factorization of an 80-by-80 matrix of random double double and quad double complex numbers, using 1, 2, 4, and 8 threads.

its 900 partial derivatives. If we perform the operations for the same number of variables, e.g.: at 30, then the time for linear algebra shrinks significantly, or e.g.: at 80, then the time for polynomial evaluation will increase significantly. In either case, the time for polynomial evaluation dominates and leads already in relatively low number of variables to good speedups.

Because the cost of LU factorization and triangular linear system solvers is respectively $O(n^3)$ and $O(n^2)$, for n variables, and because the number of threads is typically much smaller than n, the cost of the LU factorization will remain dominant, even if we would not run the back substitution on multiple cores.

To avoid the overhead from thread destruction and creation, threads will stay alive for all iterations of Newton's method along a path. The synchronization algorithm is the same as described above, with one thread managing a job counter. There is a second level of job counters to coordinate the stages inside a Newton iteration.

5. CONCLUSIONS

As expected, using threads for pleasingly parallel computations leads rather directly to good speedups. Because the number of cores remains limited on workstations, we do not encounter problems to scale the calculations to hundreds or thousands of processors.

The more convenient thread model offers the opportunity for quality up: can we compute the solutions more accurately with multiple cores in roughly the same amount of time? Arbitrary multiprecision arithmetic with dynamic memory allocation imposes locks, preventing speedups, so we turned to the quad double arithmetic implemented by the software library `QD-2.3.9`.

Experimental results on solving linear systems, showed that going from double complex to double double complex increased the computation time by a factor of about 8.7. Using quad double complex arithmetic over double double arithmetic multiplied the user CPU times by a factor of 9.1. These experimental factors are slightly above eight, the number of available cores on our workstation, so we may potentially offset these factors using all available cores.

Placing the focus on polynomial evaluation and linear system solving — the computational ingredients of Newton's method — we experienced good speedups using quad double arithmetic. In particular, doubling the accuracy from double double to quad double can be done in less than double the time if we use all eight cores.

6. REFERENCES

[1] AdaCore. The GNAT Reference Manual. At `http://gcc.gnu.org/onlinedocs/gnat_rm`.

[2] S.G. Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.

[3] D.C.S. Allison, A. Chakraborty, and L.T. Watson. Granularity issues for solving polynomial systems via globally convergent algorithms on a hypercube. *J. of Supercomputing*, 3:5–20, 1989.

[4] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation (ISSAC'91)*, pages 101–111. ACM, 1991.

[5] D.H. Bailey, R. Barrio, and J.H. Borwein. High precision computation: Mathematical physics and dynamics. Joint SIAM-RSME-SCM-SEMA Meeting on Emerging Topics in Dynamical Systems and Partial Differential Equations (DSPDEs'10), 31 May 2010, Barcelona, Spain.

[6] C. Bajaj, J. Canny, T. Garrity, and J. Warren. Factoring rational polynomials over the complex numbers. *SIAM J. Comput.*, 22(2):318–331, 1993.

[7] D. J. Bates, J.D. Hauenstein, A.J. Sommese, and C.W. Wampler. Adaptive multiprecision path tracking. *SIAM J. Numer. Anal.*, 46(2):722–746, 2008.

[8] D.J. Bates, J.D. Hauenstein, A.J. Sommese, and C.W. Wampler. Bertini: Software for numerical algebraic geometry. Available at `http://www.nd.edu/∼sommese/bertini/`.

[9] C. Beltran and Leykin. A. Certified numerical homotopy tracking. Preprint `arXiv:0912.0920v1 [math.NA]`.

[10] R.H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[11] G. Björck and R. Fröberg. Methods to "divide out" certain solutions from systems of algebraic equations, applied to find all cyclic 8-roots. In M. Gyllenberg and L.E. Persson, editors, *Analysis, Algebra and Computers in Math. research*, volume 564 of *Lecture Notes in Mathematics*, pages 57–70. Dekker, 1994.

[12] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation.* Springer–Verlag, 1998.

[13] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada.* Cambridge University Press, 2007.

[14] G. Chèze and A. Galligo. Four lectures on polynomial absolute factorization. In *Solving Polynomial Equations. Foundations, Algorithms and Applications*, volume 14 of *Algorithms and Computation in Mathematics*, pages 339–394. Springer–Verlag, 2005.

[15] R.M. Corless, A. Galligo, I.S. Kotsireas, and S.M. Watt. A geometric-numeric algorithm for factoring

multivariate polynomials. In T. Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation (ISSAC 2002)*, pages 37–45. ACM, 2002.

[16] R.M. Corless, M.W. Giesbrecht, M. van Hoeij, I.S. Kotsireas, and S.M. Watt. Towards factoring bivariate approximate polynomials. In B. Mourrain, editor, *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation (ISSAC 2001)*, pages 85–92. ACM, 2001.

[17] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[18] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

[19] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[20] A. Galligo and A. Poteaux. Continuations and monodromy on random Riemann surfaces. In H. Kai and H. Sekigawa, editors, *SNC'09: Proceedings of the 2009 conference on Symbolic-Numeric Computation*, pages 115–124. ACM, 2009.

[21] A. Galligo and M. van Hoeij. Approximate bivariate factorization: a geometric viewpoint. In J. Verschelde and S.M. Watt, editors, *SNC'07: Proceedings of the 2007 international workshop on Symbolic-Numeric Computation*, pages 1–10. ACM, 2007.

[22] T. Gao, T. Y. Li, and M. Wu. Algorithm 846: MixedVol: a software package for mixed-volume computation. *ACM Trans. Math. Softw.*, 31(4):555–560, 2005.

[23] Y. Guan and J. Verschelde. Parallel implementation of a subsystem-by-subsystem solver. In *Proceedings of the 22th High Performance Computing Symposium, Quebec City, 9-11 June 2008*, pages 117–123. IEEE Computer Society, 2008.

[24] T. Gunji, S. Kim, K. Fujisawa, and M. Kojima. PHoMpara – parallel implementation of the Polyhedral Homotopy continuation Method for polynomial systems. *Computing*, 77(4):387–411, 2006.

[25] Y. Hida, X.S. Li, and D.H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), 11-17 June 2001, Vail, CO, USA*, pages 155–162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996, software at `http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.9.tar.gz`.

[26] B. Huber and B. Sturmfels. A polyhedral method for solving sparse polynomial systems. *Math. Comp.*, 64(212):1541–1555, 1995.

[27] M. Kojima. Efficient evaluation of polynomials and their partial derivatives in homotopy continuation methods. *Journal of the Operations Research Society of Japan*, 51(1):29–54, 2008.

[28] A Leykin and F. Sottile. Galois groups of Schubert problems via homotopy continuation. *Mathematics of Computation*, 78(267):1749–1765, 2009.

[29] A. Leykin and J. Verschelde. Factoring solution sets of polynomial systems in parallel. In T. Skeie and C.-S. Yang, editors, *Proceedings of the 2005 International Conference on Parallel Processing Workshops. 14-17 June 2005. Oslo, Norway. High Performance Scientific and Engineering Computing*, pages 173–180. IEEE Computer Society, 2005.

[30] A. Leykin and J. Verschelde. Decomposing solution sets of polynomial systems: a new parallel monodromy breakup algorithm. *The International Journal of Computational Science and Engineering*, 4(2):94–101, 2009.

[31] A. Leykin, J. Verschelde, and Y. Zhuang. Parallel homotopy algorithms to solve polynomial systems. In N. Takayama and A. Iglesias, editors, *Proceedings of ICMS 2006*, volume 4151 of *Lecture Notes in Computer Science*, pages 225–234. Springer-Verlag, 2006.

[32] T.Y. Li. Numerical solution of polynomial systems by homotopy continuation methods. In F. Cucker, editor, *Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics*, pages 209–304. North-Holland, 2003.

[33] T.Y. Li, T. Sauer, and J.A. Yorke. The cheater's homotopy: an efficient procedure for solving systems of polynomial equations. *SIAM J. Numer. Anal.*, 26(5):1241–1251, 1989.

[34] T.Y. Li and C.-H. Tsai. HOM4PS-2.0para: Parallelization of HOM4PS-2.0 for solving polynomial systems. *Parallel Computing*, 35(4):226–238, 2009.

[35] A. Morgan. *Solving polynomial systems using continuation for engineering and scientific problems*. Prentice-Hall, 1987. Volume 57 of Classics in Applied Mathematics Series, SIAM 2009.

[36] A.P. Morgan and A.J. Sommese. Coefficient-parameter polynomial continuation. *Appl. Math. Comput.*, 29(2):123–160, 1989. Errata: *Appl. Math. Comput.* 51:207(1992).

[37] A. Poteaux. Computing monodromy groups defined by plane curves. In J. Verschelde and S.M. Watt, editors, *SNC'07. Proceedings of the 2007 International Workshop on Symbolic-Numeric Computation*, pages 239–246. ACM, 2007.

[38] D.N. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

[39] J.F. Ruiz. GNAT Pro for on-board mission-critical space applications. In T. Vardanega and A. Wellings, editors, *Reliable Software Technology – Ada-Europe 2005. 10th Ada-Europe International Conference on Reliable Software Technologies. York, UK, June 20-24, 2005. Proceedings*, volume 3555 of *Lecture Notes in Computer Science*, pages 248–259, 2005.

[40] J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[41] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core*. Massachusetts Institute of Technology, second edition, 1998.

[42] A.J. Sommese, J. Verschelde, and C.W. Wampler. Using monodromy to decompose solution sets of polynomial systems into irreducible components. In

C. Ciliberto, F. Hirzebruch, R. Miranda, and M. Teicher, editors, *Application of Algebraic Geometry to Coding Theory, Physics and Computation*, pages 297–315. Kluwer Academic Publishers, 2001. Proceedings of a NATO Conference, February 25 - March 1, 2001, Eilat, Israel.

[43] A.J. Sommese and C.W. Wampler. *The Numerical solution of systems of polynomials arising in engineering and science*. World Scientific, 2005.

[44] H.-J. Su, J.M. McCarthy, M. Sosonkina, and L.T. Watson. Algorithm 857: POLSYS_GLP: A parallel general linear product homotopy code for solving polynomial systems of equations. *ACM Trans. Math. Softw.*, 32(4):561–579, 2006.

[45] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Softw.*, 25(2):251–276, 1999. Software available at `http://www.math.uic.edu/~jan/download.html`.

[46] J. Verschelde and Y. Wang. Computing feedback laws for linear systems with a parallel Pieri homotopy. In Y. Yang, editor, *Proceedings of the 2004 International Conference on Parallel Processing Workshops, 15-18 August 2004, Montreal, Quebec, Canada. High Performance Scientific and Engineering Computing*, pages 222–229. IEEE Computer Society, 2004.

[47] J. Verschelde and Y. Zhuang. Parallel implementation of the polyhedral homotopy method. In T.M. Pinkston and F. Ozguner, editors, *Proceedings of the 2006 International Conference on Parallel Processing Workshops. 14-18 Augustus 2006. Columbus, Ohio. High Performance Scientific and Engineering Computing*, pages 481–488. IEEE Computer Society, 2006.

[48] Y. Wang. *Computing Dynamic Output Feedback Laws with Pieri Homotopies on a Parallel Computer*. PhD thesis, University of Illinois at Chicago, 2005.

[49] B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.

[50] Y. Zhuang. *Parallel Implementation of Polyhedral Homotopy Methods*. PhD thesis, University of Illinois at Chicago, 2007.

Parallel computations in modular group algebras

A. Konovalov, S. Linton
Centre of Interdisciplinary Research
in Computational Algebra
School of Computer Science, University of St Andrews
{alexk,sal}@cs.st-and.ac.uk

ABSTRACT

We report about the parallelisation of the algorithm to compute the normalised unit group $V(\mathbb{F}_pG)$ of a modular group algebra $\mathbb{F}_pG$ of a finite p-group G over the field of p elements $\mathbb{F}_p$ in the computational algebra system GAP. We present its distributed memory implementation using the new remote procedure call framework based on the the Symbolic Computation Software Composability Protocol (SCSCP). Using it, we were able for for the first time to perform practical computations of $V(\mathbb{F}_pG)$ for groups of orders 2^9 and 3^6.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Miscellaneous

Keywords

Group algebra, unit group, OpenMath, remote procedure call, SCSCP

1. INTRODUCTION

GAP [18] is an open-source system for computational discrete algebra, with particular emphasis on Computational Group Theory. It provides a programming language, an extensive library of functions implementing algebraic algorithms written in the GAP language, as well as large data libraries of algebraic objects. The GAP language has rather unique object oriented features invented to model mathematical objects. In particular, GAP objects accumulate information about themselves during their lifetime and, as a consequence, may subsequently change their type. The method selection is dynamically polymorphic and depends on the current type of all arguments. This type system, described in details in [8], is central to the organisation of the GAP library which is the main part of GAP system and is implemented in the GAP language. Another part of the system – its kernel – is implemented in the C language. The kernel provides the GAP runtime environment and also a set of time-critical basic functions. In the current release of GAP both the kernel and the library are sequential and do not support parallelism. In this setup, GAP users may be interested in various options to parallelise their computations without changing the core system.

One of the attempts to provide an infrastructure for distributed parallel computations in GAP was made in the ParGAP package [10], which is based on the MPI standard, and includes a subset implementation of MPI. Another example of a purpose-build C application that was used to orchestrate multiple GAP instances to solve some specific problems of computational algebra was described in [23]. Finally, one can use various job submission systems to submit tasks e.g. on Condor pools or grids, and organise, if necessary, file-based communication between tasks, but while being completely suitable for some applications, this does not provide enough interactivity and insights into the coordination of the workload for our problem, so such technologies are beyond the scope of this paper.

We present another tool that enables distributed parallel computations in GAP, and has been developed in the EU FP6 project "SCIEnce – Symbolic Computation Infrastructure in Europe" (`www.symbolic-computation.org`) is a major 5-year project that brings together developers of computer algebra systems (CAS), experts in computational algebra, OpenMath, and parallel computations. One of the project's outcomes is the development of a common standard interface that may be used for combining CAS and any other compatible software. This interface is a lightweight XML-based remote procedure call protocol called *SCSCP* (*Symbolic Computation Software Composability Protocol*, [16]) in which both data and instructions are represented as OpenMath objects [26]. At the time of writing, SCSCP has been implemented in several computer algebra systems, including GAP, KANT, Macaulay2, Maple, MuPAD, TRIP (see [15, 17] for details) and has APIs to simplify adding SCSCP interface to more systems.

We implemented SCSCP support in GAP in the SCSCP package [21]. In the present paper, we would like to introduce the package and demonstrate how it can speed up a computation of certain giant algebraic structures from the theory of group rings. Our achievement makes feasible building a database of such objects to be used in a search for interesting examples or counterexamples.

We proceed as follows. First we give basic definitions necessary to explain the main algorithm in section 2.1. The reader may however wish to skip details from the theory of group rings and go straight to section 2.2 where we describe its implementation in the GAP package LAGUNA. To explain our motivation, section 2.2 provides some examples of research that can be carried out using the LAGUNA package. Then section 2.3 presents the UnitLib package providing a database of objects computed using the main algorithm. After that we will describe the SCSCP package and compare it with the ParGAP package in section 3.1, and then explain our approach to the parallelisation in sections 3.2 and 3.3. After analysing parallel performance in section 3.4, we conclude with final remarks. We assume that the reader who uses GAP in another areas will be able to concentrate less on group ring related details but still see how the package may be used for the own research. Moreover, the reader from a parallel community who does not use GAP may skip these details and find more interesting technical aspects on design imposed by the context of the problem and the performance analysis.

2. BACKGROUND

2.1 Group rings

The theory of group rings is a branch of algebra that arose as a meeting point of group theory and ring theory. Group rings are not only of a purely theoretical interest in algebra and several other fundamental branches in mathematics, but they also have applications in several scientific fields, such as quantum physics and coding theory. They were introduced by Frobenius and Schur at the beginning of the 20th century and initially were primarily used as tools for studying representations of finite groups, becoming objects of special systematic studies only in the last 40-50 years.

For an associative ring R with multiplicative identity element and a multiplicatively written group G, the group ring RG of the group G over the ring R is defined as the set of all finite formal sums

$$RG = \left\{ \sum_{g \in G} \lambda_g g \;\middle|\; \lambda_g \in R,\ \left|\{g \in G \mid \lambda_g \neq 0\}\right| < \infty \right\}$$

with respect to a component-wise defined addition, and multiplication induced by the rule $(\lambda g)(\mu h) = (\lambda\mu)(gh)$ for $\lambda, \mu \in R$ and $g, h \in G$ extended by the distributivity law on the whole of RG.

The set of all elements of RG which are invertible with respect to the multiplication forms a group, which is called the *unit group* of RG. It is denoted by $U(RG)$, and its subgroup

$$V(RG) = \left\{ \sum_{g \in G} \lambda_g g \in U(RG) \;\middle|\; \sum_{g \in G} \lambda_g = 1 \right\}$$

is called the normalised unit group, or the group of normalised units.

A group ring is called a *group algebra* if R is a field. Furthermore, if R is a field of characteristic $p > 0$ and G has an element of order p, then RG is called a *modular group algebra*. For a finite group G, the dimension of its group algebra RG over the field R equals to the order of G. For more details, we refer to some monographs on group rings, e.g. [2] or [27].

We are interested in a special case when $R = \mathbb{F}_p$ is a field of p elements, and G is a *finite p-group* (that is, the order of G is p^n for some $n > 0$). In this case, the order of its group algebra over the field of p elements is p^{p^n}, and the order of $V(\mathbb{F}_pG)$ is p^{p^n-1} (so it is a multiplicative p-group, which allows access to some special algorithms for this class of groups). Because of their huge orders, modular group algebras of finite p-groups are sources of very challenging examples. Remarkably, it is still possible to perform efficient computations in $V(\mathbb{F}_pG)$ provided it has an efficient representation that allows to use fast algorithms. Such presentation is called a *power-commutator presentation*. It consists of generators $y_1, \ldots, y_{|G|-1}$, relations for powers

$$y_i^p = (y_{i+1})^{\alpha_{i,i+1}} \cdots (y_{|G|-1})^{\alpha_{i,|G|-1}}$$

for $1 \leq i \leq |G| - 1$ and relations for commutators

$$(y_j, y_i) = (y_{j+1})^{\alpha_{j,i,j+1}} \cdots (y_{|G|-1})^{\alpha_{j,i,|G|-1}}$$

for $1 \leq i < j \leq |G| - 1$, where exponents $\alpha_{i,k}$ and $\alpha_{i,j,k}$ are elements of the set $\{0, \ldots, p-1\}$. We will say that the group is a *pc-group* if it is represented using the power-commutator presentation, and call the set of generators $y_1, \ldots, y_{|G|-1}$ the *polycyclic generating set* of G.

In our case $V(\mathbb{F}_pG)$ has a natural polycyclic generating set $1 + S$, where S is a *weighted basis* of the augmentation ideal of $\mathbb{F}_pG$. We will briefly describe it here, referring to [6] for complete details. First we recursively define the *Jennings series* of G

$$G = G_1, G_{i+1} = [G_i, G]G_{j^p}$$

where j has to be the smallest non-negative integer such that $j \geq i/p$. Then the *dimension basis* for G is the list

$$x_{1,1}, \ldots, x_{1,l_1}, \ldots, x_{k,1}, \ldots, x_{k,l_k}$$

of elements of the group G such that

$$\{x_{i,1}G_{i+1}, \ldots, x_{i,l_i}G_{i+1}\}$$

is a minimal generating set for the elementary abelian group G_i/G_{i+1}. Now the weighted basis consists of $|G|-1$ *standard products* of the form

$$(x_{1,1} - 1)^{\alpha_{1,1}} \cdots (x_{1,l_1} - 1)^{\alpha_{1,l_1}} \cdots (x_{k,1} - 1)^{\alpha_{k,1}} \cdots (x_{k,l_k} - 1)^{\alpha_{k,l_k}},$$

where $0 \leq \alpha_{i,j} \leq p - 1$ and at least one of $\alpha_{i,j}$ is non-zero.

For example, if $\mathbb{F}_2$ is the field of two elements and D_8 is the dihedral group of order 8, given by the presentation $D_8 = \langle\, a, b \mid a^4 = b^2 = 1, ab = ba^{-1} \,\rangle$, the dimension basis of D_8 is the set $\{a, b, a^2\}$, and the *weighted basis* of $\mathbb{F}_2D_8$ is the set

$$\{(a-1), (b-1), (a^2-1), (a-1)(b-1), (a-1)(a^2-1), (b-1)(a^2-1), (a-1)(b-1)(a^2-1)\}.$$

For a practical implementation of this algorithm in GAP, it is very fast to compute $1 + S$ as the set consisting of explicitly written elements of a group algebra. However, one then needs to compute the canonical form of every power and commutator relation with respect to this polycyclic generating set, and this is a very time-consuming step which requires extensive computations in a group algebra, whose complexity grows with the growth of the order of the underlying group. We refer to the LAGUNA manual [7] for further details.

2.2 LAGUNA package

The algorithm to compute $V(\mathbb{F}_pG)$ for a finite p-group G and a field of p elements $\mathbb{F}_p$ has been implemented by the first author and Cs. Schneider in the LAGUNA package [7] for the computational algebra system GAP since its first release in 2003. The title LAGUNA is an acronym for **L**ie **A**l**G**ebras and **UN**it groups of group **A**lgebras. The package was started by R. Rossmanith as the LAG package for GAP 3.4.4 to compute Lie algebras of group algebras, and then was taken over and extended (including unit groups functionality) by other authors. LAGUNA can compute $V(\mathbb{F}_pG)$ in two representations: *natural* (slow, operates with explicitly written group algebra elements), and *abstract* (fast, operates with elements of a pc-group). Using the bijection between these two representations, one can compute, for example, the centre of $V(\mathbb{F}_pG)$ given as a pc-group and then find corresponding group algebra elements.

To demonstrate a sample calculation with LAGUNA, we may check a counterexample reported in [9] to the conjecture that $V(KG)$ and a group constructed in a special way (namely, the *wreath product* $C_p \wr G'$) have the same value of the invariant called the *nilpotency class of the group*. First we retrieve from the GAP Small Groups Library the group `G` that we need, compute its derived subgroup and then construct the group `W` that is the desired wreath product. After checking that `W` has the nilpotency class 3, we create a modular group algebra of `G`, compute its normalised unit group `V` in the pc-presentation, and immediately verify that unlike `W`, the group `V` has the nilpotency class 4.

```
gap> G := SmallGroup( 32, 6 );
<pc group of size 32 with 5 generators>
gap> D := DerivedSubgroup(G);
Group([ f3, f5 ])
gap> W := WreathProduct( CyclicGroup(2), D );
<group of size 64 with 3 generators>
gap> NilpotencyClassOfGroup(W);
3
gap> FG := GroupRing( GF(2) , G );
<algebra-with-one over GF(2), with 5 generators>
gap> V := PcNormalizedUnitGroup( FG );
<pc group of size 2147483648 with 31 generators>
gap> NilpotencyClassOfGroup( V );
4
```

Note that without the computation of the pc-presentation of $V(\mathbb{F}_pG)$ GAP would be forced to operate with explicitly written group ring elements, and a computation as in the example would not be feasible at all.

Another exemplar application of LAGUNA is related with the Modular Isomorphism Problem (MIP) asking whether a finite p-group is determined by its modular group algebra over a field of p elements; in other words, is it true that

$$\mathbb{F}_pG \cong \mathbb{F}_pH \Rightarrow G \cong H \text{ ?}$$

This problem is still widely open in general (for an overview of the current status of MIP, see [13]), although there are counterexamples for its analogs for integral group rings and for modular group algebras for a wider class of groups [12, 19]. However, even less is known for its stronger variation – the Modular Isomorphism Problem of Normalised Unit Groups (UMIP). The latter problem is asking whether a finite p-group is determined by the normalised unit group of its modular group algebra over a field of p elements; in other words, is it true that

$$V(\mathbb{F}_pG) \cong V(\mathbb{F}_pH) \Rightarrow G \cong H \text{ ?}$$

A positive answer was obtained for p-groups with cyclic $\Phi(G)$ for $p > 2$ in [3] and for 2-groups of maximal class

Table 1: Sequential computation of $V(\mathbb{F}_pG)$

IdGroup	Structure description	Runtime
8,3	D_8	23 ms
16,7	D_{16}	135 ms
32,18	D_{32}	1.8 s
64,52	D_{64}	41 s
128,161	D_{128}	21 m 44 s
256,539	D_{256}	13 hr 8 m
27,3	$(C_3\text{x}C_3) : C_3$	0.8 s
81,8	$(C_9\text{x}C_3) : C_3$	1 m 36 s
243,26	$(C_9\text{x}C_9) : C_3$	5 hr 8 m

in [4]. Using the LAGUNA package, it was also verified that UMIP holds for all groups of order 32 (see [20]). Remarkably, for the hardest pair, consisting of groups with catalogue numbers (32,13) and (32,14), both automorphism groups $Aut(V(KG))$ had the same order 2^{149}, and only by using the AutPGrp package [14] was it determined that their minimal numbers of generators were 15 and 16 respectively, so they can not be isomorphic. Again, this computation became possible only due to operating with $V(KG)$ in pc-presentation.

The runtime to compute the pc-presentation of the normalised unit group may vary dependently on the structure of the group G, but certainly the main factor is the order of G which is equal to the dimension of the group algebra. We illustrate this dependency for some series of groups in the table 1 (the measurements were performed in GAP 4.4.12 on an 8-core Intel server: dual quad-core Intel Xeon 5570 2.93GHz / RAM 48 GB / CentOS Linux 5.3). In all tables in this paper, C_n denotes the cyclic group of order n, that is the group given by the presentation $\langle a \mid a^n = 1\rangle$, D_{2n} denotes the *dihedral group* of order $2n$, given by the presentation $\langle a, b \mid a^n = b^2 = 1, ab = ba^{-1}\rangle$, G x H denotes the direct product of G and H, and $G : H$ denotes the semi-direct product of G and H, following the convention used by the GAP function `StructureDescription`.

2.3 Library of unit groups

The significant time needed to compute $V(RG)$ for G of order 128 stimulated the first author to start the development of the GAP package UnitLib [22]. This package allows to save precomputed normalised unit groups for groups from the GAP Small Groups Library in a database. The current version of the package covers all p-groups of order not greater than 243, occupying 21.5 MB of local storage for all these groups except those of order 243 and additional 32 MB of online available data for groups order 243.

During the development of the package, many technical issues related with saving and retrieving the data were solved, including deciding to store `CodePcGroup` (large integer encoding the structure of the group) in hexa-decimal format; storing the dimension basis of the group to avoid further interfering with random methods used by GAP; gzipping large files and unzipping them "on fly"; providing online access to the data, proper embedding of G into the retrieved pc-group, etc. We refer to the UnitLib manual [22] for further technical details.

The package uses the same catalogue numbers for groups as the GAP Small Group Library. In the example below, we

Table 2: **Retrieving precomputed** $V(\mathbb{F}_pG)$

Group Id	Runtime (compute)	Runtime (retrieve)	Speedup	Data file (bytes)
8,3	23 ms	3 ms	6.7	78
16,7	135 ms	6 ms	33.3	225
32,18	1.8 s	26 ms	96.2	1518
64,52	41 s	273 ms	219.8	12958
128,161	21m 44 s	7 s	192.9	110737
256,539	13 hr 8 m	9 m 3 s	102.5	925019
27,3	0.8 s	13 ms	107.7	793
81,8	1 m 36 s	673 ms	237.7	27865
243,26	5 hr 8 m	4 m 15 s	89.4	895391

will retrieve a normalised unit group from the library, and then show that it has some interesting property (specifically, its sets of bicyclic units of 1st and 2nd kind generate different subgroups of $V(\mathbb{F}_pG)$):

```
gap> V:=PcNormalizedUnitGroupSmallGroup(64,9);
<pc group of size 9223372036854775808 with 63 generators>
gap> V1:=BicyclicUnitGroupType(V,1); #auxiliary function
<pc group with 154 generators>
gap> V2:=BicyclicUnitGroupType(V,2);
<pc group with 154 generators>
gap> Size(V1); Size(V2);
268435456
268435456
gap> Size(Intersection(V1,V2));Size(ClosureGroup(V1,V2));
134217728
536870912
```

In the table 2 we compare the runtime of retrieving $V(RG)$ from the library with the runtime require for its computation from scratch, for the same groups as in the table 1. Additionally, we display the size of the resulting data file that must be stored for the future usage. The measurements were performed in GAP 4.4.12 on an 8-core Dell Poweredge 2950 server: dual quad-core Intel Xeon 5355 2.66 GHz / RAM 16 GB / CentOS Linux 4.5 for the *current version* of the package as cited in [22]. As one can see from the table 2, using the UnitLib database allows dramatic time savings. However, in order to create this database, anyway first we need to compute each normalised unit group. Since with the increasing of the order of the group this becomes more and more time-consuming, parallelisation of this computation becomes more and more important to enable efficient computation of $V(KG)$, for example, for groups of orders like 512 or 729 which are also available in the GAP Small Groups Library. Clearly, we want to make use of the available GAP codebase and we need to produce an output compatible with GAP. The tools presented in the next session made such parallelisation cost-effective and feasible without programming in any other languages except GAP.

