International Journal of Foundations of Computer Science
Vol. 22, No. 5 (2011) 1035–1055
© World Scientific Publishing Company
DOI: 10.1142/S0129054111008556

BALANCED DENSE POLYNOMIAL MULTIPLICATION ON MULTI-CORES

MARC MORENO MAZA
Ontario Research Centre for Computer Algebra
University of Western Ontario
London Ontario, N6A 5B7, Canada
moreno@csd.uwo.ca
http://www.csd.uwo.ca/moreno

YUZHEN XIE
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge Massachusetts, MA 02139, USA
yxie@csail.mit.edu
http://people.csail.mit.edu/yxie/

Received 1 March 2010
Accepted 21 June 2010
Communicated by Akihiro Fujiwara

In symbolic computation, polynomial multiplication is a fundamental operation akin to matrix multiplication in numerical computation. We present efficient implementation strategies for FFT-based dense polynomial multiplication targeting multi-cores. We show that *balanced input data* can maximize parallel speedup and minimize cache complexity for bivariate multiplication. However, unbalanced input data, which are common in symbolic computation, are challenging. We provide efficient techniques, that we call *contraction* and *extension*, to reduce multivariate (and univariate) multiplication to *balanced bivariate multiplication*. Our implementation in `Cilk++` demonstrates good speedup on multi-cores.

Keywords: Parallel symbolic computation; parallel polynomial multiplication; parallel multi-dimensional FFT/TFT; Cilk++; multi-core.

1. Introduction

Polynomials and matrices are the fundamental objects on which most computer algebra algorithms operate. In the last decade, significant efforts have been deployed by different groups of researchers for delivering highly efficient software packages for computing symbolically with polynomials and matrices. Among them are LinBox [13], MAGMA [14] and NTL [17]. However, most of these works are dedicated to sequential implementation, in particular in the case of polynomials. None of the computer algebra software packages available today offers parallel implementation of asymptotically fast algorithms for polynomial arithmetic. The work reported hereafter aims at filling this gap.

We present high-performance techniques for the implementation of multivariate polynomial multiplication targeting multi-cores. Symbolic computations with polynomials rely, indeed, directly or indirectly on multiplication. We commit ourselves to polynomials over finite fields since the so-called *modular techniques* reduce all computations to such coefficient fields. In addition, we focus on dense polynomial arithmetic because most computer algebra algorithms, such as the Euclidean Algorithm and its variants, tend to densify intermediate data, even when the input and output are sparse. See Chapter 5 in [7] for an extensive presentation of these ideas.

Dense representations permit the use of multiplication algorithms based on *Fast Fourier Transform* (FFT) which runs in quasi-linear sequential time w.r.t. output size, when counting the number of operations on coefficients. This result holds for univariate as well as for multivariate polynomials. We observe that reducing multivariate multiplication to univariate one through Kronecker's substitution is not an option in our context. Indeed, this would lead us to manipulate univariate polynomials of very large degrees, say in the order of a machine word. Meanwhile, we aim at computing over the field Z/pZ where p is a machine word prime number, for efficiency reasons. Therefore, we would not always be able to find in Z/pZ the appropriate primitive roots of unity for performing a 1-D FFT.

In the multivariate case, the row-column algorithm for multi-dimensional FFT, recalled in Section 2, proceeds one dimension after another and performs several one-dimensional FFTs along one dimension at a time. This yields concurrent execution without even requiring that each one-dimensional FFT is computed in a parallel fashion. We take advantage of this flexibility to make use of non-standard and memory-efficient one-dimensional FFT techniques, such as Truncated Fourier Transform (TFT), for which no efficient parallel algorithm is known. More importantly, we do not seek a very fine grain of parallelism in our multiplication code since it will itself be a low-level routine in higher-level codes for computing polynomial GCDs and solving polynomial systems, for which parallel algorithms are available and distributed computing is desired.

Efficient implementation of algorithms on multi-cores makes necessary to consider complexity measures such as parallel speedup and cache complexity. In Section 3, we analyze the performances of dense multiplication based on the row-column multi-dimensional FFT for these complexity measures. On bivariate input and when the partial degrees of the product are equal, the performances are nearly optimal; we call *balanced* this degree configuration. When the ratio between these two partial degrees is large, our experimentation confirms that performances are low.

Motivated by these theoretical and experimental results, we show how multivariate multiplication can be efficiently reduced to *balanced bivariate multiplication*, based on 2-D FFT. With respect to a multiplication based on n-dimensional FFT, our approach may increase the input data size by at most a factor of 2. However, it provides much larger parallel speedup as reported in our experimentation.

Our approach combines two fundamental techniques that we call *contraction* and *extension*, presented in Sections 4 and 5. The first one reduces multivariate multiplication to bivariate one, without ensuring that dimension sizes are equal; however, the work remains unchanged and in many practical cases the parallelism and cache complexity are improved substantially.

The technique of extension turns univariate multiplication to bivariate one. This has several applications. First, it permits to overcome the difficult cases where primitive roots of unity of "large" orders cannot be found in the field of coefficients. Secondly, combined with the technique of contraction, this leads in Section 6 to balanced bivariate multiplication.