3. PARALLELISATION WITH SCSCP

3.1 SCSCP and ParGAP

The Symbolic Computation Software Composability Protocol (SCSCP) is a protocol by which a CAS may offer services to a variety of possible clients, for example, another CAS running on the same or remote system; another instance of the same CAS (in a parallel computing context or in case of an installation with a limited functionality); Grid middleware; visualisation tool to display mathematical objects, e.g. graphs or lattices of subgroups; a Web server which passes on the same services as Web services using SOAP/HTTP to another clients, etc. In this setup it is important to agree about common encoding for the mathematical objects, and the obvious choice here was OpenMath [26], a standard of representing mathematical objects according to their semantics. Though there may be a first impression that the usage of OpenMath may be limited to functionality/data types for which OpenMath representation is defined in an existing OpenMath Content Dictionary (CD), the SCSCP server may have private transient CDs, specific to the service and obtainable from the server on request. Furthermore, the user may define private CDs with more efficient representation of some objects (e.g. matrices over finite fields) or pass the data in some private formats encoded as `OMSTRING`, `OMBYTES` or `OMFOREIGN` elements.

In the GAP system, support for OpenMath and SCSCP is implemented in two GAP packages with the same names. The OpenMath package [11], development of which was taken over by the first author in 2007, is an OpenMath phrasebook for GAP: it converts OpenMath to GAP and vice versa, and provides a framework that users may extend with their private content dictionaries. The SCSCP package [21], started by the authors in 2007, provides a socket-based SCSCP implementation using the GAP packages OpenMath, IO [25] and GAPDoc [24]. This allows GAP to run as either an SCSCP server or client. The server may be started interactively from the GAP session or as a GAP daemon. When the server accepts a connection from the client, it starts the "accept-evaluate-return" loop:

- accepts the `"procedure_call"` message and looks up the appropriate GAP function (which should be declared by the service provider as an SCSCP procedure);
- evaluates the result (or produces a side-effect);
- replies with a `"procedure_completed"` message or returns an error in a `"procedure_terminated"` message.

The SCSCP client performs the following basic actions:

- establishes connection with server;
- sends the `"procedure_call"` message to the server;
- waits for its completion or checks it later;
- fetches the result from a `"procedure_completed"` message or enters the break loop in the case of a `"procedure_terminated"` message.

We have used this basic functionality to build a set of instructions for parallel computations using the SCSCP framework. This allows the user to send several procedure calls to multiple instances of the same of different computer algebra systems in parallel and then collect all results, or to pick up the first available result.

One of applications built on top of SCSCP is the master-worker skeleton, which is implemented purely in GAP. It allows to run parallel computations with one master distributing individual tasks to multiple (local or remote) workers. The pool of workers must be specified by the user as a list of their hostnames and ports and may vary from a set of workers on the same multi-core server to a variety of geographically distributed workers running in various architectures. The client (i.e. master, which orchestrates the computation) works in any operating system that is able to run GAP, and it may orchestrate both GAP based and non-GAP based SCSCP servers. It uses dynamic scheduling to allocate tasks to workers at the runtime and stays idle while waiting for the next available worker. The master-worker skeleton offers basic fault-tolerance for stateless services: if a server (i.e. worker) disappears, the job will be resubmitted to another available server (it is the user's responsibility to ensure that this will not break the consistency of the result). Furthermore, it allows new workers (from a previously declared pool of potential workers) to be added during the computation. It has flexible configuration options and produces parallel trace files that can be visualised using EdenTV [5]. In our experiments, this implementation of the master-worker skeleton demonstrated almost linear (e.g. 7.5 on 8-core machine) speedup on irregular applications with low task granularity and no nested parallelism.

These features make SCSCP different from the ParGAP package, since the latter is based on MPI, has both GAP and C parts of code and requires UNIX environment to work. The master-worker computation in ParGAP will be broken if one of workers is lost, and there is no way to add new workers in the middle of computation. Finally, ParGAP currently uses string representation of GAP objects to transmit them, and SCSCP package uses OpenMath representation. Both representations allow different fallbacks in order to transmit objects which do not have a meaningful string representation or are not supported in OpenMath. We refer to the SCSCP package manual [21] for further details and examples.

The next two sections serve as a tutorial in GAP code parallelisation with SCSCP and provide some arguments to support our model. Note that the SCIEnce project also offers other parallel tools whose suitability may depend on particular user scenarios. One of such tools is the SymGrid-Par [1], which also builds on SCSCP in an essential way. It is implemented using the functional programming language Haskell, and has many advanced features including work stealing and task migration. While SymGrid-Par requires additional configuration, the GAP package SCSCP offers a parallel toolkit for those users who want to stay within the GAP system and modify a minimum of their code.

3.2 Parallelising GAP code with SCSCP

Parallelising sequential GAP code with the SCSCP package, the user should make the following interconnected decisions: which functionality should be moved from the master to the remote procedures on the worker; which arguments these procedures will accept; which results these procedures must return. Among others, the following design goals should be taken into account: achieving the best possible workload distribution; efficient marshalling of arguments and results; and eliminating, if possible, dependencies across tasks.

For example, consider a parallel search in the GAP Small Groups Library (assume that it is included at every worker's GAP installation). Let `IsGoodGroup(n,k)` be a remote procedure that takes two integers `n` and `k`, tests whether certain property holds for the `SmallGroup(n,k)` and returns `true` or `false` dependently on the result. Let the goal is to find all groups, for which that property holds. Communications costs here are minimal, tasks are completely independent, and when the property to be tested is non-trivial enough to make workers busy, a call like

```
ParListWithSCSCP( List( [1..NrSmallGroups(128)],
                  i -> [128,i] ), "IsGoodGroup");
```

is destined to work well (of course, if the property is very simple to check, one may redesign the remote procedure to tests several groups at once with some chunk size giving a better performance).

In another situation there may be some global data that should be sent to every worker before the call of the master-worker skeleton. For example, different workers may work with different cosets or different conjugacy classes of the group. In this case the group may be distributed to every worker prior to the computation, and then a reference to it may be passed on each procedure call together with a coset or conjugacy class representative. Again, it is the GAP programmer's responsibility to ensure that the data are consistent; for example, it may happen that randomised algorithms will be applied to this group at each worker and e.g. the ordering of conjugacy classes may be different. This will not be a problem if an argument of the remote procedure will be a representative of the conjugacy class, but may be unsafe if the argument will be the number specifying the position of a particular conjugacy class in the list of all conjugacy classes of a group.

In a more complicated scenario, certain events may require an update of the global data required at every worker. This is also possible: observe that a worker, as an SCSCP server, may be able to perform a variety of procedures, so one may implement one or several procedures to update its data. Moreover, a remote procedure itself may involve procedure calls to other SCSCP servers outside the workers pool, for example to calls to dedicated servers to store distributed hash tables in the orbit computation.

One final warning should be given about the specific of sending GAP objects across the network in any format (which also explains why there is no straightforward way to use existing parallel libraries). For example, GAP objects may share their subobjects, and may learn information about themselves during their lifetime. In the first case, it may happen that the OpenMath representation will not express the sharing (though we resolved this problem for polynomials in GAP, there may be other examples). In the second case the default OpenMath representation for an object may not include all information about it that is known at the moment. This should be taken into account in the design and testing of SCSCP procedures, as well as the potentially high costs of data marshalling that may reduce speedup (transmitting only required information about an object may be more efficient than transmitting the whole object).

3.3 Unit group in parallel

Now let us return to our problem of computing the pc-presentation of the normalised unit group of a modular group algebra of a finite p-group over the field of p elements. Obviously, the computation of every single relation does not depend on other relations, and can be performed in parallel. We should expect that the computation may be quite irregular depending on the particular group and on the particular weighted basis elements. Furthermore, we had to decide about the data representation and delegation of work from master to workers to ensure that the global data on

Table 3: Parallel computation of $V(\mathbb{F}_p G)$

Group Id	Total runtime (sequential)	Final non-parallelisable step duration	Overall runtime and speedup (M+7LW)	Overall runtime and speedup (M+8LW)	Overall runtime and speedup (M+8LW+8RW)	Overall runtime and speedup (M+8LW+16RW)
8,3	23 ms	< 1 ms	440 ms / 0.05	414 ms / 0.06	335 ms / 0.07	292 ms / 0.79
16,7	135 ms	< 2 ms	1824 ms / 0.07	1703 ms / 0.08	1397 ms / 0.10	1445 ms / 0.09
32,18	1.8 s	< 5 ms	9.6 s / 0.19	9.2 s / 0.20	8.7 s / 0.21	8.3 ms / 0.22
64,52	41 s	0.3 s	43 s / 0.95	39 s / 1.05	35 s / 1.17	35 s / 1.17
128,161	21 m 44 s	1 s	5 m 25 s / 4.01	4 m 55 s / 4.42	3 m 9 s /6.90	2 m 40 s / 8.15
256,539	13 hr 8 m	2 m 57 s	2 hr 17 m / 5.75	2 hr 2 m / 6.43	1 hr 4 m / 12.31	45 m / 17.51
27,3	0.8 s	< 3 ms	6.6 s / 0.12	6.6 s / 0.12	5.7 s / 0.14	5.8 s / 0.14
81,8	1 m 36 s	0.3 s	1 m 9 s / 1.39	1 m 6 s / 1.45	55 s / 1.75	54 s / 1.78
243,26	5 hr 8 m	6 s	58 m 7 s / 5.31	52 m 3 s / 5.92	27 m 23 s / 11.25	19 m 31 s / 15.92

the master and each worker are consistent and the intervals when workers may be idle are reduced to a minimum.

First of all, receiving request to compute power or commutator relation, every worker must compute it in the group algebra of the same group, which should be distributed to workers prior to the computation. Despite SCSCP allowing us to do this using *remote objects*, the specific of our application (providing the library of normalised unit groups for groups from the GAP Small Groups Library) allows another solution that reduces communication costs avoiding the necessity of sending group algebra over the network.

Since every master is supposed to have the GAP Small Groups Library as a part of its GAP installation, the group algebra for the required group may be recreated independently on each worker. Since the underlying group will be retrieved from the library, this guarantees that all these group algebras will be completely identical. The worker is keeping a pool of group algebras stored as global variables, and the catalogue number of the group is passed as an additional argument in each procedure call to ensure that the computation is performed in the right object.

Moreover, each worker will compute the same weighted basis `WB` for the group algebra (since this is completely deterministic procedure which does not call randomised algorithms). Thus, we need to design two remote procedures, one to compute the power relation for `1+WB[i]` and another compute the commutator relation for `1+WB[i]` and `1+WB[j]`. Their arguments are {`i`} or {`i,j`} respectively, and also the catalogue number of the underlying group.

The result returned by these procedures is the string representing the product of powers of elements of the polycyclic generating set of the group, e.g `F1^2*F2^-1*F3*F1^-2*F2`. Such string representation is more compact that corresponding OpenMath code, and it can be straightforwardly evaluated on the master as an element of the appropriate group constituting a part of the corresponding relation; this shows the flexible setup of SCSCP allowing private data formats to be embedded in SCSCP messages.

Inevitably, at the end of the algorithm there is sequential phase which is needed to combine all collected data into a pc-presentation and create the resulting pc-group. This phase may take quite substantial time for groups of orders larger or equal 2^9 or 3^6.

To check that the parallel implementation works correctly, we computed normalised unit groups for groups of order 8. For such a group, $V(KG)$ has order 128, so we can check that groups computed sequentially and in parallel are isomorphic by computing their catalogue number in the GAP Small Groups Library.

The sequential implementation of the algorithm by the first author and Cs. Schneider is contained in the LAGUNA package in the file `laguna/lib/laguna.gi`. Its parallel version developed by the first author is contained in the UnitLib package in the file `unitlib/lib/parunits.g`. It contains two remote procedures called `NormalizedUnitCFpower` and `NormalizedUnitCFcommutator` to compute the power and the commutator relation respectively, and the main function `ParPcNormalizedUnitGroup` with two calls to the master-worker skeleton in lines

```
rels1:=ParListWithSCSCP(
         listargs, "NormalizedUnitCFpower" );
...
rels2:=ParListWithSCSCP(
         listargs, "NormalizedUnitCFcommutator" );
```

Modification of the sequential code required identifying parts of the code that should be moved to the remote procedure, rearranging loops in the main function to create a list of arguments for `ParListWithSCSCP`, and adding a short code to convert the results of procedure calls into relations of the pc-group.

3.4 Analysing performance

In table 3 we compare the wall clock time needed to compute the normalised unit group in sequential and parallel implementations for the same groups as we used in the tables above. We used a cluster of three 8-core Intel servers (dual quad-core Intel Xeon 5570 2.93GHz / RAM 48 GB / CentOS Linux 5.3) and run parallel computations in four different combinations: the master and seven workers running on the same server (M+7LW); the master and eight workers running on the same server (M+8LW); the master, eight workers running on the same server and eight worker running on another server (M+8LW+8RW); and, finally, the master, eight workers running on the same server and sixteen workers running on two other servers (M+8LW+16RW).

Not surprisingly, for small groups parallelisation does not help at all since the cost of data marshalling and coordinating workers is too high. However, we are not concerned about this, since the time required for the sequential computation is very small and we do not need to repeat it multiple times in the context of our problem. Furthermore, starting from order 64 (in this particular example for groups of this type), the parallel version becomes faster than sequential.

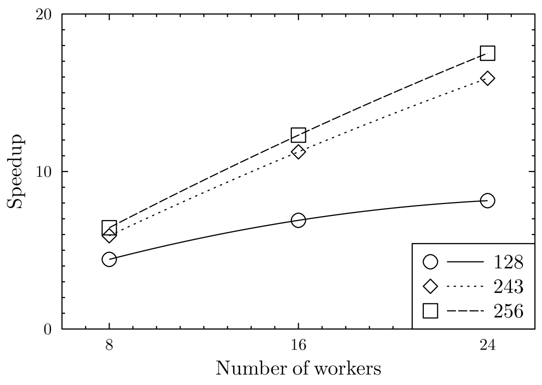


Figure 1: Scalability of the parallel computation of $V(\mathbb{F}_pG)$ for some groups of orders 128, 243 and 256

The best speedup equal to 17.51 was observed for the dihedral group of order 256 with 24 workers. Clearly, with the growth of the group order the dimension of the group algebra (which is the order of G) increases, so increases the complexity of the computation of an individual power or commutator relation. This forces workers to spend more time in the job and helps to optimise the load balancing and improve the performance.

Comparison of M+7LW and M+8LW configurations shows that it is more efficient to run one worker per each core and a master rather than to run a master on a dedicated core by the cost of having one less worker. This is explained by the fact that the master is idle while it waits for the next available worker using `select`, and this allows workers to make the maximum use of the available CPU power.

Figure 1 shows the dependency between the number of workers and the speedup for orders 128, 243 and 256.

Adding new workers does not speed up very much the computation for order 128. The scheduler tries first to allocate jobs to the workers residing locally and having lower latency, so even when the pool of workers is increased, remote workers do not get enough jobs since their competitors running on the same server with the master will be discovered and given a task earlier. However, the longer average runtime of individual jobs for orders 243 and 256 allows the master to keep all workers in the cluster sufficiently busy. Good scalability for groups of larger orders allows optimistic predictions that computations for targeted groups of orders 512 and 729 may scale well on the same cluster and even on a bigger heterogeneous network.

To further investigate the performance, below we demonstrate trace diagrams produced with the help of the EdenTV utility [5] to visualise the calculation of the normalised unit group for the dihedral group of order 128 (we changed colours from the original EdenTV output to fit a black-and-white printer). The first diagram shows the whole computation, the next two – its initial and final stages respectively.

The top row on Figure 2 corresponds to the master, and next 8 rows correspond to workers. The green colour marks intervals when the master or worker is performing some computation, while the blue colour corresponds to the waiting time, when master is waiting for the next response about the

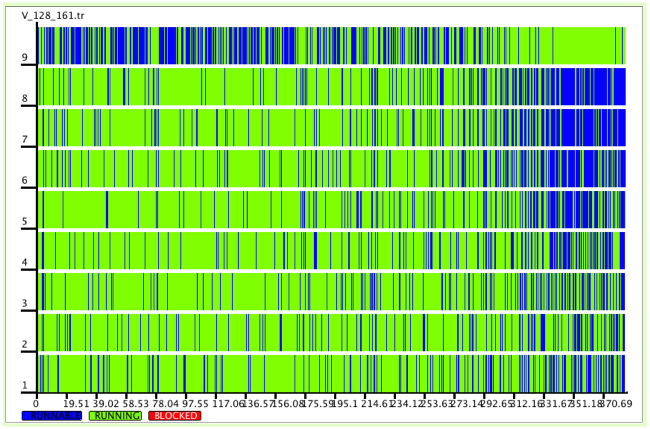


Figure 2: Parallel computation of $V(\mathbb{F}_2D_{128})$

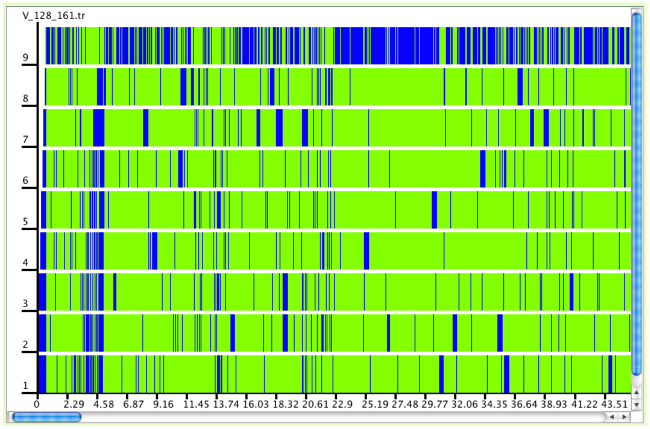


Figure 3: Initial stage of the computation of $V(\mathbb{F}_2D_{128})$

task completion or the worker is waiting for the next procedure call. Ideally, the master should be mostly blue, while workers should be mostly green; however, this is not always the case because of the irregularity and the final sequential phase.

Figure 3 shows a more detailed view of the initial stage of the computation. It is easy to detect the moment when the computation of the power relation was finished and the computation of the commutator relations was started (after 4.58 seconds). After that in the beginning workers are kept busy sufficiently well and their colour is mostly green.

Figure 4 shows more detailed view of the final stage of the computation. As the algorithm goes down the powers of augmentation ideal, individual tasks become shorter since the latter is nilpotent and many products evaluate to zero. This is why the load on the master increases, while some workers are being idle more often. At the end, the final sequential phase can be clearly seen, when the master uses all data collected from workers to create the resulting pc-group.

The similar situation could be also seen from the trace diagram for the computation of $V(\mathbb{F}_3G)$ in the configuration M+8LW+16RW for G=`SmallGroup(243,26)`. In the beginning, the load distribution is equal and workers are idle only for short intervals. At the end, the majority of tasks goes

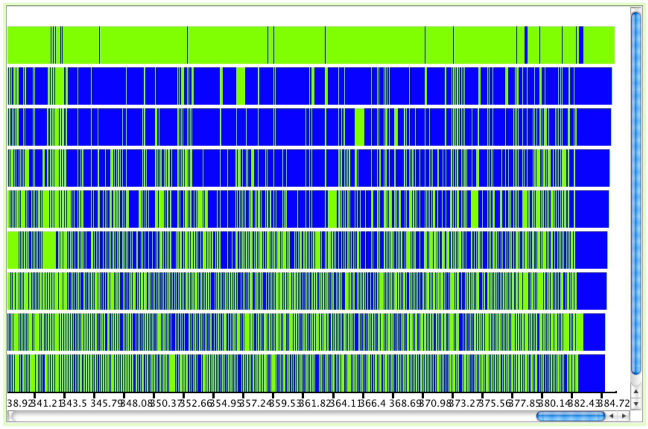


Figure 4: Final stage of the computation of $V(\mathbb{F}_2 D_{128})$

to the workers in the bottom part of the diagram, where workers 1–8 are residing on the same server with the master.

4. CONCLUSIONS AND FURTHER WORK

Using the parallel implementation of the algorithm to compute $V(\mathbb{F}_p G)$, for the first time we were able to deal with groups of such large orders as 512 and 729. For example, for G = SmallGroup(729,13) = $(C_3 \text{ x } (C_{27} : C_3)) : C_3$ it takes about 18.5 hours to compute $V(\mathbb{F}_p G)$ with one master and 24 workers on a cluster of three Intel servers (dual quad-core Intel Xeon 5570 2.93GHz/RAM 48 GB/CentOS Linux 5.3).

Our nearest goal is to complete the library for all 504 groups of order 729, and part of it is already computed. The runtime varies irregularly for different groups and we do not provide more precise measurements in this paper since it could be speeded up in the coming soon GAP 4.5 release with GMP support due to its faster integer arithmetic and, as a consequence, faster work with `CodePcGroup` (for larger groups, it might be faster/shorter to store the presentation itself rather then `CodePcGroup` that was originally designed to encode smaller groups and not those "monsters" as we generate).

In the future, with larger computational facilities to come, we may anticipate that it will be feasible to complete the library for the order 256 (56092 groups) and 5^5 (77 groups), probably even for 5^6 (684 groups). Doing this for groups of order 2^9 (10494213 groups) could be still a challenge, but even now the crucial fact is that using the provided functionality the user can compute such groups and save their description for further usage.

This computation may be revisited in the future with the outcome of the HPC-GAP project (`http://www-circa.mcs.st-and.ac.uk/hpcgap.php`) which will reengineer the GAP system to provide opportunities to use shared and distributed memory parallel programming models, and hopefully can be reimplemented with better scalability/efficiency.

We hope that GAP users from various domains will find this paper helpful in parallelising their GAP applications using SCSCP. We also hope that the feasibility of computing larger unit groups and making their database will lead to new theoretical insights into modular group algebras.

5. ACKNOWLEDGMENTS

This work has been supported by the EU FP6 project 26133 "SCIEnce – Symbolic Computation Infrastructure for Europe" (`http://www.symbolic-computation.org`).

6. REFERENCES

[1] A. Al Zain, P. Trinder, K. Hammond, A. Konovalov, S. Linton, and J. Berthold. Parallelism without pain: Orchestrating computational algebra components into a high-performance parallel system. In *Proc. IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA 2008), Sydney, Australia*, pages 99–112, 2008.

[2] V. A. Artamonov and A. A. Bovdi. Integral group rings: groups of invertible elements and classical K-theory. In *Algebra. Topology. Geometry, Vol. 27 (Russian)*, Itogi Nauki i Tekhniki, pages 3–43, 232. Akad. Nauk SSSR Vsesoyuz. Inst. Nauchn. i Tekhn. Inform., Moscow, 1989. Translated in J. Soviet Math. **57** (1991), no. 2, 2931–2958.

[3] Z. Balogh and A. Bovdi. Group algebras with unit group of class p. *Publ. Math. Debrecen*, 65(3-4):261–268, 2004.

[4] Z. Balogh and A. Bovdi. On units of group algebras of 2-groups of maximal class. *Comm. Algebra*, 32(8):3227–3245, 2004.

[5] J. Berthold and R. Loogen. Visualizing parallel functional program runs: Case studies with the eden trace viewer. In *Proc. PARCO 2007: Intl. Conf. on Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 121–128. IOS Press, 2007.

[6] A. Bovdi. Generators of the units of the modular group algebra of a finite p-group. In *Methods in ring theory (Levico Terme, 1997)*, volume 198 of *Lecture Notes in Pure and Appl. Math.*, pages 49–62. Dekker, New York, 1998.

[7] V. Bovdi, A. Konovalov, R. Rossmanith, and C. Schneider. *LAGUNA – Lie AlGebras and UNits of group Algebras, Version 3.5.0*, 2009. `http://www.cs.st-andrews.ac.uk/~alexk/laguna.htm`.

[8] T. Breuer and S. Linton. The GAP 4 type system: organising algebraic algorithms. In *ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, pages 38–45, New York, NY, USA, 1998. ACM.

[9] D. B. Coleman. Computer investigations of group algebras. In *Infinite groups and group rings (Tuscaloosa, AL, 1992)*, volume 1 of *Ser. Algebra*, pages 7–12. World Sci. Publ., River Edge, NJ, 1993.

[10] G. Cooperman. *ParGAP – Parallel GAP, Version 1.1.2*, 2004. GAP package, `http://www.ccs.neu.edu/home/gene/pargap.html`.

[11] M. Costantini, A. Konovalov, and A. Solomon. *OpenMath – OpenMath functionality in GAP, Version 10.1*, 2010. GAP package, `http://www.cs.st-andrews.ac.uk/~alexk/openmath.htm`.

[12] E. C. Dade. Deux groupes finis distincts ayant la même algèbre de groupe sur tout corps. *Math. Z.*, 119:345–348, 1971.

[13] B. Eick and A. Konovalov. The modular isomorphism problem for the groups of order 512. In *Groups St.*

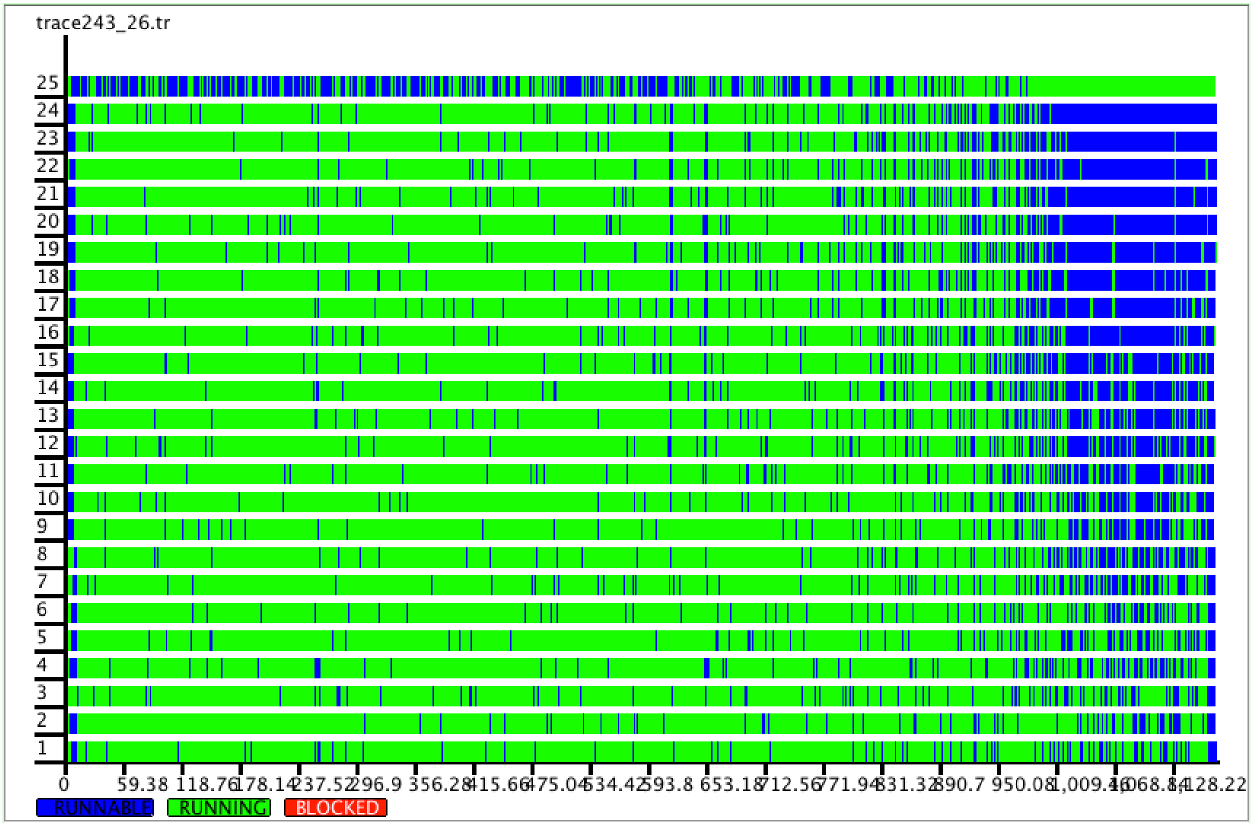


Figure 5: Parallel computation of $V(\mathbb{F}_3 G)$ for G=`SmallGroup(243,26)`

Andrews 2009, London Math. Soc. Lecture Note Ser. (Accepted).

[14] B. Eick and E. O'Brien. *AutPGrp – Computing the Automorphism Group of a p-Group, Version 1.4*, 2009. `http://www-public.tu-bs.de:8080/~beick/so.html`.

[15] S. Freundt, P. Horn, A. Konovalov, S. Lesseni, S. Linton, and D. Roozemond. OpenMath in SCIEnce: Evolving of symbolic computation interaction. In proceedings of OpenMath Workshop 2009 (to appear).

[16] S. Freundt, P. Horn, A. Konovalov, S. Linton, and D. Roozemond. Symbolic Computation Software Composability Protocol (SCSCP) specification. `http://www.symbolic-computation.org/scscp`, Version 1.3, 2009.

[17] S. Freundt, P. Horn, A. Konovalov, S. Linton, and D. Roozemond. Symbolic computation software composability. In *AISC/MKM/Calculemus, Springer LNCS 5144*, pages 285–295, 2008.

[18] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008. `http://www.gap-system.org`.

[19] M. Hertweck. A counterexample to the isomorphism problem for integral group rings. *Ann. of Math. (2)*, 154(1):115–138, 2001.

[20] A. Konovalov and A. Krivokhata. On the isomorphism problem for unit groups of modular group algebras. *Acta Sci. Math. (Szeged)*, 73(1-2):53–59, 2007.

[21] A. Konovalov and S. Linton. *SCSCP – Symbolic Computation Software Composability Protocol, Version 1.2*, 2010. GAP package, `http://www.cs.st-andrews.ac.uk/~alexk/scscp.htm`.

[22] A. Konovalov and E. Yakimenko. *LAGUNA – Library of normalized unit groups of modular group algebras, Version 3.0*, 2009. `http://www.cs.st-andrews.ac.uk/~alexk/unitlib.htm`.

[23] F. Lübeck and M. Neunhöffer. Enumerating large orbits and direct condensation. *Experiment. Math.*, 10(2):197–205, 2001.

[24] F. Lübeck and M. Neunhöffer. *GAPDoc – A Meta Package for GAP Documentation*, 2008. `http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc`.

[25] M. Neunhöffer. *IO – Bindings for low level C library IO*, 2009. `http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/io.html`.

[26] The OpenMath Society. *The OpenMath Standard, Version 2.0*, 2004. `http://www.openmath.org`.

[27] C. Polcino Milies and S. K. Sehgal. *An introduction to group rings*, volume 1 of *Algebras and Applications*. Kluwer Academic Publishers, Dordrecht, 2002.

Cache-Oblivious Polygon Indecomposability Testing

Fatima K. Abu Salem[*]
Computer Science Department, American University of Beirut
P.O. Box 11-0236, Riad El Solh Beirut 1107 2020
Lebanon
fatima.abusalem@aub.edu.lb

ABSTRACT

We examine a cache-oblivious reformulation of the (iterative) polygon indecomposability test of [19]. We analyse the cache complexity of the iterative version of this test within the ideal-cache model and identify the bottlenecks affecting its memory performance. Our analysis reveals that the iterative algorithm does not address data locality and that memory accesses progress with arbitrarily sized jumps in the address space. We reformulate the iterative computations of [19] according to a DFS traversal of the computation tree and obtain, as a result, a cache-oblivious variant which exhibits asymptotically improved spatial and temporal locality over the original one. In particular, we show that the DFS variant ensures spatial locality, and describe quantitatively the asymptotic improvements in spatial and temporal locality. In an extension to this work appearing in [3], the DFS variant is implemented in relation to absolute irreducibility of bivariate polynomials over arbitrary fields, and tested against both the original version as given in [19] and the powerful computer algebra system MAGMA. The results demonstrate significantly improved performance for the DFS variant as indicated by L1 misses, L2 misses, and total execution time.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; G.4 [**Mathematical Software**]: algorithm design and analysis; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Analysis of Algorithms*

General Terms

Algorithms

Keywords

Cache-oblivious algorithms, computer algebra, symbolic computing, Newton polygons, Convex polygon indecomposability testing, bivariate polynomials, absolute irreducibility testing

[*]The author is supported by LNCSR grant 111135-522312 and AUB-URB grant 111135-988119.

1. INTRODUCTION

With actual computers containing an ever-growing memory hierarchy, it is becoming increasingly important to aim for more than reducing the asymptotic operational count of algorithms, by also addressing their scalability, in the sense of efficient processor power utilisation. Even for algorithms which are not a "bottleneck" in practice, scalability in this context may be adversely affected if data transfer consumes a large component of the overall running time. *Caching* serves to optimise on data access across the different levels of the memory hierarchy in order to minimise the effect of I/O latency, the latter amplified by the growing gap between processor speed and that of an external memory access.

Two main paradigms for caching are the external (cache-aware) model, and the ideal-cache (cache-oblivious) model. The cache-aware model requires tuning cache parameters, such as the cache size and the cache-line length. In contrast, the cache-oblivious model does not require knowledge of, and hence tuning the algorithm according to, the cache parameters. Cache-oblivious programs thus allow for resource usage not to be programmed explicitly, and for algorithms to be portable across varying architectures, making such a paradigm "cheaper" from a software engineering viewpoint.

Cache-oblivious data structures have been extensively developed in [6, 7, 9, 10, 11, 12, 13, 21], to name a few. There is also considerable work on cache-oblivious algorithms in scientific computing, such as for matrix multiplication (e.g. [8, 15]), linear algebra and numerical solvers (e.g. [18, 22, 23]), and the FFT (e.g. [16, 17]), amongst many others. Yet, this work has little parallel in computer algebra, even though the memory hierarchy effects on scalability here are just as compelling.

In this context, we address the algorithm of [19] for integral polygon indecomposability testing, with consequences relating to the following:

1. Absolute irreducibility testing of bivariate polynomials with coefficients from an arbitrary field: This forms an important part in computer algebra systems today with applications in algebra, geometry, and number theory. It is also largely regarded as complementary to polynomial factorisation, in that it is essential to invest in an efficient test for irreducibility before resorting to the usually more expensive factorisation kernels.

2. Recovery of summands of decomposable integral poly-

gons in the sense of the Minkowski sum, which can be achieved via a simple generalisation of the indecomposability testing algorithm of [19]: The summands obtained comprise the input to the bivariate polynomial factorisation algorithm of [2], one of the fastest methods available that are tailored for sparse polynomials ([1]). Algebraic simplification, proving combinatorial identities, and solving systems of polynomial equations using Gröbner bases are applications which benefit from bivariate polynomial factorisation ([14]).

The contributions of this paper are constrained to theoretical cache complexity analysis. In particular, we analyse the cache complexity of the iterative test of [19] as well as its recursive reformulation obtained via a DFS traversal of the computation tree. We show that the DFS variant ensures spatial locality in the sense that it totally avoids jumps in the access to all data structures employed. Our analysis also determines the cache complexity of the DFS variant given by Eq. (15), and we derive that this is an asymptotic improvement under specific conditions affecting the input size, given in Eq. (21). In this paper we do not report on empirical results, which instead are reported in the manuscript [3]. In contrast, the latter manuscript does not provide theoretical cache complexity analysis, and is exclusively dedicated to empirical performance assessment using execution runtime metrics. The present paper and [3] complement each other in the study of cache oblivious polygon indecomposability testing.

2. BACKGROUND

2.1 Caching and the ideal-cache model [17]

We assume the reader is familiar with the notions of memory hierarchies, caches, cache misses and hits, and memory transfer delays. The cache is organised using cache lines, each consisting of B consecutive words. All words in a single line are transferred together between cache and main memory in one round, an operation referred to as a block transfer, memory transfer, or an I/O operation. If the cache is full and a cache miss occurs, a block gets evicted to make space. To assess the cost of I/O's, we measure two types of locality. Spatial locality refers to code being able to reference data items that are close to recently accessed ones. Temporal locality refers to code being able to re-use recently accessed data. The cache complexity is an asymptotic measure of the growth of temporal locality, and it captures the total number of I/O's taking place between any two consecutive levels of the memory hierarchy.

Several models exist for assessing the cache complexity of algorithms. Some of the simpler models (e.g. [5, 17]) reason about a two-level memory hierarchy: an internal (data) cache consisting of M computer words, connected to an external main memory of indefinite size. Here, the cache complexity is expressed as a function of the cache size M, the cache line length B, and the input size N. The ideal-cache model of [17] is a two-level memory model suitable for *cache-oblivious* algorithms. Here both cache parameters M and B need not be known in order to achieve an optimal memory performance. Thanks to the model's attributes, programmers can achieve a certain level of portability and automated optimisation. The two-level ideal cache is fully associative, is *tall* (or that $M = \Omega(B^2)$), operates under an optimal replacement strategy, and operates under an automatic replacement strategy. These assumptions can all be simulated in real at no asymptotic overhead ([17]). Unlike the external (also two-level) memory model of [5], the ideal-cache model measures throughput using the work complexity and the cache complexity, rather than cache complexity alone. Cache-oblivious algorithms thus achieve optimal cache performance using also an optimal amount of work.

2.2 Polytope indecomposability testing

For an extensive review on the theory of convex polytopes we refer the reader to [20]. Given two polytopes Q and R, their *Minkowski sum* is defined to be the set $Q + R := \{q + r \mid q \in Q, r \in R\}$. The Minkowski sum of two convex polytopes is also a convex polytope. A polytope with all vertices pairs of integers is called an integral polytope. When Q and R are integral polytopes, so is their Minkowski sum. If an integral polytope P can be expressed as a Minkowski sum $Q+R$ for some integral polytopes Q and R, then we call this expression an *integral decomposition*. The decomposition is *trivial* if Q or R has only one point, and P is *integrally decomposable* if it has at least one non-trivial decomposition; else, it is *integrally indecomposable*. A polytope of dimension 2 is a *polygon*, where the only proper faces are edges and vertices.

Higher dimensional integral polytope indecomposability testing can be achieved probabilistically by performing a number of deterministic polygon indecomposability testings, as described in [19]. We thus restrict our attention to integral convex polygons. Consider an integral convex polygon P with m vertices in $\mathbb{Z}^2$, and vertices $v_0, ..., v_{m-1}$ ordered cyclically in a counter-clockwise direction around a chosen pivot v_0. Express the edges of P as vectors of the form $E_i = v_{(i+1) \bmod m} - v_i = (a_i, b_i)$, for $0 \le i < m$, where $a_i, b_i \in \mathbb{Z}$. A vector $v = (a, b) \in \mathbb{Z}^2$ is called a primitive vector if $\gcd(a, b) = 1$. If $n_i = \gcd(a_i, b_i)$ and $e_i = (a_i/n_i, b_i/n_i)$, then $E_i = n_i e_i$, where e_i is a primitive vector, for $0 \le i < m$. Each edge E_i contains $n_i + 1$ integral points including its endpoints, and so we can think of n_i as representing the "length" of E_i. The sequence of vectors $\{n_i e_i\}_{0 \le i < m}$ is called the edge sequence of P and uniquely identifies the polygon up to translation determined by v_0. Since the boundary of a polygon forms a closed path, we have that $\sum_{0 \le i < m} n_i e_i = (0, 0)$. For convenience, an edge sequence can be identified with that obtained by extending the sequence by inserting an arbitrary number of zero vectors. Thus, we can assume that the edge sequence of a summand of P has the same length as that of P. The algorithm below depends on the fact that, if a non-trivial summand of P were to exist, it would possess an edge sequence of the form $\sum_{0 \le i < m} k_i e_i$, where $0 \le k_i \le n_i$, $k_i \ne 0$ for at least one i, and $k_i \ne n_i$ for at least one i. The algorithm attempts to trace such summands as follows:

ALGORITHM 1. *[19]*

Input: The edge sequence $\{n_i e_i\}_{0 \le i < m}$ of an integral convex polygon P starting at a vertex v_0, and $e_i \in \mathbb{Z}^2$ primitive vectors.
Output: Whether P is decomposable.
Step 1: Compute the set of all the integral points in P. Make use of this set to answer queries on whether arbitrary points in the plane belong to P. Set $A_{-1} = \emptyset$.

Step 2: For $i = 0, ..., m-2$, compute the set of points in P that are reachable via the vectors $e_0, ..., e_i$:

2.1: For each $k = 1, ..., n_i$, if $v_0 + ke_i \in P$, then add it to A_i.

2.2: For each $u \in A_{i-1}$ and $k = 0, ..., n_i$, if $u + ke_i \in P$, then add it to A_i.

Step 3: Compute the last set A_{m-1} as follows. For each $u \in A_{m-2}$ and $k = 0, ..., n_{m-1} - 1$, if $u + ke_{m-1} \in P$ and $u + ke_{m-1}$ is not already in A_{m-1}, then add it to A_{m-1}.

Step 4: Return "Decomposable" if $v_0 \in A_{m-1}$, and "Indecomposable" otherwise.

THEOREM 1. *[19] Algorithm 1 decides decomposability correctly in $\mathcal{O}(tmN)$ operations on two dimensional vectors with integer coordinates. Here, t is the number of integral points in P, m is the number of its edges, and $N = \max\{n_i\}_{0 \le i < m}$.*

2.3 Example

Consider the polygon P consisting of the triangle $\mathcal{T}$ with vertices $v_0 = (0,0)$, $v_1 = (2,0)$, and $v_2 = (2,2)$. The edges of $\mathcal{T}$ can be expressed using the vectors

$E_0 = 2\,(1 \quad 0)^{\mathbf{T}}$, $E_1 = 2\,(0 \quad 1)^{\mathbf{T}}$, and $E_2 = 2\,(-1 \quad -1)^{\mathbf{T}}$.

By Alg. 1, the following sequence of vector operations is issued:

Iteration 1 Compute $\alpha_0 = v_0 + e_0 = (1,0) \in \mathcal{T} \Rightarrow$ store α_0. Compute $\alpha_1 = v_0 + 2e_0 = (2,0) \in \mathcal{T} \Rightarrow$ store α_1.

Iteration 2 Compute $\beta_0 = v_0 + e_1 = (0,1) \notin \mathcal{T} \Rightarrow$ discard β_0. Compute $\beta_1 = v_0 + 2e_1 = (0,2) \notin \mathcal{T} \Rightarrow$ discard β_1. Store α_0 and α_1. Compute $\beta_2 = \alpha_0 + e_1 = (1,1) \in \mathcal{T} \Rightarrow$ store β_2. Compute $\beta_3 = \alpha_0 + 2e_1 = (1,2) \notin \mathcal{T} \Rightarrow$ discard β_3. Compute $\beta_4 = \alpha_1 + e_1 = (2,1) \in \mathcal{T} \Rightarrow$ store β_4. Compute $\beta_5 = \alpha_1 + 2e_1 = (2,2) \in \mathcal{T} \Rightarrow$ store β_5.

Iteration 3 Store α_0, α_1, β_2, β_4, and β_5. Compute $\gamma_0 = \alpha_0 + e_2 = (0,-1) \notin \mathcal{T} \Rightarrow$ discard γ_0. Compute $\gamma_1 = \alpha_1 + e_2 = (1,-1) \notin \mathcal{T} \Rightarrow$ discard γ_1. Compute $\gamma_2 = \beta_2 + e_2 = (0,0) \in \mathcal{T} \Rightarrow$ store γ_2. Compute $\gamma_3 = \beta_4 + e_2 = (1,0) \in \mathcal{T} \Rightarrow$ store γ_3. Compute $\gamma_4 = \beta_5 + e_2 = (1,1) \in \mathcal{T} \Rightarrow$ store γ_4.

Finally, it turns out $v_0 = (0,0)$ is in A_2, from which one concludes that $\mathcal{T}$ is decomposable.

3. MEMORY PERFORMANCE ANALYSIS

Theorem 1 makes implicit the following assumptions:

ASSUMPTION 1. *Given an arbitrary point in the plane, it is assumed that the query whether or not this point belongs to P can be answered in constant time.*

ASSUMPTION 2. *Redundancies in vector computations arising in any one iteration of Alg. 1 are assumed to be eliminated before the start of the next iteration. It is also assumed that one can eliminate each redundant vector computation in constant time.*

Refer to [4] for a discussion on how these assumptions can be simulated for free in a strictly computational model such as the RAM model. This is no longer the case, however, if one were to address the memory access costs associated with testing for inclusion and eliminating redundancies. We elaborate on the implications for each of Assump. 1 and 2 and we provide an account of the memory performance of Alg. 1 by analysing its temporal locality.

PROPOSITION 1. *Under assumptions of the ideal-cache model, and when none of the data structures employed fits in cache, Alg. 1 incurs $\mathcal{O}(\lceil t/B \rceil m) + \mathcal{O}(tmN)$ I/O's in total, assuming worst case scenario.*

PROOF. We analyse each iteration i of the loop in Step 2, whose instructions are mimicked in Step 3. A sequence of batches of related memory accesses affecting four data structures takes place as follows. A vector point in the plane is read from (or written to) each of the arrays A_i using two memory accesses (corresponding to the respective vector coordinates being read or written). For every vector point that one reads from array A_{i-1}, one checks whether this point is in the given polygon by reading two records from array E (see Assump. 1 above). If the point is in the polygon, the corresponding record in the flag matrix F is read to determine whether or not the point has been written to A_i already. If the flag is *off*, the corresponding record in F is flagged as *on*, and the vector point is written to array A_i. This batch of at most eight memory operations will be crucial in our present proof and so we label it as $\mathcal{B}$ for further use below.

We elaborate on the resulting worst-case memory performance when none of the data structures fits in cache. The vector points in set A_{i-1} are always read contiguously. In the worst-case analysis, new vector points will get produced in an order such that their y coordinates follow an arbitrary pattern in the plane. In this case, no two or more vector points produced in succession will require records in the one dimensional array E that are found in the same block from E. Now, the flag matrix F is a two dimensional array, which is typically stored in row- (or column-) major layout. Yet, in the worst-case analysis, new vector points will get produced in an order such that their Cartesian coordinates trace an arbitrary pattern, so that every time a vector point is produced and its flag needs to be checked, a new block of F needs to be accessed. Finally, all vector points which pass the inclusion test get copied contiguously to the set A_i. We now build the count per each iteration in Step 2 and per Step 3. One reads at most t vector points from A_{i-1}. As a result, one also produces at most tN vector points to be tested for inclusion and redundancy. In the worst-case analysis, every such vector point requires that two new blocks from E and F respectively be brought into the cache. When the cache is full, the optimal replacement strategy chooses to evict the cache line which is to be used furthest away in the future. Following the order of operations in every batch $\mathcal{B}$ defined above, such a line will be the one containing the latest records retrieved from E or F, but not A_{i-1} or A_i. Thus, words in the lines belonging to A_{i-1} and A_i will be consumed entirely (and contiguously) in cache before their respective lines are evicted, bringing the total number of I/O's due to the core vector computations to $\mathcal{O}(\lceil t/B \rceil)$. On the other hand, the arbitrary memory accesses to E and F may cause a new block from E or F to be brought into cache, for every record needed. Since at most $\mathcal{O}(tN)$ new vector points are tested for inclusion and redundancy, the total number of I/O's required for those tasks is $\mathcal{O}(tN)$. Assuming the worst-case scenario defined per iteration is sustained in all m iterations of Alg. 1, we obtain the total asymptotic count in Prop. 1. □

We observe the following consequent ramifications of the proof above. Firstly, we note that memory accesses resulting from the core vector computations on one hand and from testing for inclusion and eliminating redundancies on the other, are independent of each other. Secondly, only accesses to the arrays $\{A_i\}$ are dictated by the order of vector computations taking place, in contrast to those accesses associated with Assump. 1 and 2, which follow a random pattern independently of the order of vector computations. We limit ourselves in this paper to improving on the cache behaviour associated with the core vector computations only. To do this in isolation comes at the expense of increased vector computations and consequently increased memory accesses to the arrays $\{A_i\}$ as a result of the following strategy: To eliminate the effect of memory accesses associated with Assump. 2, we allow the redundant vector points to propagate from one iteration to the next. Alternatively, one must seek ways by which to predict the redundancies in vector computations before they actually happen, but this option is beyond the scope of the current work and is only slightly investigated in [3]. We eliminate the effect of memory accesses associated with Assump. 1 by allowing more vector points not necessarily belonging to the input polygon to propagate from one iteration to the next. We achieve this by resorting to a less strict inclusion test than the exact one, but which does not require any auxiliary data structures. We label the new test as *crude*, because of the way it works. Recall that the smallest rectangle embedding our given polygon P – call it $\mathcal{R}$ – is of dimensions $x_{\max} \times y_{\max}$, where $y_{\max}$ and $x_{\max}$ denote the maximum vertex coordinates in y and x respectively, and $y_{\min}$ and $x_{\min}$ denote the minimum vertex coordinates in y and x respectively. Consequently, we can accept *crudely* that an arbitrary point (a, b) propagate to the next iteration if it belongs to $\mathcal{R}$, or that $a \leq x_{\max}$ and $b \leq y_{\max}$. We show that this does not compromise correctness as follows:

Proposition 2. *In Alg. 1 above, if one replaces the exact test for inclusion in P with the crude test which decides for inclusion within the smallest rectangle $\mathcal{R}$ containing P, then the algorithm remains correct.*

Proof. If the crude test accepts a point that is already in P, then there is nothing to prove. We thus need to show that the algorithm remains correct even if the crude test accepts a point $\alpha = (a, b)$ that is in $\mathcal{R}$ but not in P. In this case, if P is decomposable, we claim that we continue to get $v_0 \in A_{m-1}$: By Thm. 16 of [19], whenever there exists a non-trivial summand of P, Alg. 1 will detect this summand using points already in P. Obviously, such points get accepted by the crude test. On the other hand, if P is indecomposable, we claim that no point in $\mathcal{R} \setminus P$ will cause v_0 to be in A_{m-1}. To see this, let $\alpha = (a, b) \in \mathcal{R} \setminus P$ be a point that is produced (and accepted) in some iteration j. Then α should have been found as $\alpha = v_0 + \sum_{0 \leq i \leq j} k_i e_i$ for some $j < m - 1$. Suppose that further iterations produce a vector point of the form

$$q = (a, b) + \sum_{j<i<m} k_i e_i = v_0 + \sum_{0 \leq i<m} k_i e_i \qquad (1)$$

such that $q = v_0$. This implies that we were able to obtain $\sum_{0 \leq i<m} k_i e_i = (0, 0)$ with at least one k_i not equal to zero and at least one k_i not equal to n_i. By Lemma 13 of [19], this implies that a summand of P has been detected, a contradiction, since P is indecomposable. □

Thereafter, we are concerned with memory accesses triggered only by the core vector computations. We begin by re-casting the run time, space requirements, temporal locality measure, and spatial locality measure of Alg. 1 using the following assumptions:

1. The crude inclusion test is employed.
2. The redundancies are propagated.
3. The number of vector points in P is bounded loosely from above, as a result of the worst-case scenario defined by that all vector points produced in Steps 2 and 3 of Alg. 1 are accepted by the inclusion test.

Let $L_A(n)$ denote the access locality function introduced in [8], which returns the maximal possible distance between two records (of some array A) that are accessed within n contiguous operations. We now have:

Proposition 3. *Let $N' = \min\{n_i\}_{0 \leq i<m}$ and $N = \max\{n_i\}_{0 \leq i<m}$. In the worst-case analysis, and assuming all redundancies in computation are propagated and the crude inclusion test is employed, Alg. 1 requires*

$$\begin{cases} \Omega({N'}^m - m) & \text{if } m \geq 2 \\ \Omega({N'}^2 - m) & \text{if } m = 1. \end{cases} \qquad (2)$$

operations and $\mathcal{O}(N^m)$ space. Furthermore, the algorithm requires the following number of I/O's in total:

$$\begin{cases} \Omega\left(\left\lceil \frac{{N'}^m - m}{B} \right\rceil\right) & \text{if } m \geq 2 \\ \Omega\left(\left\lceil \frac{{N'}^2 - m}{B} \right\rceil\right) & \text{if } m = 1. \end{cases} \qquad (3)$$

Its spatial locality is characterised by the access locality function satisfying

$$L_A(p) = p/N'. \qquad (4)$$

Proof. The calculations below correspond to the worst-case scenario, when all vector points produced turn out to be in the polygon. We also recall that all reads and writes to arrays $\{A_i\}$ happen contiguously. Also, we consider that the input polygon is non-trivial so that $m \geq 3$. The total number of vector operations can be computed as follows. Recall that n_i denotes the length of edge E_i, for $0 \leq i < m$. In the first iteration of Step 2, one produces at most $C(0) = n_0$ vector points. In the second iteration of Step 2, one produces at most $C(1) = n_1 + n_0 n_1$ points. Generally, in the i'th iteration of Step 2, one produces at most

$$\begin{aligned} &C(i-1) = n_{i-1} \\ &+ (n_{i-2} + (\ldots + (n_2 + (n_1 + n_0 n_1) n_2) \ldots) n_{i-2}) n_{i-1} \end{aligned} \qquad (5)$$

vector points, for $i = 3, \ldots, m - 1$. In Step 3 (the m'th iteration), one produces at most

$$\begin{aligned} &C(m-1) \\ &= (n_{m-1} - 1) \cdot (n_{m-2} + (n_{m-3} + (\ldots \\ &+ (n_2 + (n_1 + n_0 n_1) n_2) \ldots) n_{m-3}) n_{m-2}) \end{aligned} \qquad (6)$$

vector points. In the worst case, the total number of vector points produced at termination is equal to $\sum_{i=0}^{m-1} C(i)$.

Write

$$\begin{aligned}
&\sum_{i=0}^{m-1} C(i) = \sum_{i=0}^{m-2} C(i) + C(m-1) \\
&\in \Theta\Big(\sum_{i=0}^{m-2} C(i) \\
&+ n_{m-1}\left(n_{m-2} + (\ldots + (n_2 + (n_1 + n_0 n_1) n_2) \ldots) n_{m-2}\right)\Big) \\
&\in \Theta\big(n_0 \\
&+ n_1 + n_0 n_1 \\
&+ n_2 + n_1 n_2 + n_0 n_1 n_2 \\
&\vdots \\
&+ n_{m-2} + n_{m-3} n_{m-2} + n_{m-4} n_{m-3} n_{m-2} + \ldots \\
&\qquad + n_0 n_1 n_2 \cdots n_{m-2} \\
&+ n_{m-2} n_{m-1} + n_{m-3} n_{m-2} n_{m-1} + \ldots + n_0 n_1 n_2 \cdots n_{m-1}\big),
\end{aligned}$$

where the expansions follow from (5) and (6). Rearranging terms we get:

$$\begin{aligned}
\sum_{i=0}^{m-1} C(i) \;&\in\; \Theta\big(n_0 + n_1 + \ldots + n_{m-2} \\
&+\; n_0 n_1 + n_1 n_2 + \ldots + n_{m-2} n_{m-1} \\
&+\; n_0 n_1 n_2 + n_1 n_2 n_3 + \ldots + n_{m-3} n_{m-2} n_{m-1} \\
&\;\vdots \\
&+\; (n_0 n_1 \ldots n_{m-2}) + (n_1 n_2 \ldots n_{m-1}) \\
&+\; (n_0 n_1 \ldots n_{m-1})\big),
\end{aligned}$$

Let $N' = \min\{n_i\}_{0 \le i < m}$. We can then continue as

$$\begin{aligned}
&\sum_{i=0}^{m-1} C(i) \\
&\in \Omega\Big((m-1)N' + (m-1){N'}^2 + (m-2){N'}^3 \\
&+ \ldots + 2{N'}^{m-1} + {N'}^m\Big) \\
&= \Omega\Big(N' + {N'}^2 + {N'}^3 + \ldots + {N'}^m \\
&+ N' + {N'}^2 + {N'}^3 + \ldots + {N'}^{m-1} \\
&+ N' + {N'}^2 + {N'}^3 + \ldots + {N'}^{m-2} \\
&\vdots \\
&+ N' + {N'}^2 + {N'}^3 + \\
&2(N' + {N'}^2)\Big).
\end{aligned}$$

We now use the value of each of the geometric series above $\left(\sum_{j=0}^{k} {N'}^j = \frac{{N'}^{k+1}-1}{N'-1}\right)$ and obtain:

$$\begin{aligned}
&\sum_{i=0}^{m-1} C(i) \in \Omega\left(\left(\frac{{N'}^{m+1}-1}{N'-1} - 1\right) + \left(\frac{{N'}^m-1}{N'-1} - 1\right)\right. \\
&\left. + \ldots + \left(\frac{{N'}^4-1}{N'-1} - 1\right) + 2\left(\frac{{N'}^3-1}{N'-1} - 1\right)\right).
\end{aligned}$$

Using $\frac{{N'}^{i+1}-1}{N'-1} = \Theta({N'}^i)$, we write $\frac{{N'}^{i+1}-1}{N'-1} = c_i {N'}^i$ for some positive constant $c_i > 0$ and sufficiently large N', and we let $c = \min\{c_i\}_{2 \le i \le m}$. We get:

$$\begin{aligned}
&\sum_{i=0}^{m-1} C(i) \\
&\in \Omega\left(c_m {N'}^m + c_{m-1}({N'}^{m-1} + \ldots + c_3 {N'}^3 + 2c_2 {N'}^2 - m\right) \\
&\in \Omega\left(c\left({N'}^m + {N'}^{m-1} + \ldots + {N'}^3 + 2{N'}^2\right) - m\right) \\
&= \Omega\left(c\left(\frac{{N'}^{m+1}-1}{N'-1} + {N'}^2 - N' - 1\right) - m\right) \\
&= \begin{cases} \Omega({N'}^m - m) & \text{if } m \ge 2 \\ \Omega({N'}^2 - m) & \text{if } m = 1 \end{cases}
\end{aligned}$$

and Eq. (2) is now obtained. To determine the space complexity, note that in the worst-case analysis, all vector points generated during each iteration of Step 2 and during Step 3 are accepted by the inclusion test. The size of each array A_i corresponds to the number of vector vector points in $\mathcal{R}$ generated at the end of the i'th iteration. The largest such set is required for Step 3, during which $C(m-1) = \mathcal{O}(N^m)$ vector points are produced and stored.

To assess temporal locality, recall that all memory accesses to the one dimensional arrays A_i happen contiguously, and start at the first array location. We proceed iteratively, still assuming worst-case scenario. In the first iteration of Step 2, one writes at most n_0 vector points to A_0, and so has to perform at most $R(0) = \lceil n_0/B \rceil$ I/O's. In the second iteration of Step 2, one reads at most n_0 vector points, and then writes at most $(n_1 + n_0 n_1)$ vector points to A_1. For this, one has to perform at most

$$R(1) = \left\lceil \frac{n_0}{B} \right\rceil + \left\lceil \frac{n_1 + n_0 n_1}{B} \right\rceil = \left\lceil \frac{C(0)}{B} \right\rceil + \left\lceil \frac{C(1)}{B} \right\rceil \tag{7}$$

I/O's. Continuing in this fashion, one has to perform at most

$$R(i-1) = \left\lceil \frac{C(i-2)}{B} \right\rceil + \left\lceil \frac{C(i-1)}{B} \right\rceil \tag{8}$$

I/O's in total, during any i'th iteration of Step 2, for $i = 3$, …, $m-1$, and at most

$$R(m-1) = \left\lceil \frac{C(m-2)}{B} \right\rceil + \left\lceil \frac{C(m-1)}{B} \right\rceil \tag{9}$$

I/O's in total, during the m'th iteration in Step 3. Let $Q(N, m; M, B)$ denote the worst-case cache complexity incurred on an ideal-cache of size M and of cache-line length equal to B. Then

$$Q(N, m; M, B) = \sum_{i=0}^{m-1} R(i) = \sum_{i=0}^{m-1} \lceil C(i)/B \rceil .$$

Proceeding similarly as for the calculations leading up to Eq. (2) one obtains the worst-case cache complexity in Eq. (3) above.

To characterise data locality, note that the algorithm is such that one has to access t consecutive records from each of the vector arrays A_i in order to perform at most tN and at least tN' operations per one iteration of Step 2 and in Step 3. Put differently, one needs to access at least p/N and at most p/N' contiguous records of each array A_i in order to perform p vector operations, for some $p \in \mathbb{Z}^+$. We thus have $L_A(p) = p/N'$. $\square$

The implications of Prop. 3 can be informally stated as follows: The frequency within which one re-visits vector points in the sets A_i is high with respect to computation. Specifically, we do not seem to be maximising on the presence of data in cache, as the iterative nature of the algorithm causes points to be moved in and out of cache too often. This can be re-phrased that there are too many jumps over the data structure, and paves the way for a (natural) recursive reformulation to which we dedicate the remainder of this paper.

4. CACHE-OBLIVIOUS POLYGON INDECOMPOSABILITY TESTING

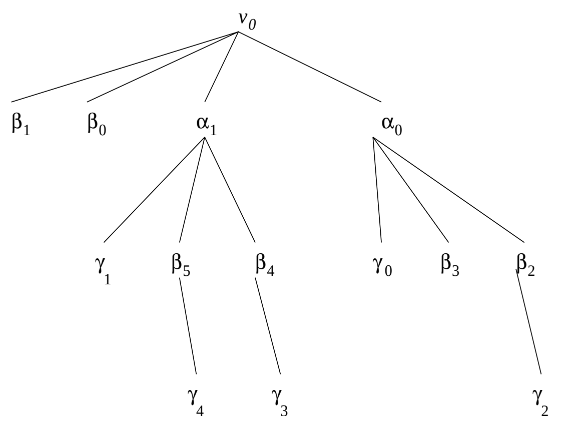


Figure 1: Computation tree.

4.1 The computation tree and DFS traversal

We begin by recasting the vector computations in a tree-like fashion. In all of the following, we let $N_j = n_j$ if $0 \leq j < m-1$ and $N_j = n_j - 1$ if $j = m-1$. Initially, let the root of the computation tree be v_0, positioned in level 0. By Step 2.1 of Alg. 1, consider all the independent computations on v_0 given by

$$\mu_i = v_0 + k_i e_i, \quad \text{for } i = 0, \ldots, m-2, \quad \text{and } k_i = 1, \ldots, N_i. \tag{10}$$

This generates level 1 of the tree where each child node of v_0 corresponds to some μ_i defined as in (10). Thereafter, level $\ell \geq 2$ of the tree can be constructed using all nodes in level $\ell - 1$ as follows. Each node μ in level $\ell - 1$ can be expressed using the vector sum

$$\mu = v_0 + \sum_{j=0}^{i_\mu} k_j e_j, \text{ for some } i_\mu < m \text{ and } k_j = 1, \ldots, N_j. \tag{11}$$

By Steps 2.2 and 3 of Alg. 1, each child node of μ in level ℓ will correspond to the vector

$$\tau = \mu + k_j e_j, \quad \text{for } j = i_\mu + 1, \ldots, m-1 \text{ and } k = 1, \ldots, N_j.$$

The nodes in the sub-tree rooted at any one node μ_ℓ in level ℓ correspond to all vector computations that can be issued using μ_ℓ. For $\ell \geq 1$, the maximum number of nodes in any such sub-tree is given by:

$$n_\ell \cdot n_{\ell+1} \cdots n_{m-1} \leq N^{m-\ell}. \tag{12}$$

The bound in (12) can be derived using an argument close to the proof of Prop. 3. Using the same example of Sect. 2.3 above, our tree structure is given in Fig. 1 and the sequence of computations in that example:

$$v_0,\ \alpha_0,\ \alpha_1,\ \beta_0,\ \beta_1,\ \beta_2,\ \beta_3,\ \beta_4,\ \beta_5,\ \gamma_0,\ \gamma_1,\ \gamma_2,\ \gamma_3,\ \gamma_4$$

proceeds in a breadth first manner through the tree. This is precisely why one has to keep record of all nodes computed in any one level in order to proceed to the next, whence the asymptotic costs in Thm. 3. However, if we were to re-cast those same computations in a depth first fashion, benefiting from the independence among vector computations in any one level of the tree, we generate recursively the following alternative sequence of computations:

$$v_0,\ \beta_0,\ \beta_1,\ \alpha_0,\ \gamma_0,\ \beta_2,\ \gamma_2,\ \beta_3,\ \alpha_1,\ \gamma_1,\ \beta_4,\ \gamma_3,\ \beta_5,\ \gamma_4. \tag{13}$$

This simple trick has crucial consequences: The recursive variant is correct since it sweeps over the same number of potential summands of the given polygon. It also has the same work complexity, since it spans the same vector computations as in the original algorithm, but with a different pattern which still respects the dependencies between various operations. The recursive variant organises the data without requiring knowledge of the size of the cache or cache lines, and hence is cache-oblivious. Finally, the change in the data structures and pattern of computations can be shown to result in improved temporal and spatial locality. In particular, the recursive version ensures that memory addresses across the new data structures it will require change with step-size at most one.

We shall speak (and possibly abuse the terminology) of a DFS-based polygon indecomposability testing algorithm to mean a depth first generation of the computation tree, as opposed to a breadth first generation of the same tree. The generic results in the literature which tackle BFS work performance as opposed to DFS (depending on the likely position of the target(s) being sought) do not carry on to our application: The computation tree being treated is not to be searched, but rather generated in its entirety, and so the number of nodes to be produced (and thus "traversed") is the same for both types of traversal, unlike the case for a tree search. Similarly, we caution against confusing the present work with cache-aware or cache-oblivious BFS and DFS algorithms on graphs and trees. Our goal is to analyse how polygon indecomposability testing can be performed cache-obliviously with the help of a natural and recursive depth first traversal of its computation tree. The input, intermediary output, and final output of polygon indecomposability testing require data structures that are different from those required to instantiate a tree (graph) and perform a search on it. Moreover, as we go from a breadth first to a depth first generation of the computation tree, the data structures required to perform indecomposability testings change, which precludes us from drawing the analogy with studies casting cache-efficient BFS versus DFS algorithms on graphs.

In the following sections we formalise our DFS variant and establish all the claims above.

4.2 The DFS variant

We introduce three one-dimensional arrays K, U, and V, each of size m, as follows: Give each node $\mu = v_0 + \sum_{j=0}^{i_\mu} k_j e_j$ (as in (11)) of the computation tree of Sect. 4.1, we write to array K such that $K[j] = k_j$ for each k_j in the tuple characterising μ as in (11). We also read from array U such that $U[j] = N_j$ for $0 \leq j \leq i_\mu$ (recall N_j from Sect. 4.1). Finally, we read from array V such that $V[j]$ contains the coordinates in $\mathbb{Z}^2$ of the primitive vector e_j.

Algorithm 2. *Input: The edge sequence $\{n_i e_i\}_{0 \leq i < m}$ of an integral convex polygon P starting at a vertex v_0 where $e_i \in \mathbb{Z}^2$ are primitive vectors.*

```
Output: Whether P is decomposable.
A. Main()
Step 1: Initialise a global vector $u \leftarrow v_0$, and a global boolean variable $b \leftarrow FALSE$. Call DFS-compute(0).
Step 2: Return "Decomposable" if $v_0$ is found in any of the leaves, and "Indecomposable" otherwise.
B. DFS-compute(j)
Step 1: $K[j] = 0$.
Step 2: While $K[j] \leq U[j]$ do:
```

2.1: If $b = FALSE$ and $j = m - 1$:
Set $b \leftarrow TRUE$ and exit the loop.
// i.e. do not compute $v_0 + ke_{m-1}$ for any given k
2.2: Compute $u \leftarrow u + K[j] \cdot V[j]$.
2.3: If $u \in P$ (or $\mathcal{R}$) and $j \neq m - 1$: Call DFS-compute$(j + 1)$.
2.4: $u \leftarrow u - K[j] \cdot V[j]$, $K[j] \leftarrow K[j] + 1$.

4.3 Cache complexity analysis

The three arrays K, U, and V require $\Theta(m)$ space. We begin by assessing the spatial locality of the DFS variant using the operation graph representation (see [8]). Let each node in the graph represent a triple (j_K, j_U, j_V) such that a particular vector operation is requiring access to $K[j_K]$, $U[j_U]$ and $V[j_V]$, in succession. An edge in this graph will connect two nodes $\mu = (j_K, j_U, j_V)$ and $\mu' = (j'_K, j'_U, j'_V)$ iff $\max(|j_K - j'_K|, |j_U - j'_U|, |j_V - j'_V|) \leq 1$. Informally speaking, an edge will connect two nodes of the graph iff the index jump in either of the three indices does not exceed 1. A close inspection of the DFS variant shows that for any given node μ in the graph, we have $j_K = j_U = j_V$, so that it suffices to keep track of the index jumps affecting either of the three arrays, say K. Using an illustration from our example in Sect. 2.3, we trace the vector computations v_0, β_0, γ_1, β_1, and α_0 in (14). We also show the associated memory accesses to array K in Fig. 2. Because of the inherent DFS traversal, we identify the following key characteristics:

1. $\mathcal{C}_1$: After each vector operation, we always either re-use an array record, or else move to its direct neighbour.

2. $\mathcal{C}_2$: As a result of $\mathcal{C}_1$, we can traverse the entire graph forward starting from the node [0] without any jumps in the access of array record. This is so in the sense that such accesses progress with step-size at most one, across all records of any one array and which are positioned in contiguous chunks of memory.

Continuing in this fashion, one can even verify that the two observations still hold until the algorithm produces all vectors in (13), thus ensuring spatial locality. We prove this claim and establish the cache complexity in Prop. 4, 5 and 6. We conclude with Corollary 1.

$$\begin{array}{lll}
op\,1: & K[0] \leftarrow 0 \Rightarrow & u \leftarrow v_0 + 0 \cdot e_0 = v_0 \quad \textit{Accept.} \\
op\,2: & K[1] \leftarrow 0 \Rightarrow & u \leftarrow v_0 + 0 \cdot e_0 + 0 \cdot e_1 = v_0 \quad \textit{Accept.} \\
op\,3: & K[2] \leftarrow 0 \Rightarrow & \ldots \quad \textit{Exceeds the bounds.} \\
op\,4: & K[1] \leftarrow 1 \Rightarrow & u \leftarrow v_0 + 0 \cdot e_0 + 1 \cdot e_1 = \beta_0 \quad \textit{Reject.} \\
op\,5: & K[1] \leftarrow 2 \Rightarrow & u \leftarrow v_0 + 0 \cdot e_0 + 2 \cdot e_1 = \beta_1 \quad \textit{Reject.} \\
op\,6: & K[1] \leftarrow 3 \Rightarrow & \ldots \quad \textit{Exceeds the bounds.} \\
op\,7: & K[0] \leftarrow 1 \Rightarrow & u \leftarrow v_0 + e_0 = \alpha_0 \quad \textit{Accept.} \\
op\,8: & K[1] \leftarrow 0 \Rightarrow & u \leftarrow v_0 + e_0 + 0e_1 = \alpha_0 \quad \textit{Accept.}
\end{array} \tag{14}$$

PROPOSITION 4. *Alg. 2 ensures spatial locality.*

PROOF. Firstly, note that this result does not get affected by the fact that redundant vector points are propagated, as is evident from our analysis below. Recall that we are tracking only the sequence of writes to array K. We prove that spatial locality is ensured by showing that, given any input, the operation graph representation will always satisfy the criteria $\mathcal{C}_1$ and $\mathcal{C}_2$ above. We proceed by induction on the number of nodes in the operation graph for a given polygon with m edges. Consider the first two nodes of the graph. Those correspond to operations 1 and 2, causing the updates $K[0] \leftarrow 0$ and $K[1] \leftarrow 0$. Clearly, criterion $\mathcal{C}_1$ holds. Assume now that the first i operations in the graph satisfy $\mathcal{C}_1$, and

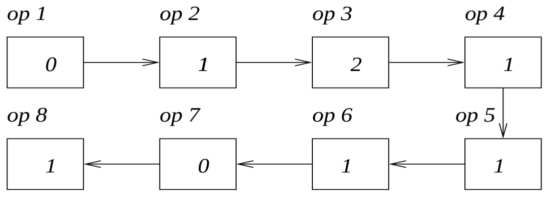


Figure 2: Operation graph representation

assume that the i'th operation causes an update on some $K[j]$. We distinguish the following three cases:

- Operation i triggers a recursive call, leading to the $(i + 1)$'st operation: $K[j + 1] \leftarrow 0$ – e.g. operations $i = 1$, 2, and 7 in Fig. 2. A new edge will be added connecting node $[j]$ to node $[j + 1]$, which establishes $\mathcal{C}_1$ for operation $i + 1$.

- Operation i triggers a return from some recursive call, leading to the $(i + 1)$'st operation: $K[j - 1] \leftarrow K[j - 1] + 1$ – e.g. operations $i = 3$ and 6 in Fig. 2. A new edge will be added connecting node $[j]$ to node $[j - 1]$, which establishes $\mathcal{C}_1$ for operation $i + 1$.

- Operation i does not trigger either a new recursive call or a return from a recursive call, leading to the $(i+1)$'st operation: $K[j] \leftarrow K[j] + 1$ – e.g. operations 4 and 5 in Fig. 2. A new edge will be added connecting node $[j]$ to itself, which establishes $\mathcal{C}_1$ for operation $i + 1$.

This concludes the inductive proof. Criterion $\mathcal{C}_2$ can now be deduced from $\mathcal{C}_1$. □

PROPOSITION 5. *In the worst-case analysis, and assuming all redundancies in computation are propagated and the crude inclusion test is employed, Alg. 2 decides decomposability correctly using*

$$\mathcal{O}\left(\frac{N^m}{N^{(1-\varepsilon)B-i}}\right) \quad \textit{I/O's}, \tag{15}$$

where $\varepsilon = \lceil m/B \rceil - m/B$, and $i = 0, \ldots, B-1$, assuming the ideal-cache model. Here, i denotes the alignment parameter.

PROOF. As before, we obtain worst-case scenario when all vector points get accepted by the inclusion test. Assume that none of the three arrays K, U, and V, fits in cache. By Alg. 2, the total number of reads to all of the three arrays is a constant times the total number of writes to array K. Hence, it suffices, for our asymptotic analysis, to focus on the sequence of writes to array K.

First, note that we write to $K[s]$ upon each call to *DFS-compute*(s), and each return from *DFS-compute*$(s + 1)$. We will shorten expression by referring to calls to *DFS-compute*(s) and returns from *DFS-compute*$(s + 1)$ as simply calls to $[s]$ and returns from $[s + 1]$ respectively. We proceed as follows:

(i) We first calculate the total number of calls to $[s]$, and the total number of returns from $[s]$, for $0 \leq s < m$.

(ii) We then determine how many of such calls and returns require I/O's between cache and main memory.

(iii) We finally use (ii) to determine the worst-case cache complexity of the DFS variant.

We elaborate as follows:

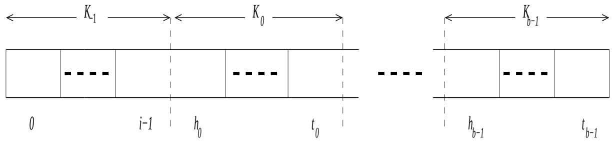


Figure 3: The blocks $\{K_j\}_{j=-1}^{b-1}$

(i) We begin by calculating the total number of calls to $[s]$, for $0 \leq s < m$, always recalling the worst-case scenario. By Step A.1 of Alg. 2, the number of calls to [0] is equal to one. Upon this call, the bound on the loop in Step 2 triggers a number of $(n_0 + 1)$ calls to [1]. Thus, the total number of calls to [1] is equal to $(n_0 + 1)$. Similarly, upon each call to [1], one encounters $(n_1 + 1)$ calls to [2]. Thus, the total number of calls to [2] is equal to $(n_0+1)(n_1+1)$. Proceeding iteratively, one obtains that the total number of calls to $[s]$ is equal to

$$(n_0 + 1) \cdot (n_1 + 1) \cdots (n_{s-1} + 1), \tag{16}$$

for $1 \leq s < m$. Note that the number of calls to each $[s]$ is equal to the number of returns from $[s]$.

(ii) By Prop. 4, accesses to K happen contiguously. Hence, not all of the writes to K resulting from the total number of calls and returns as in (16) trigger an I/O operation between cache and main memory, and we need to determine which of them actually do. Recall that B denotes the size of the cache-line as well as of the block of records that can be transferred simultaneously between cache and main memory. We reason about array K as a list of blocks of contiguous records. We define a *head* in K to be that record which, when loaded into cache, falls at the beginning of a cache-line boundary, and the associated *tail* in K to be that record which falls at the end of that same cache-line. All the B records starting at the head and ending at the tail make up a *full block*. Contiguous records of K which occupy part (i.e. less than B words) of a cache-line make up a *partial block*.

Considering all possible alignments, the first head in K, denoted by h_0, can only be one of the records at $i = 0, \ldots, B-1$. This fixes the alignment parameter i in the remainder of the present discussion. The first head h_0 defines the first block K_0 of B contiguous records, ending with the first tail $t_0 = h_0 + B - 1$. Let $b = \lceil m/B \rceil$. In the worst case, array K will be partitioned into at most $b - 1$ full blocks and two partial blocks as follows. Once we have defined K_0, the remainder of the full blocks K_j, for $j = 1, \ldots, b - 2$, are defined by the head $h_j = i + jB$ and tail $t_j = h_j + B - 1$. The two non-full blocks – which we label K_{-1} and K_{b-1} – are as follows:

- When $K[0]$ does not fall at the beginning of a cache-line boundary, we get the partial block K_{-1} of size less than B, with (atypical[1]) head $h_{-1} = 0$ and tail $t_{-1} = i - 1$.
- When $K[m-1]$ does not fall at the end of a cache-line boundary, we get the partial block K_{b-1} of size less than B, with head $h_{b-1} = i + (b-1)B$ and (atypical) tail $t_{b-1} = m - 1$.

We now claim the following properties:

[1]Atypical here refers to the head not falling at the beginning of a cache-line boundary. Analogously for the tail.

- $\mathcal{P}_1$: Each call to [0] or a block head requires an I/O operation, but each call to a non-head does not.
- $\mathcal{P}_2$: Each return from a block head requires an I/O operation, but each return from a non-head does not.

The proofs of $\mathcal{P}_1$ and $\mathcal{P}_2$ depend on the following rule:

Remark 1. *Assuming worst-case scenario when all vector points produced get accepted by the inclusion test, the condition in Step 2.3 of Alg. 2 will always hold true. This, along with the DFS pattern of the computations, result in the following:*

1. A call to each $[s]$ has to be followed by a chain of calls up to and including the call to $[m-1]$, uninterrupted by any function returns.

2. A return from each $[s]$ has to be preceded by a chain of returns emanating from $[m-1]$, uninterrupted by any function calls.

We can prove the two properties $\mathcal{P}_1$ and $\mathcal{P}_2$ by induction on the number of blocks. We start with $\mathcal{P}_1$. Consider the initial call issued to [0]. Based on the alignment of the blocks of K as explained above, the initial call requires access to either K_{-1} or K_0, depending on how $K[0]$ is situated with respect to a cache-line boundary:

Case (a): If $K[0]$ falls at the beginning of a cache-line boundary, we have $K[0] = h_0$, in which case K_0 will be transferred to occupy one full cache-line of length B, upon the call to [0]. We now examine each subsequent call to a non-head $[k]$ of block K_0, for $k = 1, \ldots, B - 1$. Because of the DFS traversal, each such call can only follow immediately after a sequence of calls and returns affecting the previous records at $k - 1, k - 2, \ldots, k - h$, for $h = 1, \ldots, k$. But these records belong to the local block K_0. Thus, no call to the non-head node $[k]$ of K_0 requires an I/O operation, for all $k = 1, \ldots, B - 1$, and so $\mathcal{P}_1$ holds initially.

Case (b): if $K[0]$ does not fall at the beginning of a cache-line boundary, we have $K[0] = h_{-1}$, and a proof of the base case can be sketched similarly to case (a) above, with K_{-1} instead of K_0. We leave the details for the reader.

We now assume that $\mathcal{P}_1$ holds for the next j blocks – i.e. that each call to the heads $h_1, \ldots, h_j$ requires an I/O operation but each call to the non-heads preceding h_{j+1} does not. We show that $\mathcal{P}_1$ is maintained for block K_{j+1} – i.e. that each call to h_{j+1} requires an I/O operation but each call to the non-heads preceding h_{j+2} does not. A call is issued to h_{j+1} only immediately after a call to the preceding tail t_j of K_j. In turn, and because of the DFS traversal, the call to t_j can only follow after a sequence of calls and returns affecting the preceding records at $h_j, h_j + 1, \ldots, t_j - 1$. But these records belong to the block K_j, which, by induction, should have been loaded into cache the moment the call to its head h_j was issued. By the address alignment properties of K_j, the tail of this block is found at the end of the cache-line containing K_j. Therefore, the call to the adjacent head h_{j+1} requires an I/O operation in order to load K_{j+1} into cache. One can further show, similarly to the base case, that all calls to non-heads of block K_{j+1} do not require an I/O operation. This establishes $\mathcal{P}_1$.

To prove $\mathcal{P}_2$, we proceed by induction starting at the last block of K and working backwards across the blocks. Assuming worst-case scenario when all vector points produced get accepted by the inclusion test, the first return in the course of the algorithm is issued from $[m - 1]$. Depending on how $K[m-1]$ is situated with respect to a cache-line boundary, we have one of two cases:

Case (c): If $K[m-1]$ does not fall at the end of a cache-line boundary, this record will be the tail of block K_{b-1}. In this case, the return issued from $[m-1]$ follows only immediately after a sequence of operations triggered by a call to h_{b-1}. By property $\mathcal{P}_1$ above, the call to h_{b-1} requires that K_{b-1} be loaded into cache, so that the first return from $[m-1]$ accesses data that is already local. We now examine each subsequent return from a non-head $[k]$ of K_{b-1}, where $k = m-2$, $m-3$, …, $h_{b-1}+1$. By Remark 1, each such return can only follow immediately after a sequence of returns emanating from $[m-1]$, $[m-2]$, …, $[k+1]$, which affect records in the local block K_{b-1}. Thus, none of the returns from the non-heads of K_{b-1} require an I/O operation. However, the return from h_{b-1} requires access to the tail t_{b-2} of K_{b-2}, which, because of the alignment property of K_{b-1}, is not found in the cache-line containing K_{b-1}. Thus, an I/O operation is required, by which the entire block K_{b-2} is loaded into cache. With this, $\mathcal{P}_2$ holds initially.

Case (d): If $K[m-1]$ falls at the end of a cache-line boundary, this record will be the tail of either block K_{b-2} or K_{b-1} (depending on the alignment parameter i). A similar proof of the base case can be sketched similarly to case (c) above. We leave the details for the reader.

We now assume that $\mathcal{P}_2$ holds for the next penultimate j blocks, for $j < b-1$ – i.e. that each return from a head h_j requires an I/O operation, but each return from a non-head located after h_{j-1} does not. We show that $\mathcal{P}_2$ is maintained for block K_{j-1} – i.e. that each return from the head h_{j-1} requires an I/O operation, but each return from a non-head located after h_{j-2} does not. A return is issued from the tail t_{j-1} of K_{j-1} only immediately after a return from h_j. By induction, at the point of invocation of a return from h_j, the block K_j is already in cache. By the address alignment property of block K_j, h_j is found at the beginning of the cache-line containing this block, so that the return from h_j requires an I/O operation by which the entire block K_{j-1} is loaded into cache. No subsequent return from the non-head nodes of block K_{j-1} would then require an I/O operation. This establishes $\mathcal{P}_2$.

(iii) Let $Q'(N,m;M,B)$ denote the worst-case cache complexity incurred on an ideal-cache of size M and of cache-line length equal to B. We use (ii) to determine $Q'(N,m;M,B)$. Write $\varepsilon = b - m/B$. By Properties $\mathcal{P}_1$ and $\mathcal{P}_2$, $Q'(N,m;M,B)$ is obtained by summing up all the numbers of calls to and returns from the block heads $\{h_j\}_{-1\le j<b}$, with the total number of calls to and returns from the first head h_{-1} equal to 2. For the remaining heads, the total number of calls to each of them (also equal to the total number of returns from each of them) is given by (16), and by $(h_{-1}=0, \{h_j = i+jB\}_{0\le j<b})$, we have

$$\begin{aligned}
&Q'(N,m;M,B)\\
&< 2\left(1+\textstyle\sum_{j=0}^{b-1}(n_0+1)(n_1+1)\dots(n_{i+jB-1}+1)\right)\\
&\le 2\left(1+\textstyle\sum_{j=0}^{b-1}(N+1)^{i+jB}\right)\\
&= 2\left(1+(N+1)^i\cdot\textstyle\sum_{j=0}^{b-1}(N+1)^{jB}\right)\\
&= 2\left(1+(N+1)^i\cdot\tfrac{(N+1)^{bB}-1}{(N+1)^B-1}\right)\\
&= \Theta\left(N^i\cdot\tfrac{N^{((m/B)+\varepsilon)\cdot B}}{N^B}\right) = \Theta\left(\tfrac{N^m}{N^{(1-\varepsilon)B-i}}\right).
\end{aligned}$$

□

PROPOSITION 6. *In the worst-case analysis, and assuming all redundancies in computation are propagated and the crude inclusion test is employed, the spatial locality of Alg. 2 is characterised by the access locality function satisfying*

$$L_K(p) = \log_{N'} p. \qquad (17)$$

PROOF. As earlier, it suffices to focus on the writes to array K. We examine each record $K[m-s]$ of K, for $s = m$, …, 1. Between each two consecutive updates on $K[m-s]$, one has to perform all the vector computations in the subtree whose root corresponds to some vector μ in level $m-s$. By Sect. 4.1 we know that, in the worst-case analysis, the number of such vector computations is at most

$$n_{(m-s)}\cdot n_{(m-s+1)}\cdots n_{(m-1)}. \qquad (18)$$

To perform all of these operations, only records $K[m-s]$, …, $K[m-1]$ need to be accessed, spanning a range of length s. The maximum such range occurs for $s = m-1$, in which case $m-1$ records are accessed, in order to perform

$$p = n_1\cdot n_2\cdots n_{(m-1)} \qquad (19)$$

vector operations. With

$$(N')^{m-1} \le p \le N^{m-1}, \qquad (20)$$

it follows that one traverses a maximum range of $m-1$ contiguous records of K in order to perform at least N^{m-1} and at most N^{m-1} vector operations. We now apply the appropriate logarithms to bases N and N' respectively, and rephrase that one needs to access at least $\log_N p$ and at most $\log_{N'} p$ contiguous records of array K in order to perform p vector operations, for some $p \in \mathbb{Z}^+$. We thus have $L_K(p) = \log_{N'} p$. □

COROLLARY 1. *The cache-oblivious DFS variant outperforms Alg. 1 with respect to spatial locality for all given polygons. It outperforms Alg. 1 with respect to temporal locality for polygons satisfying*

$$m = \mathcal{O}\left(N'^m - \frac{BN^m}{N^{(1-\varepsilon)B-i}}\right), \qquad (21)$$

where $\varepsilon = \lceil m/B\rceil - m/B$, *and* $i = 0,\dots,B-1$.

PROOF. For the spatial complexity, the claim is evident from inspecting (4) versus (17). For the temporal locality, recall the functions $Q(N,m;M,B)$ and $Q'(N,m;M,B)$ denoting the worst-case cache complexity depicted in (3) versus (15) respectively. Recall that we are considering non-trivial input polygons such that $m \ge 3$. For the DFS variant to perform asymptotically less cache misses, we require that

$$\frac{N^m}{N^{(1-\varepsilon)B-i}} = \mathcal{O}\left(\frac{N'^m - m}{B}\right),$$

applicable when

$$m = \mathcal{O}\left(N'^m - \frac{BN^m}{N^{(1-\varepsilon)B-i}}\right).$$

□

REMARK 2. *We remark that the requirement in (21) is realistic to achieve in several instances. We sketch one scenario when this is possible. Take for example the case where* *the alignment parameter i is incorporated in code such that $i = 0$ (note that this can be done independently of cache parameters). Consider also the less stringent requirement on the input size such that $N' = \Theta(N)$. For (1) we would require that $m = \mathcal{O}\left(N^m\left(1 - \left(B/N^{(1-\varepsilon)B}\right)\right)\right)$, which is very realistic to achieve, since N grows much faster than B for sufficiently large input polygons that one may encounter frequently in practice.*

5. CONCLUSION

The cache-oblivious reformulation of the polygon indecomposability test of [19] allows access to its data structures to progress with step-size at most one, thus ensuring spatial locality. Also, under conditions governing the input size and the alignment of data, our analysis reveals that the DFS variant achieves improved temporal locality over the original one. The practical ramifications are significant as confirmed through the experiments and benchmarks produced in [3]. Some issues remain to be tackled before the full scale of the DFS variant is realised. Specifically, one needs to investigate means to resolve the redundancies in vector computations at a un-compelling cost. One possibility is to seek ways by which redundant vector computations can be predicted before they actually happen. In [3], we obtain that this can be done at least for the naive indecomposability testing algorithm.

6. ACKNOWLEDGMENTS

The author thanks the Lebanese National Council for Scientific Research for supporting phases of this work.

7. REFERENCES

[1] F. K. Abu Salem. An efficient sparse adaptation of the polytope method over $\mathbb{F}_p$ and a record-high binary bivariate factorisation. *Journal of Symbolic Computation*, 43(5):311–341, 2008.

[2] F. K. Abu Salem, S. Gao, and A. G. B. Lauder. Factorisation algorithms for univariate and bivariate polynomials over finite fields. In *Proc. 2004 International Symposium on Symbolic and Algebraic Computation. ACM Press*, pages 4–1, 2004.

[3] F. K. Abu Salem and R. N. Soudah. An empirical study of cache-oblivious polygon indecomposability testing. *To appear in Computing (DOI: 10.1007/s00607-010-0086-z)*, 2010.

[4] F. K. Abu Salem. Brief supplement to the manuscript on cache-oblivious polygon indecomposability testing. *Technical report (available from `www.cs.aub.edu.lb/fa21/Papers/COIrredTheor.pdf`)*, 2010.

[5] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[6] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[7] L. Arge, M. de Berg, and H. Haverkort. Cache-oblivious R-trees. In *Proc. of the 21st ACM Symposium on Computational Geometry*, pages 170–179, 2005.

[8] M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on the Peano curve. In *Proc. PPAM 2006*, pages 1042–1049, 2006.

[9] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, 2002.

[10] M. A. Bender, M. Farach-colton, J. T. Fineman, T. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proc. of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 81–92, 2007.

[11] M. A. Bender, M. Farach-colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 223–242, 2006.

[12] G. S. Brodal and R. Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proc. of the 13th International Symposium on Algorithms and Computation*, pages 219–228, 2002.

[13] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[14] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[15] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication, or tracking BLAS3 performance from source code. *SIGPLAN Not.*, 32:206–216, 1997.

[16] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[18] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proc. of the 19th ACM International Conference on Supercomputing*, pages 361–366, 2005.

[19] S. Gao and A. G. B. Lauder. Decomposition of polytopes and polynomials. *Discrete and Computational Geometry*, 26:89–104, 2001.

[20] B. Grünbaum. *Convex Polytopes*. John Wiley and Sons, 1967.

[21] R. E. Ladner, R. Fortna, and B.-H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics, From Algorithm Design to Robust and Efficient Software*, pages 78–92, 2002.

[22] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[23] D. S. Wise, C. L. Citro, J. J. Hursey, F. Liu, and M. A. Rainey. A paradigm for parallel matrix algorithms: Scalable Cholesky. In *Euro-Par 2005*, pages 687–698, 2005.

Parallel Sparse Polynomial Interpolation over Finite Fields [*]

Seyed Mohammad Mahdi Javadi
School of Computing Science
Simon Fraser University
Burnaby, B.C. Canada.
sjavadi@cecm.sfu.ca.

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
mmonagan@cecm.sfu.ca.

ABSTRACT

We present a probabilistic algorithm to interpolate a sparse multivariate polynomial over a finite field, represented with a black box. Our algorithm modifies the algorithm of Ben-Or and Tiwari from 1988 for interpolating polynomials over rings with characteristic zero to characteristic p by doing additional probes.

To interpolate a polynomial in n variables with t non-zero terms, Zippel's (1990) algorithm interpolates one variable at a time using $O(ndt)$ probes to the black box where d bounds the degree of the polynomial. Our new algorithm does $O(nt)$ probes. It interpolates each variable independently using $O(t)$ probes which allows us to parallelize the main loop giving an advantage over Zippel's algorithm.

We have implemented both Zippel's algorithm and the new algorithm in C and we have done a parallel implementation of our algorithm using Cilk [2]. In the paper we provide benchmarks comparing the number of probes our algorithm does with both Zippel's algorithm and Kaltofen and Lee's hybrid of the Zippel and Ben-Or/Tiwari algorithms.

Categories and Subject Descriptors:
I.1.2 [Symbolic and Algebraic Manipulation]: Algorithms

General Terms: Algorithms, Theory.

Keywords: sparse polynomial interpolation, parallel interpolation algorithms, Ben-Or Tiwari.

1. INTRODUCTION

Let p be a prime and let $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ be a multivariate polynomial with $t > 0$ non-zero terms which is represented with a *black box* $\mathbb{Z}_p^n \rightarrow \mathbb{Z}_p$. On input $(\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}_p^n$, the black box evaluates and outputs $f(x_1 = \alpha_1, \ldots, x_n = \alpha_n)$. Given also a degree bound d on the degree of f, our goal is to interpolate the polynomial f with minimum number of evaluations (probes to the black box).

Sparse interpolation is a key part of many algorithms in computer algebra such as polynomial GCD computation [17, 7, 13] over $\mathbb{Z}$. Here one computes the GCD modulo p where p is chosen to be a machine size prime. We are interested in algorithms whose computational complexity is polynomial in t, n, d and $\log p$. In 1979 Richard Zippel presented the first such algorithm. Zippel's algorithm is probabilistic. It relies heavily on the assumption that if a polynomial is zero at a *random* evaluation point, then it is the zero polynomial with high probability. Zippel's algorithm interpolates f one variable at a time, sequentially. It makes $O(ndt)$ probes to the black box. In 1990, Zippel in [18] improved his 1979 algorithm to use evaluation points of the form $(\alpha_1^i, \ldots, \alpha_k^i) \in \mathbb{Z}_p^k$ so that the linear systems to be solved become transposed Vandermonde systems which can be solved in $O(t^2)$ time instead of $O(t^3)$ – see [8].

In 1988, Ben-Or and Tiwari [1] presented a deterministic algorithm for interpolating a multivariate polynomial with integer, rational, real or complex coefficients. Given a bound T on the number of terms t of the polynomial f, the algorithm evaluates the black box at powers of the first n primes; it evaluates at the points $(2^i, 3^i, 5^i, \ldots, p_n^i)$ for $0 \le i < 2T$. If $M_j(x_1, \ldots, x_n)$ are the monomials of the t non-zero terms of f, it then uses Berlekamp/Massey algorithm [12] from coding theory to find the monomial evaluations $M_j(2, 3, 5, \ldots, p_n)$ for $1 \le j \le t$ and then determines the degree of each monomial M_j in x_k by trial division of $M_j(2, 3, 5, \ldots, p_n)$ by p_k. This algorithm is not variable by variable. Instead, it interpolates the polynomial f with $2T$ probes to the black box which can all be computed in parallel. The major disadvantage of the Ben-Or/Tiwari algorithm is that the evaluation points are large ($O(T \log n)$ bits long – see [1]) and computations over $\mathbb{Q}$ encounter an expression swell which makes the algorithm very slow. This problem was addressed by Kaltofen *et al.* in [9] by running the algorithm modulo a power of a prime of sufficiently large size; the modulus must be greater than $\max_j M_j(2, 3, 5, \ldots, p_n)$.

In [6], Huang and Rao describe how to make the Ben-Or/Tiwari approach work over finite fields GF(q) with at least $4t(t-2)d^2 + 1$ elements. Their idea is to replace the primes $2, 3, 5, \ldots, p_n$ in Ben-Or/Tiwari by linear (hence irreducible) polynomials in GF$(q)[y]$. Their algorithm is Las Vegas and does $O(dt^2)$ probes. Although the authors discuss how to parallelize the algorithm, the factor of t^2 may limit this approach.

In 2000, Kaltofen et al. in [10, 11] design a hybrid algorithm, a hybrid of the Zippel and Ben-Or Tiwari algorithms, which they call a "racing algorithm". To reduce the number

*This work was supported by NSERC of Canada and the MITACS NCE of Canada

of probes when interpolating the next variable in Zippel's algorithm, their algorithm runs a Newton interpolation and a univariate Ben-Or/Tiwari simultaneously, stopping when the first succeeds. However, this further sequentializes the algorithm. In Section 5, we compare the number of probes made by this algorithm to our new algorithm.

In 2009, Giesbrecht, Labahn and Lee in [4] present two new algorithms for sparse interpolation for polynomials with floating point coefficients. The first is a modification of the Ben-Or/Tiwari algorithm that uses $O(t)$ probes. In principle, this algorithm can be made to work over finite fields $GF(q)$ for applications where one can choose q. One needs $q-1$ to have n distinct prime factors all $> d$. One would also need $q-1$ to have no large prime factors so that the discrete logarithms needed could be done efficiently using the Pohlig-Hellman algorithm [14]. We have not explored the feasibility of this approach.

Our approach for sparse interpolation over $\mathbb{Z}_p$ is to use evaluation points of the form $(\alpha_1^i, \ldots, \alpha_n^i) \in \mathbb{Z}_p^n$ and modify the Ben-Or/Tiwari algorithm to do extra probes to determine the degrees of the variables in each monomial in f. We do a factor of at most $2n$ more evaluations in order to recover the monomials from their images. A motivation for our new algorithm is to use the Ben-Or/Tiwari approach in modular algorithms (e.g. GCD computations in characteristic 0 – see [7]) where the prime p is chosen to be a machine prime so that arithmetic in $\mathbb{Z}_p$ is efficient.

Our paper is organized as follows. In Section 2 we present an example showing the main flow and the key features of our algorithm. We then identify possible problems that can occur and how the new algorithm deals with them in Section 3. In Section 4 we present our new algorithm and analyze its sequential time complexity. Finally, in Section 5 we compare the C implementations of our algorithm and Zippel's algorithm with the racing algorithm of Kaltofen and Lee [11] on various sets of polynomials.

2. THE IDEA AND AN EXAMPLE

Let $f = \sum_{i=1}^{t} a_i M_i \in \mathbb{Z}_p[x_1, \ldots, x_n]$ be the polynomial represented with the black box with $a_i \in \mathbb{Z}_p \backslash \{0\}$. Here t is the number of non-zero terms in f. $M_i = x_1^{e_{i1}} \times x_2^{e_{i2}} \times \cdots \times x_n^{e_{in}}$ is the i'th monomial in f where $M_i \neq M_j$ for $i \neq j$. Let $d \geq \deg f$ be a bound on the degree of f so that $e_{ij} \leq d$ for all $1 \leq i, j \leq n$.

We demonstrate our algorithm on the following example. Here we use x, y and z for variables instead of x_1, x_2 and x_3.

Example 1 *Let $f = 91yz^2 + 94x^2yz + 61x^2y^2z + 42z^5 + 1$ and $p = 101$. Given the number of terms $t = 5$, the number of variables $n = 3$, a degree bound $d = 5$ and the black box that computes f, we want to find f.*

The first step is to pick $n = 3$ generators $\alpha_1, \alpha_2, \alpha_3$ of $\mathbb{Z}_p^*$. We evaluate the black box at the points $\beta_0, \ldots, \beta_{2t-1}$ where $\beta_i = (\alpha_1^i, \alpha_2^i, \ldots, \alpha_n^i)$. Thus we make $2t$ probes to the black box. The reason to use generators instead of random values from $\mathbb{Z}_p$ is that it decreases the probability of two distinct monomials having the same evaluation. For our example, let the generators be $\alpha_1 = 66, \alpha_2 = 12$ and $\alpha_3 = 3$ and let v_i be the output of the black box on input β_i and let $V = (v_0, \ldots, v_{2t-1})$. In this example we obtain

$$V = (87, 78, 65, 41, 49, 38, 87, 29, 23, 86).$$

Now we use the Berlekamp/Massey algorithm [12] (See [10] for a more accessible reference). The input to this algorithm is a sequence of elements $b_0, b_1, \ldots, b_{2t-1}, \ldots$ where $b_i \in \mathbb{Z}_p$. The algorithm computes a linear generator *for the sequence, i.e. the univariate polynomial $\Lambda(z) = z^t - \lambda_{t-1}z^{t-1} - \cdots - \lambda_0$ such that*

$$b_{t+i} = \lambda_{t-1}b_{t+i-1} + \lambda_{t-2}b_{t+i-2} + \cdots + \lambda_0 b_i$$

for all $i \geq 0$. In our example the input is $V = (v_0, \ldots, v_{2t-1})$ and the output is

$$\Lambda_1(z) = z^5 + 28z^4 + 62z^3 + 54z^2 + 11z + 46.$$

The next step is to find the roots of $\Lambda_1(z)$. We know (see [1]) that this polynomial is the product of exactly $t = 5$ linear factors. The roots are $r_1 = 1, r_2 = 7, r_3 = 41, r_4 = 61$ and $r_5 = 64$. Ben-Or and Tiwari prove that for each $1 \leq i \leq t$, there exists $1 \leq j \leq t$ such that

$$m_i = M_i(\alpha_1, \ldots, \alpha_n) \equiv r_j \bmod p.$$

The main step now is to determine the degrees of each monomial in f in each variable. Consider the first variable x. Let α_{n+1} be a new random generator of $\mathbb{Z}_p^*$. In this example we choose $\alpha_4 = 34$. This time we choose the evaluation points $\beta'_0, \ldots, \beta'_{2t-1}$ where $\beta'_i = (\alpha_{n+1}^i, \alpha_2^i, \ldots, \alpha_n^i)$. Note that this time we are evaluating the first variable at powers of α_{n+1} instead of α_1. We evaluate the black box at these points and apply the Berlekamp/Massey algorithm on the sequence of the outputs to compute the linear generator for the new sequence

$$\Lambda_2 = z^5 + 45z^3 + 54z^2 + 60z + 42.$$

Let $\bar{r}_1, \ldots, \bar{r}_5$ be distinct roots of Λ_2.
We know that $M_i(\alpha_{n+1}, \alpha_2, \ldots, \alpha_n)$ is a root of Λ_2 for $1 \leq i \leq n$. On the other hand we have

$$\frac{M_i(\alpha_{n+1}, \alpha_2, \ldots, \alpha_n)}{M_i(\alpha_1, \alpha_2, \ldots, \alpha_n)} = (\frac{\alpha_{n+1}}{\alpha_1})^{e_{i1}}. \quad (1)$$

Let $r_j = M_i(\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\bar{r}_k = M_i(\alpha_{n+1}, \alpha_2, \ldots, \alpha_n)$. From Equation 1 we have

$$\bar{r}_k = r_j \times (\frac{\alpha_{n+1}}{\alpha_1})^{e_{i1}},$$

i.e. for every root r_j of Λ_1, $r_j \times (\frac{\alpha_{n+1}}{\alpha_1})^{e_{i1}}$ is a root of Λ_2 for some e_{i1} which is the degree of some monomial in f with respect to x. This gives us a way to compute the degree of each monomial M_i in the variable x. In this example we have $\frac{\alpha_{n+1}}{\alpha_1} = 25$. We start with the first root of Λ_1 and check if $r_1 \times (\frac{\alpha_{n+1}}{\alpha_1})^i$ is a root of Λ_2 for $0 \leq i \leq d$. For $r_1 = 1$ we have that $r_1 \times (\frac{\alpha_{n+1}}{\alpha_1})^0$ is a root of Λ_2 and for $0 < i \leq d$, $r_1 \times (\frac{\alpha_{n+1}}{\alpha_1})^i$ is not a root of Λ_2, hence we conclude that the degree of the first monomial of f in x is 0. We continue this to find the degrees of all the monomials in f in the variable x. We obtain

$$e_{11} = 0, e_{21} = 0, e_{31} = 0, e_{41} = 2, e_{51} = 2.$$

We proceed to the next variable y. This time we evaluate the black box at $\beta''_0, \ldots, \beta''_{2t-1}$ where $\beta''_i = (\alpha_1^i, \alpha_{n+1}^i, \alpha_3^i, \ldots, \alpha_n^i)$ and apply the Berlekamp/Massey algorithm on the sequence of the outputs to compute

$$\Lambda_3 = z^5 + 5z^4 + 27z^3 + 36z^2 + 93z + 40$$

the linear generator for the new sequence. Let $\tilde{r}_1, \dots, \tilde{r}_5$ be distinct roots of Λ_3. Again using the same approach as above, we find that the degrees of the monomials in the second variable y to be

$$e_{12} = 0, e_{22} = 1, e_{32} = 0, e_{42} = 2, e_{52} = 1.$$

Finally we proceed to the last variable z. This time we evaluate z at powers of α_{n+1} instead of α_3 and compute the following linear generator for the sequence of outputs obtained by probing the black box

$$\Lambda_4 = z^5 + 27z^4 + 99z^3 + 18z^2 + 16z + 41.$$

We compute the degrees with the same technique and obtain

$$e_{13} = 0, e_{23} = 2, e_{33} = 5, e_{43} = 1, e_{53} = 1.$$

The reader may observe that determining the degrees of the monomials M_i in each variable represent n independent tasks which can therefore be done in parallel. At this point we have computed all the monomials. Recall that $M_i = x_1^{e_{i1}} \times x_2^{e_{i2}} \times \dots \times x_n^{e_{in}}$ hence we have

$$M_1 = 1, M_2 = yz^2, M_3 = z^5, M_4 = x^2y^2z \text{ and } M_5 = x^2yz.$$

Now we need to compute the coefficients. We do this by solving one linear system of equations. We computed the roots of Λ_1 and we have computed the monomials such that $M_i(\alpha_1, \dots, \alpha_n) = r_i$. Recall that v_i is the output of the black box on the input $\beta_i = (\alpha_1^i, \dots, \alpha_n^i)$ hence we have

$$v_i = a_1 r_1^i + a_2 r_2^i + \dots + a_t r_t^i$$

for $0 \le i \le 2t-1$. Note that the system of equations obtained from the above set of equations is a Vandermonde system which can be solved in $O(t^2)$ time and $O(t)$ space (See [18]). After solving we obtain

$$a_1 = 1, a_2 = 91, a_3 = 42, a_4 = 61 \text{ and } a_5 = 94$$

and hence $f = 1 + 91yz^2 + 42z^5 + 61x^2y^2z + 94x^2yz$ is interpolated and we are done.

3. PROBLEMS

The evaluation points $\alpha_1, \dots, \alpha_n, \alpha_{n+1}$ must satisfy certain conditions for our new algorithm to work properly. Here we identify all problems.

3.1 Distinct Monomials

The first condition is that for $i \neq j$

$$M_i(\alpha_1, \dots, \alpha_n) \neq M_j(\alpha_1, \dots, \alpha_n) \text{ in } \mathbb{Z}_p$$

so that $\deg(\Lambda_1(z)) = t$. Also, at the k'th step of the algorithm, when computing the degrees of the monomials in x_k, we must have

$$\forall\ 1 \le i \neq j \le t, \quad m_{i,k} \neq m_{j,k} \text{ in } \mathbb{Z}_p$$

where $m_{i,k} = M_i(\alpha_1, \dots, \alpha_{k-1}, \alpha_{n+1}, \alpha_{k+1}, \dots, \alpha_n)$ so that $\deg(\Lambda_{k+1}(z)) = t$. To reduce the probability of monomial evaluations colliding, we pick α_i to have order $> d$. The easiest way to do this is to use generators of $\mathbb{Z}_p^*$. There are $\phi(p-1)$ generators where ϕ is Euler's totient function. We now give an upper bound on the probability that no monomial evaluations collide when we use generators for evaluations.

Theorem 1 *Let $\alpha_1, ..., \alpha_n$ be generators from $\mathbb{Z}_p$ chosen at random and let $m_i = M_i(\alpha_1, \dots, \alpha_n)$. Then the probability that two or more monomials evaluate to the same value (we get a collision) is*

$$\le \binom{t}{2} \frac{d}{\phi(p-1)} < \frac{dt^2}{2\phi(p-1)}.$$

PROOF. Consider the polynomial

$$A = \prod_{1 \le i < j \le t} (M_i(x_1, \dots, x_n) - M_j(x_1, \dots, x_n)).$$

Observe that $A(\alpha_1, \dots, \alpha_n) = 0$ iff two monomial evaluations collide. Recall that the Schwartz-Zippel lemma ([16, 17]) says that if $r_1, \dots, r_n$ are chosen at random from any subset S of a field K and $F \in K[x_1, \dots, x_n]$ is non-zero then

$$\text{Prob}(F(r_1, \dots, r_n) = 0) \le \frac{\deg f}{|S|}.$$

Our result follows from noting that $d \ge \deg f$ and thus $\deg A \le \binom{t}{2} d$ and $|S| = \phi(p-1)$, the number of primitive elements in $\mathbb{Z}_p$. $\square$

3.2 Root Clashing

Let $r_1, \dots, r_t$ be the roots of $\Lambda_1(z)$ which is the output of the Berlekamp/Massey algorithm on the sequence of the outputs from the black box on the first set of evaluation points $\alpha_1, \dots, \alpha_n$. Suppose at the k'th step, we want to compute the degrees of all the monomials in the variable x_k. As mentioned in the Example 1, the first step is to compute Λ_{k+1}. Then if $\deg_{x_k}(M_i) = e_{ik}$ we have $\bar{r}_i = r_i \times (\frac{\alpha_{n+1}}{\alpha_k})^{e_{ik}}$ is a root of Λ_{k+1}. If $r_i \times (\frac{\alpha_{n+1}}{\alpha_k})^{e'}$, $0 \le e' \neq e_{ik} \le d$ is also a root of Λ_{k+1} then we may not be able to uniquely identify the correct degree of the i'th monomial in the k'th variable x_k. We will illustrate this with an example.

Example 2 *Consider the polynomial given in Example 1. Suppose instead of choosing $\alpha_4 = 34$, we choose $\alpha_4 = 72$ which is another generator of $\mathbb{Z}_p^*$. Since α_1, α_2 and α_3 are the same as before, Λ_1 does not change and hence the roots of Λ_1 are $r_1 = 1, r_2 = 7, r_3 = 41, r_4 = 61$ and $r_5 = 64$. In the next step we substitute $\alpha_4 = 72$ for α_1 and compute $\Lambda_2 = z^5 + 61z^4 + 39z^3 + 67z^2 + 37z + 98$. We proceed to compute the degrees of the monomials in x but we find that*

$$r_4 \times (\frac{\alpha_4}{\alpha_1})^2 = 15 \text{ and } r_4 \times (\frac{\alpha_4}{\alpha_1})^4 = 7$$

are both roots of Λ_2 and hence we can not decide the correct degree of the last monomial in x.

Theorem 2 *If $\deg \Lambda_1(z) = \deg \Lambda_{k+1}(z) = t$ then the probability that we cannot uniquely compute the degrees of all $M_i(x_1, \dots, x_n)$ in x_k is at most $\frac{d^2t^2}{4\phi(p-1)}$.*

PROOF. Let $S_i = \{r_j \times (\frac{\alpha_{n+1}}{\alpha_k})^i \mid 1 \le j \le t\}$ for $0 \le i \le d$. We assume that $r_i \neq r_j$ for all $1 \le i \neq j \le t$. We will not be able to uniquely identify the degree of the j'th monomial in x_k if there exists $\bar{d}$ such that $r_j \times (\frac{\alpha_{n+1}}{\alpha_k})^{\bar{d}} = \bar{r}_i$ is a root of $\Lambda_{k+1}(z)$ and $0 \le d \neq e_{jk} \le d$ where e_{jk} is $\deg_{x_k}(M_j)$. But we have $\bar{r}_i = r_i \times (\frac{\alpha_{n+1}}{\alpha_k})^{e_{ik}}$ thus $r_j \times (\frac{\alpha_{n+1}}{\alpha_k})^{\bar{d}} = r_i \times (\frac{\alpha_{n+1}}{\alpha_k})^{e_{ik}}$. Without loss of generality, assume $\tilde{d} = \bar{d} - e_{ik} > 0$. We have $r_i = r_j \times (\frac{\alpha_{n+1}}{\alpha_k})^{\tilde{d}}$ and hence

$r_i \in S_{\tilde{d}} \Rightarrow S_0 \cap S_{\tilde{d}} \neq \emptyset$. Hence we will not be able to compute the degrees in x_k if $S_0 \cap S_i \neq \emptyset$ for some $1 \le i \le d$. Let

$$g(x) = \prod_{1 \le l \neq j \le t} (r_j x^i - r_l \alpha_k^i).$$

We have $r_l = r_j \times (\frac{\alpha_{n+1}}{\alpha_k})^i \in S_0 \cap S_i$ iff $g(\alpha_{n+1}) = 0$. Using the Schwartz-Zippel lemma, the probability that $g(\alpha_{n+1}) = 0$ is at most $\frac{\deg g}{\phi(p-1)} = \frac{\binom{t}{2} i}{\phi(p-1)} < \frac{it^2}{2\phi(p-1)}$. If we sum this quantity for all $1 \le i \le d$ we obtain that the overall probability is at most $\frac{d^2t^2}{4\phi(p-1)}$. $\square$

Using Theorem 2, the probability that we will not be able to uniquely identify the degrees of the monomials in *all* the variables is at most $\frac{nd^2t^2}{4\phi(p-1)}$, i.e. for p, s.t. $\phi(p-1) > \frac{nd^2t^2}{2}$ with probability at least half, the algorithm succeeds without dealing with any problem. We will now discuss our solution to this problem. Note that we assume the images of the monomials are distinct, i.e. $\forall\ 1 \le i \neq j \le t,\ m_{i,k} \neq m_{j,k}$. Suppose we have computed Λ_{k+1} and we want to compute the degrees of the monomials in x_k and let $R_1 = \{r_1, \ldots, r_t\}$ be the set of all the roots of Λ_1 and $R_k = \{\bar{r}_1, \ldots, \bar{r}_t\}$ be the set of all the distinct roots of Λ_{k+1}. Let

$$D_j = \{(i, r) \mid 0 \le i \le d,\ r = r_j \times (\frac{\alpha_{n+1}}{\alpha_1})^i \in R_k\}.$$

D_j contains the set of all possible degrees of the j'th monomial M_j in the k'th variable x_k. We know that $(e_{jk}, \bar{r}_j) \in D_j$ and hence $|D_j| \ge 1$. If $|D_j| = 1$ for all $1 \le j \le t$, then the degrees are unique and this step of the algorithm is complete. Let G_k be a balanced bipartite graph defined as follows. G_k has two independent sets of nodes U and V each of size t. Nodes in U and V represent elements in R_1 and R_k respectively, i.e. $u_i \in U$ and $v_j \in V$ are labeled with r_i and $\bar{r}_j$. We connect $u_i \in U$ to $v_j \in V$ with an edge of weight (degree) d_{ij} if and only if $(d_{ij}, \bar{r}_j) \in D_i$.

Lemma 1 *We can uniquely identify the degrees of all the monomials in x_k, if and only if the bipartite graph G_k has a unique* perfect *matching.*

The proof of this lemma is immediate by looking at the structure of the graph G_k. We illustrate with an example.

Example 3 *Let f be the polynomial given in Example 1 and suppose for some evaluation points $\alpha_1, \ldots, \alpha_4$ we obtain the graph G_1 as shown in Figure 1. This graph has a perfect matching, i.e. the set of edges $\{(r_i, \bar{r}_i) \mid 1 \le i \le 5\}$. If there was an edge connecting r_1 to $\bar{r}_2$ then the new graph would no longer have a unique perfect matching and we would fail to uniquely compute the degrees of monomials in x.*

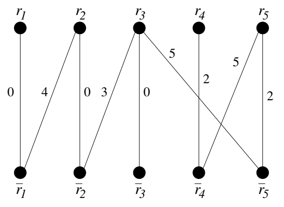


Figure 1: The bipartite graph G_1

We now give a solution for the case where G_k does not have a unique perfect matching for some $1 \le k \le n$. The solution involves $2t$ more probes to the black box. Suppose we choose a random element $\alpha_{n+2} \in \mathbb{Z}_p$ such that $\gamma = \frac{\alpha_{n+2}}{\alpha_{n+1}}$ is a generator of $\mathbb{Z}_p^*$ (or is of order greater than d). Let $\beta_i = (\alpha_1^i, \ldots, \alpha_{k-1}^i, \alpha_{n+2}^i, \alpha_{k+1}^i, \ldots, \alpha_n^i)$ and let v_i be the output of the black box on input β_i $(0 \le i \le 2t-1)$. On input $V = (\beta_0, \ldots, \beta_{2t-1})$, the Berlekamp/Massey algorithm computes a linear generator $\Lambda'_{k+1}(z)$ for V. Let $\{\tilde{r}_1, \ldots, \tilde{r}_t\}$ be the set of distinct roots of Λ'_{k+1}. Let G'_k be the balanced bipartite graph, obtained from Λ_1 and Λ'_{k+1}.

Definition 1. We define $\bar{G}_k$, the intersection of G'_k and G_k, as follows. $\bar{G}_k$ has the same nodes as G'_k and there is an edge between r_i and $\tilde{r}_j$ with weight (degree) d_{ij} if and only if r_i is connected to $\bar{r}_j$ in G_k and to $\tilde{r}_j$ in G'_k, both with the same degree d_{ij}.

Lemma 2 *Let $e_{ij} = \deg_{x_j}(M_i)$. The two nodes r_i and $\tilde{r}_i$ are connected in $\bar{G}_k$ with degree e_{ij}.*

We take advantage of the following theorem which implies we need at most one extra set of probes.

Theorem 3 *Let $\bar{G}_k = G_k \cap G'_k$. $\bar{G}_k$ has a* unique *perfect matching.*

Proof. Let U and V be the set of independent nodes in $\bar{G}_k$ such that $u_i \in U$ and $v_j \in V$ are labeled with r_i and $\tilde{r}_j$ respectively where $\tilde{r}_j$ is a root of Λ'_{k+1}. We will prove that each node in V has degree exactly 1 and hence there is a unique perfect matching. The proof is by contradiction. Suppose the degree of $v_j \in V$ is at least 2. With out loss of generality assume that r_1 and r_2 are both connected to $\tilde{r}_j$ with degrees d_{1j} and d_{2j} respectively (See Figure 2).

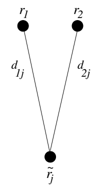


Figure 2: Node $\tilde{r}_j$ of graph $\bar{G}_k$

Using Definition 1 we have

$$\bar{r}_j = r_1 \times (\frac{\alpha_{n+1}}{\alpha_k})^{d_{1j}} = r_2 \times (\frac{\alpha_{n+1}}{\alpha_k})^{d_{2j}} \text{ and}$$

$$\tilde{r}_j = r_1 \times (\frac{\alpha_{n+2}}{\alpha_k})^{d_{1j}} = r_2 \times (\frac{\alpha_{n+2}}{\alpha_k})^{d_{2j}}.$$

Dividing the two sides of these equations results in

$$(\frac{\alpha_{n+2}}{\alpha_{n+1}})^{d_{1j}} = (\frac{\alpha_{n+2}}{\alpha_{n+1}})^{d_{2j}}.$$

Since we chose α_{n+2} such that $\frac{\alpha_{n+2}}{\alpha_{n+1}}$ has a sufficiently large order (greater than the degree bound d) we have $d_{1j} = d_{2j} \Rightarrow r_1 = r_2$. But this is a contradiction because both r_1 and r_2 are roots of Λ_1 which we assumed are distinct. $\square$

Lemma 2 and Theorem 3 prove that the intersection of G_k and G'_k will give us the correct degrees of all the monomials in the k'th variable x_k. We will illustrate with an example.

Example 4 *Let $f = -10\,y^3 - 7\,x^2yz - 40\,yz^5 + 42\,y^3z^5 - 50\,x^7z^2 + 23\,x^5z^4 + 75\,x^7yz^2 - 92\,x^6y^3z + 6\,x^3y^5z^2 + 74\,xyz^8 + 4$ and $p = 101$. We choose the first set of evaluation points to be $\alpha_1 = 66, \alpha_2 = 11, \alpha_3 = 48$ and $\alpha_4 = 50$. For the first variable x we will obtain the bipartite graph G_1 shown in Figure 3.*

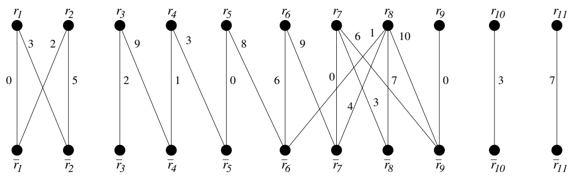


Figure 3: The bipartite graph G_1

This graph does not have a unique perfect matching, so we proceed to choose a new evaluation point $\alpha_5 = 89$. This time we will get the bipartite graph G_1' shown in Figure 4.

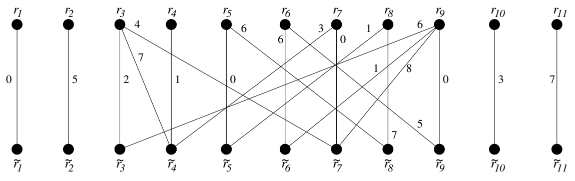


Figure 4: The bipartite graph G_1'

Again G_1' does not have a unique perfect matching. We compute the intersection of G_1 and G_1': $\bar{G}_1 = G_1 \cap G_1'$. $\bar{G}_1$ is shown in Figure 5.

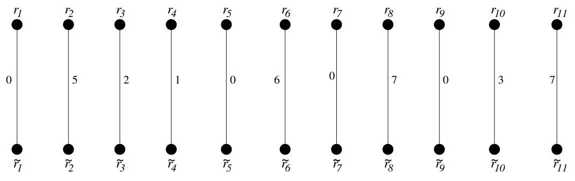


Figure 5: The bipartite graph $\bar{G}_1$

As stated by Theorem 3, $\bar{G}_1$ has a unique perfect matching and the degree of every monomial in x is correctly computed.

In this section we proved that if the prime p is sufficiently large ($\phi(p-1)$ must be approximately dt^2 for us to be able to get distinct images of monomials with reasonable probability), we will be able to compute the degrees of all the t monomials in each variable x_k using up to $4t$ evaluation points. If the graph G_k has a unique perfect matching, we will be able to compute the degrees in x_k with only $2t$ probes to the black box.

We conclude this section with the following lemma which we will later use in Section 4.

Lemma 3 *Let G_k be the bipartite graph for the k'th variable. Let $u_{i_1} \to v_{j_1} \to u_{i_2} \to v_{j_2} \to \cdots \to v_{j_s} \to u_{i_1}$ be a cycle in G_k where $u_l \in U$ is labeled with r_l (a root of Λ_1) and $v_m \in V$ is labeled with $\bar{r}_m$ (a root of Λ_{k+1}). Let d_{lm} be the weight (degree) of the edge between u_l and v_m. We have $\sum_{m=1}^{s} d_{i_m j_m} - \sum_{m=1}^{s} d_{i_{m+1} j_m} = 0$.*

Proof. It is easy to show that $r_{i_1} = (\frac{\alpha_{n+1}}{\alpha_k})^{\bar{d}} r_{i_s}$ where $\bar{d} = d_{i_1j_1} - d_{i_2j_1} + d_{i_2j_2} - d_{i_3j_2} + \cdots + d_{i_{s-1}j_{s-1}} - d_{i_sj_{s-1}}$. Also both u_{i_1} and u_{i_s} are connected to v_{j_s} in G_k hence we have $r_{i_1} = (\frac{\alpha_{n+1}}{\alpha_k})^{d_{i_1j_s}} \bar{r}_{i_s}$ and $r_{i_s} = (\frac{\alpha_{n+1}}{\alpha_k})^{d_{i_sj_s}} \bar{r}_{i_s}$. These three equations yield to $r_{i_1} = (\frac{\alpha_{n+1}}{\alpha_k})^{\tilde{d}} r_{i_1}$ where $\tilde{d} = d_{i_1j_1} - d_{i_2j_1} + d_{i_2j_2} - d_{i_3j_2} + \cdots + d_{i_{s-1}j_{s-1}} - d_{i_sj_{s-1}} + d_{i_sj_s} - d_{i_1j_s}$. But if $\frac{\alpha_{n+1}}{\alpha_k}$ is of sufficiently high order, $\tilde{d}$ must be zero thus $\sum_{m=1}^{s} d_{i_m j_m} - \sum_{m=1}^{s} d_{i_{m+1} j_m} = 0$. $\square$

Example 5 *In G_1' shown in Figure 4, there is a cycle $r_3 \to \bar{r}_4 \to r_7 \to \bar{r}_7 \to r_3$. The weights (degrees) of the edges in this cycle are as $7, 3, 0$ and 4. We have $7 - 3 + 0 - 4 = 0$.*

4. THE ALGORITHM

Algorithm: Interpolation

Input: A *black box* $\mathbf{B} : \mathbb{Z}_p^n \to \mathbb{Z}_p$ that on input $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}_p^n$ outputs $f(\alpha_1, \ldots, \alpha_n)$ where $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$.
Input: A degree bound $d \geq \deg(f)$.
Input: A bound $T \geq t$ on the number of terms in f.
Output: The polynomial f or FAIL.

```
 1: Choose $n+1$ generators $\alpha_1 \neq \ldots \neq \alpha_{n+1}$ of $\mathbb{Z}_p^*$ randomly.
 2: repeat choose $\gamma$ to be a random generator of $\mathbb{Z}_p^*$ and let
    $\alpha_{n+2} = \alpha_{n+1} \times \gamma$ until $\alpha_{n+2} \notin \{\alpha_1, \ldots, \alpha_{n+1}\}$.
 3: Let $\beta_i = (\alpha_1^i, \ldots, \alpha_n^i)$ for $0 \leq i \leq 2T-1$.
 4: for $k$ from 1 to $n+1$ in parallel do
 5:    Compute $\Lambda_k(z)$:
 6:    Compute $v_i = \mathbf{B}(\beta_i)$ for $0 \leq i \leq 2t-1$ using $\alpha_{n+1}^i$ instead
       of $\alpha_{k-1}^i$ when $k > 1$.
 7:    Use the Berlekamp/Massey algorithm to compute a linear
       generator $\Lambda_k \in \mathbb{Z}_p[z]$ for the sequence $v_0, \ldots, v_{2t+1}$.
 8: end for
 9: Set $t = \max(\deg \Lambda_1(z), \ldots \Lambda_{n+1}(z))$. If the degree of the $\Lambda$'s
    are not all equal to $t$ then repeat steps 1 through step 8 once.
    If this does not yield equal degree $\Lambda$'s then ($p$ is likely too
    small so) return FAIL.
10: Compute $\{r_1, \ldots, r_t\}$ the set of distinct roots of $\Lambda_1(z)$.
11: for $k$ from 1 to $n$ in parallel do
12:    Determine $\deg_{x_k}(M_i)$ for $1 \leq i \leq t$:
13:    Construct the graph $G_k$ as described in Section 3.
14:    if $G_k$ has a unique perfect matching then
15:       Set $e_{ik} = d_{il}$ where $d_{il}$ is the weight (degree) of the edge
          that matches the node $r_i$ to $\bar{r}_l$ in the perfect matching.
16:    else
17:       Construct the graph $G_k'$ as described in Section 3.
          Note, this requires $2t$ more probes to $\mathbf{B}$.
18:       Find the intersection of $G_k$ and $G_k'$: $\bar{G}_k = G_k \cap G_k'$.
19:       Set $e_{ik} = d_{il}$ where $d_{il}$ is the weight (degree) of the edge
          that matches the node $r_i$ to $\bar{r}_l$ in the perfect matching
          of graph $\bar{G}_k$.
20:    end if
21: end for
22: Let $S = \{a_1 r_1^i + a_2 r_2^i + \cdots + a_t r_t^i = v_i \mid 0 \leq i \leq 2t'-1\}$.
    Solve the linear system $S$ for $(a_1, \ldots, a_t) \in \mathbb{Z}_p^t$.
23: Let $g = \sum_{i=1}^{t} a_i M_i$ where $M_i = \prod_{j=1}^{n} x_j^{e_{ij}}$.
24: Pick non-zero $a_1, \ldots, a_n$ from $\mathbb{Z}_p$ at random.
    If $\mathbf{B}(a_1, \ldots, a_n) \neq g(a_1, \ldots, a_n)$ then return FAIL.
25: return $g$.
```

Remark 1 The algorithm is probabilistic. If the degrees of the $\Lambda's$ are all equal to t then the algorithm will compute f with probability 1. If the degrees of the $\Lambda's$ are all equal but less than t then the algorithm cannot compute f; that is, $g \neq f$. The check in step 24 detects incorrect g with probability at least $1 - d/(p-1)$ (the Schwartz-Zippel lemma). Thus by doing one additional probe to the black box, we verify the output g with high probability. Kaltofen and Lee in [11] also use additional probes to verify the output this way.

Remark 2 For simplicity, our presentation of the algorithm assumes the term bound T is good. In applications where a good term bound is not available, one should first compute $\Lambda_1(z)$ using T, and then use $t = \deg \Lambda_1(z)$ when computing $\Lambda_2, \ldots, \Lambda_{n+1}$.

4.1 Complexity Analysis

We now discuss the sequential complexity of the algorithm assuming $t = T$. We need to consider the cost of probing the black box. Let $E(n, t, d)$ be the cost of one probe to the black box. If G_k has a unique perfect matching for $1 \leq k \leq n$ then we can correctly compute the degrees using only G_k. In this case the total number of probes is $2(n+1)t$ in the first loop. In the worst case where G_k does not have a unique perfect matching for all $1 \leq k \leq n$, we need to do additional $2nt$ probes to the black box in the second loop to construct all G'_k graphs. In this case the total number of probes to the black box is $2(n+1)t + 2nt = 2(2n+1)t$. Hence the total cost of probes to the black box is $O(ntE(n, t, d))$.

The $n+1$ calls to the Berlekamp/Massey algorithm in the first loop (as presented in [10]) cost $O(t^2)$ time each. The Vandermonde system of equations at Step 22 can be solved in $O(t^2)$ using the technique given in [18]. Note that as mentioned in [18], when inverting a $t \times t$ Vandemonde matrix defined by $k_1, \ldots, k_t$, one of the most expensive parts of this technique is to compute the master polynomial $M(z) = \prod_{i=1}^{t}(z - k_i)$. However, in our algorithm we can use the fact that $M(z) = \prod_{i=1}^{t}(z - r_i) = \Lambda_1(z)$.

To compute the roots of $\Lambda_1(z)$ at Step 10 of the algorithm, we use Rabin's Las Vegas algorithm [15]. If $f \in \mathbb{Z}_p[z]$ is a product of linear factors, Rabin's algorithm tries to split it into two factors of lower degree by computing the $\gcd((z - \beta)^{(p-1)/2} - 1, \Lambda_1(z))$ for randomly chosen $\beta \in \mathbb{Z}_p$. Since $\deg_z(\Lambda_1) = t$, the cost of finding the t roots of $\Lambda_1(z)$, assuming classical algorithms for polynomial arithmetic in $\mathbb{Z}_p[z]$ are used, is $O(t^2 \log p)$. See Algorithm 14.15 of [3].

We can compute the information needed to construct the bipartite graph G_k in $O(dt^2)$ time. This involves evaluating $\Lambda_{k+1}(z)$ at d points for each monomial and testing if it is zero or not. Also computing the intersection of G_k and G'_k can be done in $O(td \log d)$ time. This is because we know that each node in the intersection is of degree one (See proof of Theorem 3). Thus the overall time complexity is

$$O(t^2(\log(p) + nd) + ntE(n, t, d)).$$

Remark 3 The algorithm, as presented, corresponds to our parallel implementation in Cilk. Further parallelism in the algorithm could be exploited. For example, one could compute all probes to the black box **B** in step 6 and step 17 in parallel. When determining the degree of the monomials in step 13 and 17, one can parallelize the evaluations of $\Lambda_{k+1}(z)$. The most expensive sequential component is the computation of the roots of $\Lambda_1(z)$ in step 10 which has complexity $O(t^2 \log p)$. With asymptotically fast arithmetic this is $\tilde{O}(t \log p)$.

4.2 Optimizations

Let $D = \deg(f)$. If the prime p is large enough, i.e. $p > \frac{nD^2t^2}{4\epsilon}$ then with probability $1 - \epsilon$ the degree of every monomial in x_k can correctly be computed using only G_k and without needing any extra probes to the black box. In fact in this case, with high probability, every r_i will be matched with exactly only one $\bar{r}_j$ and hence every node in G_k would have degree one (e.g. see Figure 5). But if $d \gg D$, i.e. the degree bound d is not tight, the probability that we could identify the degrees uniquely drops significantly even though p is large enough. This is because the probability that *root clashing* (see Section 3) happens, linearly depends on d. In this case, with probability $1 - \epsilon$, the degree of M_i in x_k would be $\min\{d_{ij} \mid (d_{ij}, r_i) \in G_k\}$, i.e. the edge connected to r_i in G_k with minimum weight (degree) is our desired edge in the graph which will show up in the perfect matching. We apply the following theorem.

Theorem 4 *Let H_k be a graph obtained by eliminating all edges connected to r_i in G_k except the one with minimum weight (degree) for all $1 \leq i \leq t$. If the degree of every node in H_k is exactly one, then e_{ik} is equal to the weight of the edge connected to r_i in H_k.*

This theorem can be proved using Lemma 3 and the fact that there can not be any cycle in the graph H_k. We will give an example.

Example 6 *Let $f = 25y^2z + 90yz^2 + 93x^2y^2z + 60y^4z + 42z^5$. Here $t = 5, n = 3, d_{max} = 5$ and $p = 101$. We choose the following evaluation points $\alpha_1 = 85, \alpha_2 = 96, \alpha_3 = 58$ and $\alpha_4 = 99$. Suppose we want to construct G_2 in order to compute the degrees of the monomials in y. Suppose our degree bound is $d = 40$ which is not tight. The graph G_2 and H_2 are shown in Figures 6 and 7 respectively.*

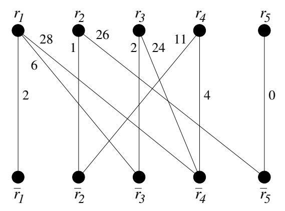


Figure 6: The bipartite graph G_2

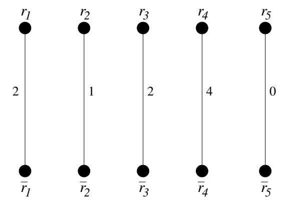


Figure 7: The bipartite graph H_2

The graph H_2 has the correct degrees of the monomials in variable y.

Theorem 4 suggests the following optimization. In the construction of the bipartite graph G_k, connect r_i to $\bar{r}_j$ with degree d_{ij} *only if* there is no $\bar{d} < d_{ij}$ such that $r_i \times (\frac{\alpha_{n+1}}{\alpha_k})^{\bar{d}}$ is a root of Λ_{k+1}, i.e. the degree of the node r_i in U is always one for all $1 \leq i \leq n$. If there is a perfect matching in this graph, this perfect matching is unique because this implies that the degree of each node $\bar{r}_j$ in V is also one (e.g. see Figure 7). If not, go back to and complete the graph G_k. This optimization makes our algorithm sensitive to the actual degree of $f(x_1, ..., x_n)$ in each variable.

The second optimization is to compute the degree of each monomial $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \dots x_n^{e_{in}}$ in the last variable x_n without doing any more probes to the black box. Suppose we have computed the degree of M_i in x_k for $1 \le k < n$. We know that $M_i(\alpha_1, \dots, \alpha_n)$ is equal to r_i, a root of Λ_1. Hence $r_i = \alpha_1^{e_{i1}} \cdot \alpha_2^{e_{i2}} \cdot \dots \cdot \alpha_n^{e_{in}}$. Since we know the degrees e_{ij} for $1 \le j < n$ we can determine e_{in} by division by α_n. This reduces the total number of probes from $4(n+1)t$ to $4nt$.

5. BENCHMARKS

Here, we compare the performance of our new algorithm, Zippel's algorithm and the racing algorithm of Kaltofen and Lee from [11]. We have implemented Zippel's algorithm and our new algorithm in C. We have also implemented an interface to call the interpolation routines from Maple. The racing algorithm is implemented in Maple in the ProtoBox package by Lee [11]. Since this algorithm is not coded in C, we only report (see columns labelled PBox) the number of probes it makes to the black box.

We give benchmarks comparing the performances on five problem sets. The polynomials in the first four benchmarks were generated at random. The fifth set of polynomials are taken from [11]. We count the number of probes to the black box and measure the total CPU time (for our new algorithm and Zippel's algorithm only). All the timings given in this section are in CPU seconds and were obtained using Maple 13 on a 64 bit Intel Core i7 920 @ 2.66GHz, running Linux. This is a 4 core machine. For our algorithm, we report the real time for 1 core and (in parentheses) 4 cores.

The black box in our benchmarks computes a multivariate polynomial with coefficients in $\mathbb{Z}_p$ where $p = 3037000453$ is a 31.5 bit prime. In all benchmarks, the black box simply evaluates the polynomial at the given evaluation point. To evaluate efficiently we compute and cache the values of x_i^j mod p in a loop in $O(nd)$. Then we evaluate the t terms in $O(nt)$. Hence the cost of one black box probe is $O(nd+nt))$ arithmetic operations in $\mathbb{Z}_p$.

Benchmark #1

This set of problems consists of 13 multivariate polynomials in $n = 3$ variables. The i'th polynomial ($1 \le i \le 13$) is generated at random using the following Maple command:

```
> randpoly([x1,x2,x3], terms = 2^i, degree = 30) mod p;
```

The i'th polynomial will have about 2^i non-zero terms. Here $D = 30$ is the total degree hence the maximum number of terms in each polynomial is $t_{max} = \binom{n+D}{D} = 5456$. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 30$. The timings and the number of probes are given in Table 1. In this table "DNF" means that the algorithm did not finish after 12 hours.
As i increases, the polynomial f becomes denser. For $i > 6$, f has more than $\sqrt{t_{max}}$ non-zero terms. This is indicated by a horizontal line in Table 1 and also in subsequent benchmarks. The line approximately separates sparse inputs from dense inputs. The last polynomial ($i = 13$) is 99.5% dense.

The data in Table 1 shows that for sparse polynomials $1 \le i \le 6$, our new algorithm does a lot fewer probes to the black box compared to Zippel's algorithm. It also does fewer probes than the racing algorithm (PBox). However, as the polynomials get denser, Zippel's algorithm has a better performance. For a completely dense polynomial with t non-zero terms, Zippel's algorithm only does $O(t)$ probes to the black box while the new algorithm does $O(nt)$ probes.

i	t	New Algorithm		Zippel		PBox
		Time	Probes	Time	Probes	Probes
1	2	0.00 (0.00)	12	0.00	217	20
2	4	0.00 (0.00)	24	0.00	341	39
3	8	0.00 (0.00)	48	0.00	558	79
4	16	0.00 (0.00)	96	0.01	868	156
5	32	0.00 (0.00)	192	0.01	1519	282
6	64	0.01 (0.01)	384	0.03	2573	517
7	128	0.03 (0.02)	768	0.08	4402	962
8	253	0.11 (0.06)	1518	0.21	6417	1737
9	512	0.44 (0.24)	3072	0.55	9734	3119
10	1015	1.66 (0.88)	6090	1.16	12400	5627
11	2041	6.50 (3.44)	12246	2.43	15128	DNF
12	4081	25.3 (13.4)	24486	4.56	16182	DNF
13	5430	44.3 (23.3)	32580	5.93	16430	DNF

Table 1: benchmark #1: $n = 3$ and $D = 30$

To show how effective the first optimization described in Section 4.2 is, we run both our algorithm and Zippel's algorithm on the same set of polynomials but with a bad degree bound $d = 100$. The timings and the number of probes are given in Table 2. One can see that our algorithm is unaffected by the bad degree bound; the number of probes and CPU timings are the same.

i	t	New Algorithm		Zippel's Algorithm	
		Time	Probes	Time	Probes
1	2	0.00 (0.00)	12	0.01	707
2	4	0.00 (0.00)	24	0.01	1111
3	8	0.00 (0.00)	48	0.02	1818
4	16	0.00 (0.00)	96	0.03	2828
5	32	0.00 (0.00)	192	0.07	4949
6	64	0.01 (0.01)	384	0.14	8383
7	128	0.04 (0.02)	768	0.36	14342
8	253	0.12 (0.07)	1518	0.79	20907
9	512	0.45 (0.24)	3072	1.97	31714
10	1015	1.67 (0.89)	6090	3.97	40400
11	2041	6.50 (3.45)	12246	8.18	49288
12	4081	25.3 (13.4)	24486	15.16	52722
13	5430	44.1 (23.4)	32580	19.62	53530

Table 2: benchmark #1: bad degree bound $d = 100$

Benchmark #2

In this set of benchmarks the i'th polynomial is in $n = 3$ variables and is generated at random in Maple using

```
> randpoly([x1,x2,x3], terms = 2^i, degree = 100) mod p;
```

This set of polynomials differs from the first benchmark in that the total degree of each polynomial is set to be 100 in the second set. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 100$. The timings and the number of probes are given in Table 3. Comparing this table to the data in Table 1 shows that the number of probes to the black box in our new algorithm does not depend on the degree of the target polynomial.

i	t	New Algorithm		Zippel		PBox
		Time	Probes	Time	Probes	Probes
1	2	0.00 (0.00)	12	0.01	707	19
2	4	0.00 (0.00)	24	0.01	1111	45
3	8	0.00 (0.00)	48	0.02	1919	89
4	16	0.00 (0.00)	96	0.04	3434	167
5	31	0.00 (0.00)	186	0.08	6161	320
6	64	0.02 (0.01)	384	0.19	10504	623
7	127	0.05 (0.02)	762	0.49	18887	1149
8	253	0.17 (0.09)	1518	1.38	32219	2137
9	511	0.66 (0.34)	3066	4.36	56863	4103
10	1017	2.54 (1.31)	6102	13.99	98677	7836
11	2037	9.83 (5.09)	12222	43.23	166650	DNF
12	4076	38.7 (19.9)	24456	121.68	262802	DNF
13	8147	152. (78.3)	48882	282.83	359863	DNF

Table 3: benchmark #2: $n = 3$ and $D = 100$

Benchmarks #3 *and* #4

These sets of problems consist of 14 random multivariate polynomials in $n = 6$ variables and $n = 12$ variables all of total degree $D = 30$. The i'th polynomial will have about 2^i non-zero terms. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 30$. The timings and the number of probes are given in Tables 4 and 5.

i	t	New Algorithm		Zippel		PBox
		Time	Probes	Time	Probes	Probes
1	2	0.00 (0.00)	24	0.01	496	37
2	3	0.00 (0.00)	36	0.01	651	59
3	8	0.00 (0.00)	96	0.01	1364	140
4	16	0.00 (0.00)	192	0.02	2511	284
5	31	0.00 (0.00)	372	0.05	4340	521
6	64	0.02 (0.01)	768	0.15	8060	995
7	127	0.06 (0.03)	1524	0.44	14601	1871
8	255	0.21 (0.09)	3060	1.51	27652	3615
9	511	0.81 (0.35)	6132	5.19	50530	6692
10	1016	3.10 (1.33)	12192	17.94	90985	12591
11	2037	12.2 (5.21)	24444	65.35	168299	DNF
12	4083	48.1 (20.5)	48996	230.60	301320	DNF
13	8151	189. (80.1)	97812	803.26	532549	DNF

Table 4: benchmark #3: $n = 6$ and $D = 30$

To assess the parallel implementation of our algorithm, Table 6 reports timings for benchmark #4 for our algorithm running on 1, 2 and 4 cores showing the speedup we obtain using 2 and 4 cores. We report (in column roots) the time spent computing the roots in step 10 of $\Lambda_1(z)$ using our implementation of Rabin's algorithm which uses classical polynomial arithmetic, and (in column solve) the time solving the linear system for the coefficients in step 22 and (in column probes) the total time spent probing the black box. The data shows that computing the roots will become a bottleneck for our parallel implementation for more cores. Thus for 2 cores and 4 cores we report two timings. The first (in column time 1) is for our parallel algorithm as presented. For the second (faster) time (in column time 2) we have parallelized the second and subsequent steps of the root finding algorithm which yields a modest speedup. The data

i	t	New Algorithm		Zippel		PBox
		Time	Probes	Time	Probes	Probes
1	2	0.00 (0.00)	44	0.03	1736	67
2	4	0.00 (0.00)	96	0.04	3038	121
3	8	0.00 (0.00)	192	0.08	5053	250
4	15	0.00 (0.00)	360	0.20	10230	470
5	32	0.02 (0.01)	768	0.54	18879	962
6	63	0.04 (0.02)	1512	1.79	36735	1856
7	127	0.15 (0.05)	3048	6.10	69595	3647
8	255	0.54 (0.17)	6120	22.17	134664	7055
9	507	2.01 (0.60)	12168	83.44	259594	13440
10	1019	7.87 (2.33)	24456	316.23	498945	26077
11	2041	31.0 (9.16)	48984	1195.13	952351	DNF
12	4074	122.3 (35.9)	97776	4575.83	1841795	DNF
13	8139	484.6 (141.)	195336	>10000	-	DNF

Table 5: benchmark #4: $n = 12$ and $D = 30$

can be interpreted as follows. For $i = 13$ the sequential time is 484.6s. Of this, 34.7s was spent computing the roots of $\Lambda_1(z)$ and 5.02s was spent solving for the coefficients. Thus the algorithm has a sequential component of $34.7 + 5.02 = 39.7$s and so the maximum possible speedup on 4 cores is a factor of $484.6/((484.6 - 39.7)/4 + 39.7) = 3.21$ compared with the observed speedup factor of $484.6/152.5 = 3.18$. One way to remove this bottleneck would be to use a fast multiplication and division algorithm for $\mathbb{Z}_p[z]$.

Benchmark #5

In this benchmark, we compare our new algorithm and the racing algorithm on seven target polynomials (below) from [11, p. 393]. Note, f_6 is dense. The number of probes for each algorithm is reported in Table 7.

$$f_1(x_1,\ldots,x_9) = x_1^2x_3^3x_4x_6x_8x_9^2 + x_1x_2x_3x_4^2x_5^2x_8x_9+$$
$$x_2x_3x_4x_5^2x_8x_9 + x_1x_3^3x_4^2x_5^2x_6^2x_7x_8^2 + x_2x_3x_4x_5^2x_6x_7x_8^2$$
$$f_2(x_1,\ldots,x_{10}) = x_1x_2^2x_4^2x_8x_9^2x_{10}^2 + x_2^2x_4x_5^2x_6x_7x_9x_{10}^2+$$
$$x_1^2x_2x_3x_5^2x_7^2x_9^2 + x_1x_3^2x_4^2x_7^2x_9^2 + x_1^2x_3x_4x_7^2x_8^2$$
$$f_3(x_1,\ldots,x_9) = 9x_2^3x_3^3x_5^2x_6^2x_8^3x_9^3 + 9x_1^3x_2^2x_3^3x_5^2x_7^2x_8^2x_9^3+$$
$$x_1^4x_3^4x_4^2x_5^4x_6^4x_7x_8^5x_9 + 10x_1^4x_2x_3^4x_4^4x_5^4x_7x_8^3x_9 + 12x_2^3x_4^3x_6^3x_7^2x_8^3$$
$$f_4(x_1,\ldots,x_9) = 9x_1^2x_3x_4x_6^3x_7^2x_8x_{10}^4 + 17x_1^3x_2x_5^2x_6^2x_7x_8^3x_9^4x_{10}^3+$$
$$3x_1^3x_2^2x_6^3x_{10}^2 + 17x_2^2x_3^4x_4^2x_7^4x_8^3x_9x_{10}^3 + 10x_1x_3x_5^2x_6^2x_7^4x_8^4$$
$$f_5(x_1,\ldots,x_{50}) = \textstyle\sum_{i=1}^{i=50} x_i^{50}$$
$$f_6(x_1,\ldots,x_5) = \textstyle\sum_{i=1}^{i=5} (x_1 + x_2 + x_3 + x_4 + x_5)^i$$
$$f_7(x_1,x_2,x_3) = x_1^{20} + 2x_2 + 2x_2^2 + 2x_2^3 + 2x_2^4 + 3x_3^{20}$$

i	n	d	$\#f_i$	New Algorithm	ProtoBox
1	9	3	5	90	126
2	10	2	5	100	124
3	9	3	5	90	133
4	9	4	5	100	133
5	50	50	50	5000	251
6	5	5	251	2510	881
7	3	20	6	36	41

Table 7: benchmark #5.

		1 core				2 cores			4 cores		
i	t	time	roots	solve	probe	time 1	time 2	(speedup)	time 1	time 2	(speedup)
6	63	0.04	0.01	0.00	0.04	0.03	0.02		0.02	0.02	
7	127	0.15	0.02	0.00	0.15	0.08	0.08	(1.87x)	0.06	0.05	(3x)
8	255	0.54	0.05	0.00	0.41	0.30	0.28	(1.93x)	0.18	0.17	(3.18x)
9	507	2.02	0.18	0.02	1.48	1.11	1.06	(1.91x)	0.67	0.60	(3.37x)
10	1019	7.94	0.65	0.08	5.76	4.35	4.17	(1.90x)	2.58	2.33	(3.41x)
11	2041	31.3	2.47	0.32	22.7	17.1	16.3	(1.92x)	9.94	9.16	(3.42x)
12	4074	122.3	9.24	1.26	90.0	67.1	64.7	(1.89x)	38.9	35.9	(3.41x)
13	8139	484.6	34.7	5.02	357.3	264.9	255.8	(1.89x)	152.5	141.5	(3.42x)

Table 6: Parallel speedup timing data for benchmark #4 for the new algorithm.

6. CONCLUSION

Our sparse interpolation algorithm is a modification of the Ben-Or/Tiwari algorithm [1] for polynomials over finite fields. It does a factor of between n and $2n$ more probes where n is the number of variables. Our benchmarks show that for sparse polynomials, it usually does fewer probes to the black box than Zippel's algorithm and the racing algorithm of Kaltofen and Lee. Unlike Zippel's algorithm and the racing algorithm, our algorithm does not interpolate each variable sequentially and thus can more easily be parallelized. Our parallel implementation using Cilk, which parallelized only the main loops, demonstrates a good speedup. The downside of our algorithm is that it is clearly worse than Zippel's algorithm and the racing algorithm for dense polynomials. This disadvantage is partly compensated for by the increased parallelism.

Although we presented our algorithm for interpolating over $\mathbb{Z}_p$, it also works over any finite field $GF(q)$. Furthermore, if p (or q) is too small, one can work inside a suitable extension field. We conclude with some remarks about the choice of p in applications where one may choose p.

Theorem 1 says that monomial collisions are likely when $\frac{dt^2}{2\phi(p-1)} > \frac{1}{2}$, that is when $\phi(p-1) < dt^2$. In our benchmarks we used the 31.5 bit prime 3037000453. This is the biggest prime that we can use when programming in C on a 64 bit machine using signed 64 bit machine integers. Using this prime, if $d = 30$, monomial collisions will likely occur when $t > 5,808$ which means 31.5 bit primes are too small for large applications.

It is not difficult to choose p so that $p - 1 = 2q$ with q also prime. The largest such 31.5 bit prime is 3037000427. Solving $\phi(p-1) < dt^2$ for t with $d = 30$ using this prime gives $t > 7,114$. This choice of prime also makes it easy to find generators. However, for $p-1 = 2q$, since -1 is the only element of order 2, any value from the interval $[2, p-2]$ will have order q or $2q$. If we use elements of order q as well as generators in our algorithm, then the probability that two monomials collide is less than $dt^2/(2p-6)$ (using $|S| = p-3$ in the proof of Theorem 1). Solving $p - 3 < dt^2$ for t using $d = 30$ and $p = 3037000427$ yields $t > 10,061$.

The 31.5 bit prime limitation is not a limitation of the hardware, but of the C programming language. On a 64 bit machine, one can use 63 bit primes if one programs multiplication in assembler. We are presently implementing this.

7. ACKNOWLEDGMENTS

We would like to thank Wen-shin Lee for making the source code of the ProtoBox Maple package available to us.

8. REFERENCES

[1] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. of the twentieth annual ACM symposium on Theory of computing*, pages 301–309. ACM, 1988.

[2] Cilk 5.4.6 Reference Manual, `http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf`. Supercomputing Technologies Group, MIT Lab for Computer Science. `http://supertech.lcs.mit.edu/cilk`.

[3] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003.

[4] M. Giesbrecht, G. Labahn and W. Lee. Symbolic-numeric sparse interpolation of multivariate polynomials. *J. Symb. Comput.*, 44:943–959, 2009.

[5] D. Grigoriev, M. Karpinski, and M. Singer. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM J. Comput.* 19:1059–1063, 1990.

[6] M. A. Huang and A. J. Rao. Interpolation of sparse multivariate polynomials over large finite fields with applications. *Journal of Algorithms*, 33:204–228, 1999.

[7] S. M. M. Javadi and M. Monagan. A sparse modular gcd algorithm for polynomials over algebraic function fields. In *Proc. of ISSAC '07*, pages 187–194. ACM, 2007.

[8] E. Kaltofen and Y. N. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In *Proc. of ISSAC '88*, pages 467–474. Springer-Verlag, 1989.

[9] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proc. of ISSAC '90*, pages 135–139. ACM, 1990.

[10] E. Kaltofen, W. Lee, and A. A. Lobo. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In *Proc. of ISSAC '00*, pages 192–201. ACM, 2000.

[11] E. Kaltofen and W. Lee . Early termination in sparse interpolation algorithms. *J. Symb. Comput.*, 36(3-4):365–400, 2003.

[12] J. L. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. on Information Theory*, 15:122–127, 1969.

[13] M. Monagan J. de Kleine and A. Wittkopf. Algorithms for the non-monic case of the sparse modular gcd algorithm. In *Proc. of ISSAC '05*, pages 124–131. ACM, 2005.

[14] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. on Information Theory*, 24:106−110, 1978.

[15] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput*, 9:273–280, 1979.

[16] Jack Schwartz, Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27:701−717, 1980.

[17] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. of EUROSAM '79*, pages 216–226. Springer-Verlag, 1979.

[18] Richard Zippel. Interpolating polynomials from their values. *J. Symb. Comput.*, 9(3):375–403, 1990.

Spiral-Generated Modular FFT Algorithms

[Extended Abstract]

Lingchuan Meng
Drexel University
Philadelphia, PA, USA
lm433@drexel.edu

Jeremy R. Johnson
Drexel University
Philadelphia, PA, USA
jjohnson@cs.drexel.edu

Franz Franchetti
Carnegie Mellon University
Pittsburgh, PA, USA
franzf@ece.cmu.edu

Yevgen Voronenko
Carnegie Mellon University
Pittsburgh, PA, USA
yvoronen@ece.cmu.edu

Marc Moreno Maza
University of Western Ontario
London, Canada
moreno@csd.uwo.ca

Yuzhen Xie
University of Western Ontario
London, Canada
yxie@csd.uwo.ca

ABSTRACT

This paper presents an extension of the SPIRAL system to automatically generate and optimize FFT algorithms for the discrete Fourier transform over finite fields. The generated code is intended to support modular algorithms for multivariate polynomial computations in the `modpn` library used by Maple. The resulting code provides an order of magnitude speedup over the original implementations in the `modpn` library, and the SPIRAL system provides the ability to automatically tune the FFT code to different computing platforms.

Categories and Subject Descriptors

D.1.2 [**Software**]: Programming Techniques, Automatic Programming; G.4 [**Mathematics of Computing**]: Mathematical Software, Efficiency; I.1.3 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation, Languages and Systems

General Terms

Algorithms, Performance

Keywords

FFT, modular arithmetic, code generation, vectorization, high performance computing, autotuning

1. INTRODUCTION

Fast Fourier Transforms (FFTs) are at the core of many operations in scientific computing. In computer algebra, FFTs are used for fast polynomial and integer arithmetic and modular methods (i.e. computation by homomorphic images). In recent years, the use of fast arithmetic has become prevalent and has stimulated the development of software libraries, such as `modpn` [FLMS06, LM06, LMP09] providing hand-optimized low-level routines implementing fast algorithms for multivariate polynomial computations over finite fields, in support of higher-level code. The `modpn` library has been integrated into the computer algebra system MAPLE and runs on all computer platforms supported by MAPLE. The implementation techniques employed in `modpn` are often platform-dependent, since cache size, associativity properties and register sets have a significant impact. In order to take advantage of platform-dependent optimizations, in the context of quickly evolving hardware accelaration technologies, automated performance tuning has become necessary and should be incorporated into the `modpn` library.

SPIRAL [www.spiral.net] is a library generation system that automatically generates platform-tuned implementations of digital signal processing algorithms with an emphasis on fast transforms. Currently, SPIRAL can generate highly optimized fixed-point and floating-point FFTs for a variety of platforms with automatic tuning, and has support for vectorization, threading, and distributed memory parallelization. The code produced is competitive with the best available code for these platforms and SPIRAL is used by Intel for its IPP (Integrated Performance Primitives) and MKL (Math Kernel Library) libraries.

In this work, SPIRAL was extended to generate algorithms for FFT computation over finite fields. This addition required adding a new data type, several new rules and a new transform definition. In addition, the backend was extended to enable the generation of scalar and vectorized code for modular arithmetic. With these enhancements, the SPIRAL machinery can be applied to modular transforms needed by the `modpn` library. In this paper we present preliminary results showing that the code generated by SPIRAL is approximately eleven to twenty-three times faster than the original FFT code in `modpn`.

2. SPIRAL

The SPIRAL system [PMJ05] uses a mathematical framework for representing and deriving algorithms. Algorithms are expressed symbolically as sparse matrix factorizations and are derived using rewrite rules; additional rules are used to symbolically manipulate algorithms into forms that take advantage of the underlying hardware, including vectorization [FVP08] and parallelism [FVP06]. The sequence of applications of breakdown rules is encoded as a *ruletree* which can be translated into a formula and compiled with a special-purpose compiler into efficient code. A search engine with a feedback loop is used to tune implementations to particular platforms. New transforms are added by introducing new symbols and their definitions, and new algorithms can be generated by adding new rules.

Spiral was developed for floating point and fixed point computation; however, many of the transforms and algorithms carry over to finite fields. For example, the DFT of size n is defined when there is a primitive nth root of unity and many factorizations of the DFT matrix depend only on properties of primitive nth roots. In this case, the same machinery in Spiral can be used for generating and optimizing modular transforms. All that is needed is support for new data types and code generation and the addition of new transforms and rules.

For the modular FFT, we added a modular data type, support for modular arithmetic and code generation, and defined the n-point modular DFT

$$\mathbf{ModDFT}_{n,p,\omega} = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n},$$

where ω_n is a primitive nth root of unity in Z_p, and the Cooley-Tukey factorization (rewrite rule)

$$\mathbf{ModDFT}_{n,p,\omega} = (\mathbf{ModDFT}_{r,p,\omega_r} \otimes \mathrm{I}_s)\, \mathrm{T}_s^n (\mathrm{I}_r \otimes \mathbf{ModDFT}_{s,p,\omega_s})\, \mathrm{L}_r^n,$$

where $n = rs$, $\otimes$ denotes the Kronecker or tensor product, T is a diagonal matrix called the twiddle matrix, and L is a special permutation matrix called stride permutation. Additional rules for other FFT algorithms can be used, but for initial experiments this was the only rule used.

3. PERFORMANCE RESULTS

This section reports on preliminary experimental data comparing the performance of hand-coded FFTs from the `modpn` library and FFTs automatically generated by Spiral. Spiral generated algorithms using the Cooley-Tukey rule, and dynamic programming to select an "optimal" recursive breakdown strategy. Dynamic programming is only a heuristic since an optimal algorithm of a given size can depend on the context in which it is called; however, experience shows that it makes good choices. All experiments were performed on an Intel Core i7 965 quad-core processor running at 3.2 GHz with 12 GB of RAM. Generated code was compiled with gcc version 4.3.4-1 with optimization set to O3. Vector code used SSE4.1 with 4-way 32 bit integer vectors. Since there is no vector version of integer division with remainder, in order to vectorize our Spiral generated FFT code on the Core i7, Montgomery's trick [Mont85] was used. Initial experiments were performed using 32 bit integers and 16 bit primes. Figure 1 compares the performance of power of two FFTs of size 4 through 4096 using the original `modpn` code and scalar and vectorized code generated by Spiral, where performance is reported in Gops (giga-ops) or billions of operations per second (higher is better), which is calculated assuming that the DFT of size N takes a total of $(3/2)N\lg(N)$ additions, subtractions and nontrivial multiplications. The Spiral generated vector code is between 11 and 23 times faster than the original `modpn` code.

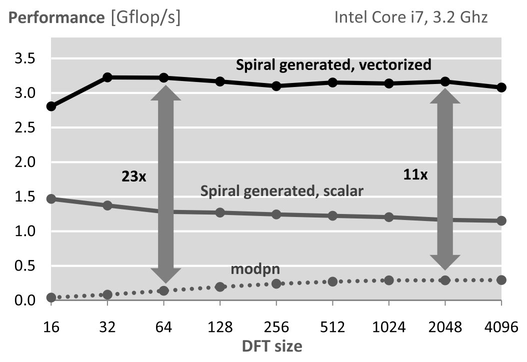


Figure 1: Performance Comparison

4. REFERENCES

[FLMS06] A. Filatei, X. Li, M. Moreno Maza, and É. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*, pp 93–100, New York, NY, USA, 2006. ACM Press.

[FVP06] Franz Franchetti, Yevgen Voronenko and Markus Püschel, "FFT Program Generation for Shared Memory: SMP and Multicore," Proc. Supercomputing (SC), 2006.

[FVP08] Franz Franchetti, Yevgen Voronenko and Markus Püschel, "A Rewriting System for the Vectorization of Signal Transforms," Proc. High Performance Computing for Computational Science (VECPAR), Lecture Notes in Computer Science, Springer, Vol. 4395, pp. 363-377, 2006.

[LM06] X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In A. Iglesias and N. Takayama, editors, *Proc.* International Congress of Mathematical Software - ICMS 2006, pp 12–23. Springer, 2006.

[LMP09] Xin Li, Marc Moreno Maza, and Wei Pan. Computations modulo regular chains. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pp 239–246, New York, NY, USA, 2009. ACM.

[Mont85] P. L. Montgomery. Modular Multiplication Without Trial Division. Mathematics of Computation, vol. 44, no. 170, pp. 519–521, 1985.

[PMJ05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo SPIRAL: Code Generation for DSP Transforms Proceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation," Vol. 93, No. 2, 2005, pp. 232-275.

[www.spiral.net] Spiral project website. http://www.spiral.net, 2010.

High performance linear algebra using interval arithmetic

[Extended Abstract]

Hong Diep NGUYEN
INRIA
Université de Lyon
Laboratoire LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL)
École Normale Supérieure de Lyon, 46 allée d'Italie, 69007 Lyon, France
Hong.Diep.Nguyen@ens-lyon.fr

Nathalie REVOL
INRIA
Université de Lyon
Laboratoire LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL)
École Normale Supérieure de Lyon, 46 allée d'Italie, 69007 Lyon, France
Nathalie.Revol@ens-lyon.fr

ABSTRACT

In this paper, we describe implementations of interval matrix multiplication and verified solution to a linear system, using entirely BLAS routines, which are fully optimized and parallelized.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Reliability and Robustness

General Terms

Algorithm, Performance, Reliability, Verification

Keywords

efficient implementation, linear algebra, interval arithmetic, interval matrix multiplication, certified solution of a linear system.

1. INTERVAL MATRIX PRODUCT

Rump first proposed a fast and parallel implementation of interval matrix multiplication based on midpoint-radius interval arithmetic [6]. His implementation uses four floating-point matrix products to compute an interval matrix times interval matrix operation. Thus, his implementation of the interval matrix product relies on the floating-point matrix product, for which highly optimized implementations exist, such as in BLAS. Such an implementation is much more efficient than the naive implementation, where the interval operations take place for the product of each pair of coefficients and for the related additions. Even if, in principle, the naive approach requires fewer floating-point operations, in practice it implies to change the rounding mode (downwards or upwards) before each floating-point operation. This involves a penalty in terms of performance. Indeed, on many processors, the only way to change the rounding mode is by setting a global flag, and thus changing the context. This prevents the implementation from using the pipelining capabilities of the processor. On other processors, such as the 64-bit Intel Itanium architecture, it is possible to specify the rounding mode within the code of the operation, and thus it is possible to exploit the pipeline. However, accessing the code is possible only at the assembly level, but not at a higher level. In practice, we know of no library for interval arithmetic which takes benefit of this possibility. These considerations explain why Rump's implementation is one of the most efficient today.

On the other hand, Rump's approach always provides results overestimating the exact results. Rumps established in [6] that the overestimation factor in the worst case is 1.5.

We propose an implementation to reduce this overestimation factor, which trades some of the efficiency of Rump's method for accuracy. Our approach is based on an observation of interval product when one multiplier does not contain zero. Let $\mathbf{a}, \mathbf{b}$ be two intervals. Let ma, ra and mb, rb be their midpoint and radius respectively. If $\mathbf{a}$ does not contain zero, ie. $|ma| \geq ra$, then the product between $\mathbf{a}$ and $\mathbf{b}$ is an interval $\mathbf{c}$ whose midpoint and radius are computed by

$$\begin{aligned} mc &= ma * mb + \texttt{sign}(ma) * ra * \frac{|mb + rb| - |mb - rb|}{2} \\ rc &= |ma| * rb + ra * \frac{|mb + rb| + |mb - rb|}{2} \end{aligned}$$

This observation leads to another formula for the product of two interval matrices, which uses six floating-point matrix products in the case where one of the operands does not contain zero, i.e. when each coefficient does not contain zero. However, the coefficients can be of different signs. Let us remark that this formula is exact in the absence of rounding errors, ie. there is no overestimation of the result.

For the general case where intervals may contain zero in their interior, we decompose one of the operand matrices into two parts, one which is centered in zero and one which does not contain zero. Let two intervals $\mathbf{a}$ and $\mathbf{b}$, represented by their endpoints: $\mathbf{a} = [\underline{a}, \bar{a}]$ and $\mathbf{b} = [\underline{b}, \bar{b}]$. The formula for the product of $\mathbf{a}$ and $\mathbf{b}$, where $\mathbf{a}$ is centered in zero i.e. $\underline{a} = -\bar{a}$, is an interval $\mathbf{c}$ whose endpoints are $\bar{c} = \bar{a} \cdot \text{mag}(\mathbf{b})$ and $\underline{c} = -\bar{c}$, with $\text{mag}(\mathbf{b}) = \max(|\underline{b}|, |\bar{b}|)$. Using the symmetry of this formula, the product between a centered-in-zero matrix and another interval matrix can be computed exactly by only one floating-point matrix product. Again, the other matrix product requires six floating-point matrix products. Hence in total, the product of two general matrices requires seven floating-point matrix products to compute the two sub-products. Finally, a matrix addition, which is negligible in terms of computational time, yields the final result.

Nonetheless, although each sub-product is computed exactly, the computed result is an overestimation of the exact result, because of the subdistributive property of interval operations. We establish, similarly as in [5], that the overestimation factor in the worst case of our algorithm is less than 1.18. In some cases, eg. when one of the operand matrices does not contain zero in its interior, our algorithm provides results with no overestimation (in exact arithmetic).

2. CERTIFIED SOLUTION OF A LINEAR SYSTEM

Another fundamental problem addressed here is the certification of the solution of a linear system. Given a linear system of equations $Ax = b$, our goal is to compute an approximate solution as close as possible to the exact solution, and at the same time compute an enclosure of the error upon this computed solution. We call "certified solution of a linear system" this pair consisting of an approximate solution and an enclosure of the error. In particular, the knowledge of the enclosure of the error enables us to (under-)estimate the number of exact digits obtained for this solution.

To this end, we base our approach on the iterative refinement method, as explained in [2]. At the first step, an approximate solution $\tilde{x}$ of the linear system $Ax = b$ is computed by some floating-point method. The classical iterative refinement method consists in solving the so-called residual system, i.e. the linear system satisfied by the error $e = x^* - \tilde{x}$ between the exact (unknown) solution s^* and the approximate solution $\tilde{x}$: $Ae = b - A\tilde{x}$. The residual is the vector $b - A\tilde{x}$. Then, the computed approximation $\tilde{e}$ of e is added to $\tilde{x}$ as a correction term: the refined solution is then $\tilde{x} + \tilde{e}$. This refinement step can be repeated to improve further the approximate solution $\tilde{x}$. Our method, detailed in [4], differs from the floating-point iterative refinement, by the fact that we use the residual system to refine the error bound – which is an interval containing the error – using an interval improvement method, namely Jacobi or Gauss-Seidel iterations detailed in [3]. Hence, instead of computing an approximate residual, we must compute an enclosure of the exact residual. Therefore, the residual is computed twice in floating-point arithmetic, with upward and downward rounding modes $\mathbf{r} = [(b - Ax)_\downarrow, (b - Ax)_\uparrow]$.

To ensure the convergence property of the interval iterative refinement, the matrix of residual system is first preconditioned by an approximate inverse R of the coefficient matrix, in order to be transformed into a (hopefully) diagonally dominant matrix. If the preconditioned matrix is not a diagonally dominant matrix, of more precisely if the method cannot guarantee that it is a H-matrix, then the process fails. Again, the preconditioned matrix is computed twice, once with upward and once with downward rounding modes, $\mathbf{K} = [(R*A)_\downarrow, (R*A)_\uparrow]$, before the property of being a H-matrix is checked. The residual is also preconditioned by R, $\mathbf{z} = R * \mathbf{r}$. Then this preconditioned, interval system $\mathbf{K}e = \mathbf{z}$ will be used to improve and to finally obtain a tight enclosure of the error.

If it can be established that $\mathbf{K}$ is a H-matrix, then the interval improvement consists in computing a new interval **e'** from the old one **e** which encloses more closely the exact value of the error. Let $\mathbf{D}, \mathbf{L}, \mathbf{U}$ be the diagonal, the lower triangular matrix and the upper triangular matrix of $\mathbf{K}$, respectively. The interval improvement is expressed by:

$$\begin{array}{lcll} \mathbf{D} * \mathbf{e'} & = & \mathbf{z} - (\mathbf{L} + \mathbf{U}) * \mathbf{e} & \text{(Jacobi)} \\ \mathbf{D} * \mathbf{e'} & = & \mathbf{z} - (\mathbf{L} * \mathbf{e'} + \mathbf{U} * \mathbf{e}) & \text{(Gauss-Seidel)} \end{array}$$

Each step of interval improvement comprises either an interval matrix-vector product (Jacobi iterations) or the solution of an interval triangular linear system (Gauss-Seidel iterations), which are in any way costly to perform. To sort out this problem, we exploit the fact that the product between an interval matrix centered in zero and an interval vector can be computed by only one floating-point matrix vector product. Indeed, we inflate the matrix $\mathbf{K}$ in such a way that it is centered in a diagonal matrix without deteriorating noticeably the quality of the result. This technique is motivated by the fact that the residual system being preconditioned by an approximate inverse of the coefficient matrix, then the preconditioned matrix should be close to Identity.

Using this relaxation technique, each interval improvement step can be expressed by one floating-point matrix vector product. Experimental results [4] show that this technique helps to reduce significantly the execution time meanwhile still achieving the same accuracy as the original one.

As we can see, except for the preconditioning stage, which is of order $\mathcal{O}(n^3)$ and which seems unavoidable if using interval arithmetic to solve a linear system, the rest of our approach is expressed only by floating-point matrix-vector products, which are of order $\mathcal{O}(n^2)$ and negligible to the preconditioning stage. Moreover, the whole method requires solely floating-point operations, namely matrix-matrix and matrix-vector products. Thus, in practice, it can be implemented using BLAS routines, which are highly optimized and parallelized on almost any computer. The use of a floating-point library such as BLAS alleviates the burden of optimizing and parallelizing interval routines.

References

[1] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee and E. J. Riedy. *Bounds from Extra Precise Iterative Refinement*, ACM Trans. Mathematical Software, Vol. 32, No. 2, June 2006, pp. 325-351.

[2] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM Press, 2002. Chapter 12: Iterative Refinement.

[3] A. Neumaier. *Interval methods for systems of equations*, Cambridge University Press, 1990. Chapter 4: The solution of square linear systems of equations.

[4] H. D. Nguyen and N. Revol. *Solve and Certify a Linear System*, Reliable Computing, vol. 15, 2010.

[5] H. D. Nguyen. *Efficient implementation of interval matrix multiplication*, INRIA Research Report, April 2010, `http://hal.inria.fr/inria-00469472/en/`.

[6] S. M Rump. *Fast and parallel interval arithmetic*, BIT, Vol. 39, No. 3, 1999, pp. 534–554.

Parallel Computation of Determinants of Matrices with Polynomial Entries for robust control design

[Extended Abstract]

Kinji Kumura
Graduate School of Informatics, Kyoto University
36-1 Yoshida-Honmachi, Sakyo-ku
Kyoto, 606-8501, Japan
kkimur@amp.i.kyoto-u.ac.jp

Hirokazu Anai
Fujitsu Laboratories Ltd / Kyushu University
4-1-1 Kamikodanaka, Nakahara-ku
Kawasaki 211-8588, Japan
h.anai@kyudai.jp

ABSTRACT

In this paper we consider computing determinants of polynomial matrices symbolically. Determinant computation of matrices with polynomial entries in a small number of variables is of particular interest since it commonly appears in solving engineering design problems. A parallel algorithm based on multivariate Newton polynomial interpolation with "cut-surface" (total degree bound) is presented and its efficiency is demonstrated by showing computational results for some practical examples from control system design.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Algorithms; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; J.6 [**Computer Applications**]: Computer-Aided Engineering

General Terms

Algorithms, Experimentation

Keywords

parallel determinant computation, polynomial matrix, Newton polynomial interpolation, total degree bound

1. INTRODUCTION

Nowadays computer algebra is becoming increasingly common as an effective tool in many fields i.e., science, engineering and education. In control community, symbolic approaches have been actually used in a wide variety of contexts [4]. We have been developing a parametric robust control toolbox based on symbolic computation such as quantifier elimination (QE) on MATLAB (also a Maple version) [3]. A special QE algorithm based on the Sturm-Habicht sequence computation is employed in the toolbox for speeding-up its performance, since it requires solving a particular class of formulas: sign definiteness of an univariate polynomial with parametric coefficients. We note that no parallelized algorithm is used in the toolbox so far.

In order to boost up the limit of its practical applicability of computer algebra algorithms, the development of parallel algorithms are important. Moreover, recent years have seen the advent of cloud computing, which enables us to use such a tool on the Internet, as well as services which is provided in the form of SaaS (software as a service). Thus it appears certain that parallel algorithms play a significant role in these context. In this paper we present a parallel algorithm for determinant computation of polynomial matrices based on multivariate Newton polynomial interpolation incorporating total degree bounds, which we call a "cut-surface" from the their shapes in a Newton polygon of determinants .

The Sturm-Habicht sequence computation consists of many resultant computations of polynomial matrices and hence the speeding up of the determinant computation is the crucial potion (see [2]). Our parametric robust control toolbox aims at at designing low degree fixed-structure controllers, that is, the so-called PI (2 design parameters) and PID (3 parameters) controllers. The low degree fixed structure controllers are frequently used in industry and hence an important class of control system design problems. So what is required for our current purpose is to develop the efficient determinant computation of matrices with polynomial entries in 2 or 3 variables. We show the efficiency of our proposed parallel algorithm by demonstrating computational results for some examples from control system design together with the comparison other implementations of determinant computation such as Maple, Singular, and Mathematica.

2. ALGORITHM

There have been several previous works on parallel computation of determinants of polynomial matrices. For example, in [5] the method based on multivariate Lagrange interpolation is proposed. We propose yet another parallel algorithm based on multivariate Newton interpolation. Our approach is more effective particularly for computing determinants of matrices with entries of polynomials in a small number of variables efficiently by virtue of usage of total degree bounds.

Basic idea of our method: In general, an interpolation approach is quite suitable for parallelization since the evaluations of a determinant at sampling points can be executed independently. Our parallel algorithm for computing determinants of matrices whose elements are polynomials is based on multivariate Newton interpolation. The key point to reduce the computational cost is usage of the total degree bounds on determinants. We can guess the "tighter" form of the determinant, that is, reduce the number of the candidates for supports of the determinant, by using not only the degree bounds for each variable but also the total degree bounds. This greatly improves the computational efficiency of determinant computation. Since the total degree bounds produce tilted surfaces in the Newton polygon of the determinant, we call them "cut-surfaces". We note that it is indicated by H.Werner [7] that the Newton interpolation under the situation of monotonic decrease of the degrees on the Newton polygon is possible. The novelty of our proposal is as follows: We provide a useful realization of the remark in [7], i.e., combination of the Newton interpolation and the total degree bounds, for efficient parallel determinant computation. Moreover we also combine a modular technique with our interpolation scheme (in evaluating a determinant at a sampling point) via the Chinese remainder theorem.

Degree & coefficient bounds: As for the degree bound for each variable, we employ $\min(S_{maxd_r}, S_{maxd_c})$ where S_{maxd_r} is the sum of maximum degree of each row and S_{maxd_c} is that of each column. Then we employ the coefficient bound on determinants for polynomial matrices proposed by Goldstein and Graham [1] for using the modular technique.

3. COMPUTATIONAL EXAMPLES

Here we briefly show a part of our experimental results due to the page limit. We have implemented the proposed algorithm in C language. All computations are done on a multi-core processor: Intel Core i5 M520 with 4GB memory and the OS is Windows 7 (64bit).

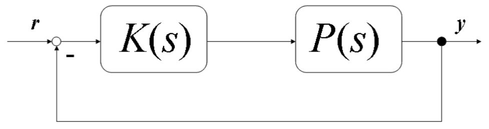


Figure 1: A feedback control system

We consider a feedback control system of the form Figure 1 with a PI(proportional-integral) $K(s) = k + \frac{m}{s}$ or PID(proportional-integral-derivative) controller $K(s) = k + \frac{m}{s} + s\ell$ and a plant $P(s)$. The problem we solve here is to find feasible parameters (k, m and ℓ) so that the system satisfies the H_∞-norm constraint: $\| S(s) \| < 1$ for a sensitive function of the system $S(s) = \frac{1}{1+P(s)K(s)}$ (see [3]). The H_∞-norm constraint can be recast as a sign definite condition (SDC) given by the following: $\forall x > 0 \quad f(x) > 0$, where f is a univariate polynomial with parametric coefficients. The SDC is solved by a QE using the Sturm-Habicht sequence. The crucial part in the Sturm-Habicht sequence computation for the SDC is discriminant computation of f. We formulated the corresponding SDCs for the feedback systems with the following practical plants and computed the discriminants, which requires determinant computation of polynomial Dixon matrices.

$\langle 1 \rangle$ a certain ladder type system for a flexible structure
$\langle 2 \rangle$ an large analog integrated circuit from [6]
$\langle 3 \rangle$ a swing-arm of a Hard Disk Drive

Here we show the timing data of computing determinants only for 3 parameter cases for $\langle 1 \rangle$ with results by other

deg(f)	Maple	Singular	Mathematica	our method
15	14.617	>1 min	9.672	0.391
17	29.063	>1 min	20.202	0.746
19	55.255	>1 min	37.284	1.433

Table 1: Timing data for a PID controller (sec)

computer algebra systems: Maple 13 (64bit), Singular-3-1-1 (32bit) and Mathematica 6 (32bit) in Table 1.[1] We have three different size systems for $\langle 1 \rangle$, which lead to the SDCs with degree 15, 17 and 19. From Table 1 we can say that our approach is effective for computing determinants of polynomial matrices in a small number of variables. Further results and discussion will be shown at the conference.

4. CONCLUSION

We have shown a parallelized algorithm for computing determinants of polynomial matrices based on the Newton interpolation using total degree bounds and the Chinese remainder theorem. We applied our algorithm to compute the determinants derived from various robust control design problems and confirmed that our approach is effective for computing determinants of matrices with entries of polynomials in a small number of variables through the experimental results for practical control design problems.

5. REFERENCES

[1] A. Goldstein and R. Graham. A Hadamard-type bound on the coefficient of a determinant of polynomials. *SIAM Review*, 16:394–395, 1974.

[2] L. González-Vega, H. Lombardi, T. Recio, and M.-F. Roy:. Sturm-Habicht sequence. In *Proceedings of ISSAC'89*, pages 136–146, Portland, 1989. ACM Press.

[3] N. Hyodo, M. Hong, H. Yanami, S. Hara, and H. Anai. Solving and visualizing nonlinear parametric constraints in control based on quantifier elimination. *Appl. Algebra Eng. Commun. Comput.*, 18(6):497–512, 2007.

[4] N. P. Karampetakis and A. I. G. V. (Edts). *Special issue on the use of computer algebra systems for computer aided control system design, International Journal of Control Vol. 79, No. 1.* Taylor & Francis, 2006.

[5] A. Marco and J.-J. Martinez. Parallel computation of determinants of matrices with polynomial entries. *J. Symb. Comput.*, 37(3):749–760, 2004.

[6] X.-D. Tan and C.-J. R. Shi. Hierarchical symbolic analysis of analog integrated circuits via determinant decision diagrams. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 19(4):401–412, 2000.

[7] H. Werner. Remarks on Newton-type multivariate interpolation for subsets of grids. *Computing*, 25:181–191, 1980.

[1]Note that all computational times by our method here have been obtained by running the serial version of our implementation for fare comparison. The parallel version is much faster. We will also report the comparison between serial and parallel versions of our method at the conference. For the actual matrices, see `http://www-is.amp.i.kyoto-u.ac.jp/kkimur/SDC/sdc.html`

Cache Friendly Sparse Matrix-vector Multiplication

[Extended Abstract]

Sardar Anisul Haque
University of Western Ontario
ON N6A 5B7, Canada
shaque4@csd.uwo.ca

Shahadat Hossain
University of Lethbridge
AB T1K 3M4, Canada
shahadat.hossain@uleth.ca

Marc Moreno Maza
University of Western Ontario
ON N6A 5B7, Canada
moreno@csd.uwo.ca

1. INTRODUCTION

Sparse matrix-vector multiplication or *SpMXV* is an important kernel in scientific computing. For example, the conjugate gradient method (CG) is an iterative linear system solving process where multiplication of the coefficient matrix A with a dense vector x is the main computational step accounting for as much as 90% of the overall running time. Though the total number of arithmetic operations (involving nonzero entries only) to compute Ax is fixed, reducing the probability of cache misses per operation is still a challenging area of research. This preprocessing is done once and its cost is amortized by repeated multiplications. Computers that employ cache memory to improve the speed of data access rely on reuse of data that are brought into the cache memory. The challenge is to exploit data locality especially for unstructured problems: modeling data locality in this context is hard.

Pinar and Heath [8] propose column reordering to make the nonzero entries in each row contiguous. However, column reordering for arranging the nonzero entries in contiguous location is NP-hard [8]. In a considerable volume of work [2, 6, 8, 9, 10] on the performance of *SpMXV* on modern processors, researchers propose optimization techniques such as reordering of the columns or rows of A to reduce, for example, indirect access and improving data locality, and blocking for reducing memory load and loop overhead.

In this extended abstract, we present a new column ordering algorithm, based on the binary reflected Gray codes, that runs in linear time with respect to the number of nonzero entries. We analyze the cache complexity of *SpMXV* when the sparse matrix is ordered by our technique. The results from numerical experiments, with very large test matrices, clearly demonstrate the performance gains rendered by our proposed technique.

Categories and Subject Descriptors

F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices; G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems (direct and iterative methods)

General Terms

Algorithms

Keywords

Sparse Matrix, Cache Complexity, Binary reflected Gray Code

2. BINARY REFLECTED GRAY CODE ORDERING

We develop a new column ordering algorithm based on *binary reflected Gray code* (BRGC for short) for sparse matrices. We will call it BRGC ordering. A p-bit *binary reflected Gray code* [3] is a Gray code denoted by G^p and defined by $G^1 = [0, 1]$ and

$$G^p = [0G_0^{p-1}, \ldots, 0G_{2^{p-1}-1}^{p-1}, 1G_{2^{p-1}-1}^{p-1}, \ldots, 1G_0^{p-1}],$$

where G_i^p is the i-th binary string of G^p. We call i the rank of G_i^p in G^p. For example the rank of 011 in G^3 is 2. We consider each column of a $m \times n$ sparse matrix A as a binary string of length m where each nonzero is treated as 1. Hence, we have n binary strings of length m, say $\{b_0, b_1, \ldots, b_{n-1}\}$. Let Π be the permutation of these strings satisfying the following property. For any pair of indices i, j with $i \neq j$, the rank of $b_{\Pi(j)}$ in G^m is less than that of $b_{\Pi(i)}$ if and only if $\Pi(i) < \Pi(j)$ holds. We refer to A_{brgc} as our sparse matrix A after its columns have been permuted by Π. One can check that the BRGC ordering sorts the columns of A according to their ranks in G^m in descending order.

On average, our sorting algorithm proceeds in ρ (the average number of nonzeros in a column) successive phases, which are described below. During the first phase, we sort the columns by increasing position of their first nonzero entries from above, creating equivalence classes where any two columns are uncomparable for this ordering. During the second phase, in each equivalence class, we sort the columns by decreasing position of their second nonzero entries from above, thus, refining the equivalence classes of the first phase into new classes where again any two columns are uncomparable for this second ordering. More generally, during the k-th phase, in each equivalence class obtained at the $(k-1)$-th phase, we sort the columns by increasing position (resp. decreasing position) of their k-th nonzero entry from above, if k is odd, (resp. if k is even) thus, refining again the equivalence classes. Continuing in this manner, we obtain the desired sorted matrix. Observe that whenver an equivalence class is a singleton, it no longer participates to the next sorting phases.

Based on the above procedure and the counting sort algorithm [4], the matrix A_{brgc} is obtained from A using $O(\tau)$ integer comparisons (on average) and $O(n+\tau)$ data-structure updates, where τ is the total nonzero entries in A [7].

Let C be an equivalence class obtained after the ℓ-th phase and before the $(\ell+1)$-th phase. We call *nonzero stream at level* $(\ell+1)$ *in* C the set of the $(\ell+1)$-th nonzero entries from above in the columns of C. In the nonzero stream at level $(\ell+1)$ in C, a set of nonzeros having the same row index is called a *step*.

3. CACHE COMPLEXITY

Consider the ideal cache [5] of Z words, with cache line of L words. Assume that n is large enough such that the vector x does not fit into the cache. During *SpMXV*, the total number of accesses in x is τ. These accesses are usually irregular. Note that n of those accesses are cold misses. Each of the other $\tau - n$ accesses creates a cache miss with probability $(n - Z/L)/n$, since no spatial locality should be expected in accessing x. Therefore, the total number of expected cache misses in accessing x is computed as follows.

$$\overline{Q_1} = Z/L + (\tau - Z/L)\frac{n-Z/L}{n}.$$

We claim that A_{brgc} has at least nonzero streams at level 1 and 2. Indeed, each column has at least some nonzeros, which implies that A_{brgc} has nonzero stream at level 1. Observe that each step of the nonzero stream at level 1 is expected to have ρ entries. Moreover, we assume $\rho \geq 2$. This leads to the formation of the nonzero stream at level 2. Therefore, the total number of nonzeros, in all the nonzero streams of level 1 and 2, is $2n$. Due to the LRU replacement policy, one can expect that the n multiplications with the nonzeros in the nonzero stream at level 1 incur the same amount of cache misses as if x was scanned in a regular manner during *SpMXV*. Next, we observe that each of the accesses in x for multipliying with nonzeros in the nonzero streams at level 2 creates cache misses with probability $\frac{n/\rho - Z/L}{n/\rho}$. More generally, each of the other access in x creates cache miss with probability $\frac{cn/\rho - Z/L}{cn/\rho}$, where, c is the average number of nonzero streams under one step of first level nonzero stream and $1 \leq c \leq \rho$. Therefore, the expected cache misses in accessing x is given by:

$$\overline{Q_2} = n/L + Z/L + (n - Z/L)\frac{n/\rho - Z/L}{n/\rho} + (\tau - 2n)\frac{cn/\rho - Z/L}{cn/\rho}.$$

We apply the computer algebra system MAPLE to analyze the difference between $\overline{Q_1}$ and $\overline{Q_2}$. For the large matrices of [1], the equality $n = O(Z^2)$ holds for level 2 cache and our calculations show that we have, $\overline{Q_1}$ - $\overline{Q_2} \approx n$.

4. EXPERIMENTAL RESULTS

We selected 10 matrices from [1] for our experimentation. The basic information for each test matrix is given in Table 4. We run our experiments on an *intel core 2 processor Q6600*. It has L2 cache of 8MB and the CPU frequency is 2.40 GHz [11]. We measure the CPU time (given in seconds) for 1000 *SpMXV*s for three variants reported in Table 4: with BRGC ordering, without any preprocessing and after a random re-ordering of the columns. It shows that

Matrix name	m	n	τ
fome21	67748	216350	465294
lp_ken_18	105127	154699	358171
barrier2-10	115625	115625	3897557
rajat23	110355	110355	556938
hcircuit	105676	105676	513072
GL7d24	21074	105054	593892
GL7d17	1548650	955128	25978098
GL7d19	1911130	1955309	37322725
wikipedia-20051105	1634989	1634989	19753078
wikipedia-20070206	3566907	3566907	45030389

Table 1: Test matrices.

Matrix name	BRGC ordering	no ordering	random ordering
fome21	3.6	3.9	4.8
lp_ken_18	2.7	3.1	3.3
barrier2-10	19.0	19.1	23.2
rajat23	3.0	3.0	3.4
hcircuit	2.6	2.5	2.9
GL7d24	3.0	3.2	3.1
GL7d17	484.6	625.0	580.7
GL7d19	784.6	799.0	899.2
wikipedia-20051105	258.9	321.0	411.5
wikipedia-20070206	731.5	859.0	1046.0

Table 2: CPU time for 1000 *SpMXV*s.

the cost of BRGC ordering is amortized by 1000 *SpMXV*s for all of the matrices. Our experimental results also show that the cost of BRGC ordering algorithm, as a preprocessing step, can be much less than $\sqrt{n}$ *SpMXV*s and thus can improve the performances of CG-type algorithms in practice. Note that other column ordering algorithms reported in [8] and their performances are compared with BRGC ordering algorithm in [6]. As reported in [6], BRGC algorithm outperforms these other column ordering algorithms on three different computer architectures.

5. REFERENCES

[1] T. Davis, Uni. of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/

[2] E. Im, Optimizing the performance of sparse matrix-vector multiplication. *PhD Thesis, Uni. of California Berkeley*, 2000.

[3] D. Kreher and D. Stinson, *Combinatorial Algorithms :Gen., Enum., and Search.* CRC Press, 1999.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* 2nd Edition McGraw-Hill, 2001.

[5] M. Frigo, C. E. Leiserson, Prokop, Harald, Ramachandran, and Sridhar, Cache-Oblivious algorithms. FOCS '99: Proc. of the 40th Annual Symp. on Foundations of Comp. Sc., 1999

[6] S. Haque, A computational study of sparse matrix storage scheme M.Sc. Thesis, Uni. of Lethbridge , 2008.

[7] S. Haque, and M. Moreno Maza, Algorithms for sorting large objects, Tech. Report, Uni. of Western Ontario, 2010.

[8] A. Pinar and M. Heath, Improving performance of sparse matrix-vector multiplication. In *Supercomputing '99: Proc. of the 1999 ACM/IEEE conf. on Supercomputing (CDROM)*, New York, USA, 1999.

[9] S. Toledo, Improving the memory-system performance of sparse-matrix vector multiplication, In *IBM J. Res. Dev.*, vol. 41, num. 6, 1997.

[10] R. Vuduc, Automatic performance tuning of sparse matrix kernels. *PhD Thesis, Uni. of California Berkeley*, 2003.

[11] Intel Webpage, Intel core 2 quad processor q6600. http://ark.intel.com/Product.aspx?id=29765

Parallelising the computational algebra system GAP

[Extended Abstract]

R. Behrends,
A. Konovalov, S. Linton
School of Computer Science
University of St Andrews
{rb,alexk,sal}@cs.st-and.ac.uk

F. Lübeck
LDFM
RWTH Aachen
frank.luebeck@math.rwth-aachen.de

M. Neunhöffer
School of Mathematics and Statistics
University of St Andrews
neunhoef@mcs.st-and.ac.uk

ABSTRACT

We report on the project of parallelising GAP, a system for computational algebra. Our design aims to make concurrency facilities available for GAP users, while preserving as much of the existing code base (about one million lines of code) with as few changes as possible and without requiring users—a large percentage of whom are domain experts in their fields without necessarily having a background in parallel programming—to have to learn complicated parallel programming techniques. To this end, we preserve the appearance of sequentiality on a per-thread basis by containing each thread within its own data space. Parallelism is made possible through the notion of migrating objects out of one thread's data space into that of another one, allowing threads to interact and via limited use of lockable shared data spaces.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Miscellaneous

General Terms

Design, Languages, Performance

Keywords

GAP, shared memory programming, data spaces

1. INTRODUCTION

GAP (`http://www.gap-system.org`), is an open-source system for computational discrete algebra, with particular emphasis on Computational Group Theory. It provides a programming language, an extensive library of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects. The kernel of the system is implemented in C, and the library is implemented in the GAP language. Both the kernel and the library are sequential and do not support parallelism.

In the 4-year long EPSRC project "HPC-GAP: High Performance Computational Algebra and Discrete Mathematics" (`http://www-circa.mcs.st-and.ac.uk/hpcgap.php`), we aim at reengineering the GAP system to allow parallel programming in GAP both in shared and distributed memory programming models. Below we report on the current progress with extending the system with new functionality for shared memory programming in GAP (in contrast with ParGap [5], which assumes a distributed system).

2. CURRENT USERS OF GAP

The main driver for our programming model is that we have a fairly large user base (estimated in thousands of users and a large number of package authors) who have a considerable investment in the current sequential architecture. While many of them will likely be interested in leveraging the increased performance of multicore processors, it is unlikely that they will accept changes that invalidate their existing code. In addition, the GAP standard library and the packages distributed with GAP are about one million lines of code in size, a rewrite of which would be prohibitive.

We are looking at the following types of users as the ones that a parallel programming model should accommodate.

(1) Domain experts who do not have expertise in parallel programming or lack the resources to invest into it. These users range from the student who uses GAP as a desktop calculator for computational algebra problems to the maintainer of an existing GAP package where the rewrite for a parallel model would be cumbersome. These users do not get sufficient benefits out of parallel programming to justify the costs. For them, at least the appearance of a sequential environment must be maintained and sequential code must not suffer performance penalties[1].

(2) Users who wish to leverage the benefits of parallelism, but are not expert parallel programmers themselves. These users need to be provided with a set of high level tools, such as parallel skeletons [2], to parallelise their algorithms without having to worry about the common pitfalls of parallel programming.

(3) The third type of user is the parallelism expert. Parallelism experts are familiar with how to optimise algorithms for parallel programming and need to be provided with the

[1]Performance concerns [4] are one reason why we have so far rejected Software Transactional Memory as a programming model, despite its relative simplicity for non-experts.

necessary low-level tools to control how threads interact in order to optimise performance. At the same time, these low level tools need to come with reliable error detection to minimise the occurrence of race conditions or deadlocks.

3. THE PARALLEL ENVIRONMENT

Our parallel programming model builds on the notion of segregating threads through disjoint *data spaces.* Data spaces form a partition of GAP objects, i.e. each GAP object is the member of exactly one data space. Only one thread at a time can have exclusive access to a data space (checked at runtime), which ensures mutual exclusion.[2]

We distinguish between three types of data spaces. First, there is a *thread-local data space* associated with each thread, to which that thread always has exclusive access; second, there are an indefinite number of *shared data spaces* to which threads can gain exclusive or shared access through an associated read-write lock; finally, there is a single *public data space*, to which all threads have shared access at all times. If a thread has exclusive access to a data space, that means that it can perform any operation on objects within this data space; if a thread has shared access to a data space, it can perform any operation on these objects that does not interfere with other threads performing similar operations. An example is read-only operations.

In order for threads to interact, objects can be migrated between data spaces[3]. An object can only be migrated out of a data space by a thread if that thread has exclusive access to that data space. This constraint makes migration a cheap operation: each object has a descriptor (implemented as a C pointer) for the data space that contains it; objects are migrated by updating that descriptor. Because the thread has exclusive access to the data space (and thus, the object), this can be implemented as a simple memory write[4].

Before performing an operation on an object, a thread checks the data space descriptor of the object to ensure that the thread has the proper access to perform an operation. The most common cases are optimised for: access to the thread's thread-local data space; the public data space; and the most recently used shared data space. Checks that can be statically shown to be superfluous are eliminated.

Thread-local data spaces accommodate our first type of users. A piece of code that executes solely within a single thread-local data space is indistinguishable from a purely sequential program and is guaranteed to not have race conditions or deadlocks.

Shared data spaces aim primarily at the third type of user, the parallelism expert. Access to them is controlled by explicit read-write locks, which may be cumbersome for the non-expert to use. Our model still protects the programmer against the two most common types of errors, race conditions, and deadlocks. Shared data spaces are used to implement concurrent data structures (such as a shared cache) that can then be reused by non-experts.

Race conditions are automatically avoided, because a thread needs to lock a shared data space before it can access the objects within. Failure to lock a data space causes the data space descriptor check to fail with a runtime error.

To avoid deadlocks, we require that there be a partial order on locks and that locks are acquired following that order. That means that when a thread acquires a lock B while holding a lock A, then A must be less than B based on that partial order. This order does not have to be specified by the user, but it has to exist. To ensure that it exists, we record for each lock the set of *successor locks* that were acquired while it was being held. Then, when a lock A is acquired, GAP checks that none of the successor locks of A are currently being held.

This implementation incurs little or no overhead if there is no nesting of locks, which is a very common case [1]. It also requires no explicit annotation (unlike [3], which uses the same basic idea), which would be cumbersome for the non-expert user, as such annotations tend to be often required even in otherwise purely sequential code that uses parallelised libraries.

The public data space only contains atomic objects, i.e. objects that support solely atomic operations and thus can be acted on by any number of threads concurrently (this includes all fully immutable objects). This is both a convenience feature so that the programmer does not have to write explicit locking code for simple objects and allows more efficient access to objects that can be implemented without locking (such as atomic counters or lock-free queues).

A very common use case for atomic objects is that several types of GAP objects have attributes that accumulate and cache information that is often expensive to compute. Adding to accumulated data is an idempotent operation (the result is always the same, even if it is calculated repeatedly) and two threads can perform it concurrently without locking.

[2]Our programming model intentionally does not prescribe a specific scheduling algorithm. Scheduling logic can and should be encapsulated in parallel skeletons [2] to allow problem-specific optimizations, as well as general scheduling algorithms.

[3]Programming is meant to be similar to pure message passing approaches, but allows for multiple threads to share data (primarily, because we send references of objects between threads rather than copies of objects). Thus, unlike pure message passing approaches, we need to protect against race conditions and other typical shared memory errors.

[4]For some cases, explicit memory barriers are needed.

4. REFERENCES

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

[2] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 718–731, London, UK, 1996. Springer-Verlag.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, volume 37 (11) of *ACM SIGPLAN Notices*, pages 211–230. ACM, Nov. 2002.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[5] G. Cooperman. *ParGAP – Parallel GAP, Version 1.1.2*, 2004. GAP package, `http://www.ccs.neu.edu/home/gene/pargap.html`.

Author Index