The techniques proposed in this paper are implemented in the Cilk++ language [9], which extends C++ to the realm of multi-core programming based on the multi-threaded model realized in [6]. The Cilk++ language is also equipped with a provably efficient parallel scheduler by work-stealing [2]. We use the sequential C routines for 1-D FFT and 1-D TFT from the `modpn` library [11]. Our integer arithmetic modulo a prime number relies also on the efficient functions from `modpn`, in particular the improved Montgomery trick [15], presented in [12]. This trick is another important specific feature of 1-D FFTs over finite fields

All our benchmarks, except the last one, are carried out on an Intel 16-core machine with 16 GB memory. Its processors are Xeon E7340 @ 2.40 GHz. Both $L1$ instruction cache and $L1$ data cache have 32 KB and are 8-way set-associative. There is a 16-way set-associative $L2$ unified cache with 4 MB. The cache line size of both $L1$ and $L2$ is 64 bytes. The last benchmark is a repeat of the second last one using 16 cores on an AMD 32-core machine with 128 GB memory. All the CPUs are Opteron 8354 @ 2.2 GHz. This machine has three layers of cache. The $L1$ instruction and $L1$ data caches both have 64 KB and are 2-way set-associative. Each quad-core shares a 512 KB 8-way set-associative $L2$ cache. All the cores share a 32-way set-associative $L3$ cache with 2 MB. At each level, the cache line size is 64 bytes.

2. Background

Throughout this paper $\mathbb{K}$ designates the finite field Z/pZ with p elements, where $p > 2$ is a prime number. In this section, we review algorithms and complexity results for multiplying multivariate polynomials over $\mathbb{K}$ by means of FFT techniques. We start by stressing the specific features of FFT computations over finite fields.

2.1. *FFTs over finite fields*

Using the Cooley-Tukey algorithm [3] (and other algorithms such as Bluestein's) one can compute the *Discrete Fourier Transform* (DFT) of a vector of s complex numbers within $O(s \lg(s))$ scalar operations. For vectors with coordinates in the prime field $\mathbb{K}$ two difficulties appear with respect to the complex case.

First, in the context of symbolic computation, it is desirable to restrict ourselves to radix 2 FFTs since the radix must be invertible in $\mathbb{K}$ and one may want to keep the ability of computing modulo small primes p, even $p = 3, 5, 7, \ldots$ for certain types of modular methods, such as those for polynomial factorization; see [7] for details. As a consequence the FFT of a vector of size s over $\mathbb{K}$ has the same running time for all s in a range of the form $[2^{\ell}, 2^{\ell+1})$. This *staircase* phenomenon can be smoothened by the so-called *Truncated Fourier Transform* (TFT) [8]. In most practical cases, the TFT performs better in terms of running time and memory consumption than the radix-2 Cooley-Tukey Algorithm; see the experimentation reported in [12]. However, the TFT has its own practical limitations. In particular, no efficient parallel algorithm is known for it.

Another difficulty with FFTs over finite fields comes from the following fact: a primitive s-th root of unity exists in $\mathbb{K}$ if and only if s divides $p-1$. Therefore, the product of two univariate polynomials f, g over $\mathbb{K}$ can be computed by evaluation and interpolation based on the radix 2 Cooley-Tukey Algorithm (see the algorithm of Section 2.2 with $n = 1$) if and only if the degree d of the product fg is less than the largest power of 2 dividing $p-1$. When this holds, computing fg amounts to $\frac{9}{2}\lg(s)s+3s$ operations in $\mathbb{K}$ using the Cooley-Tukey Algorithm (and $\frac{9}{2}(\lg(s)+1)(d+1)+3s$ operations in $\mathbb{K}$ using TFT) where s is the smallest power of 2 greater than d. When this does not hold, one can use other techniques, such as the Schönage-Strassen Algorithm [7] which introduces "virtual primitive roots of unity". However, this increases the running time to $O(s\lg(s)\lg(\lg(s)))$ scalar operations.

2.2. *Multivariate multiplication*

Let $f, g \in \mathbb{K}[x_1, \ldots, x_n]$ be two multivariate polynomials with coefficients in $\mathbb{K}$ and with n ordered variables $x_1 < \cdots < x_n$. For each i, let d_i and d'_i be the degree in x_i of f and g respectively. For instance, if $f = x_1^3x_2 + x_3x_2^2 + x_3^2x_1^2 + 1$ we have $d_1 = 3$ and $d_2 = d_3 = 2$. We assume the existence of primitive s_i-th roots of unity ω_i, for all i, where s_i is a power of 2 satisfying $s_i > d_i + d'_i$. Then, the product fg is computed as follows.

Step 1: Evaluate f and g at each point of the n-dimensional grid $((\omega_1^{e_1}, \ldots, \omega_n^{e_n}), 0 \le e_1 < s_1, \ldots, 0 \le e_n < s_n)$ via multi-dimensional FFT.
Step 2: Evaluate fg at each point P of the grid, simply by computing $f(P)\,g(P)$,
Step 3: Interpolate fg (from its values on the grid) via multi-dimensional FFT.

Assuming that, for $1 \le i \le n$, for $0 \le e_j < s_i$, all $\omega_i^{e_j}$ are precomputed, the above procedure amounts to

$$\frac{9}{2}\sum_{i=1}^{n}(\prod_{j\neq i} s_j)s_i\lg(s_i) + (n+1)s = \frac{9}{2}s\lg(s) + (n+1)s \qquad (1)$$

operations in $\mathbb{K}$, where $s = s_1\cdots s_n$. In practice the benefit of using 1-D TFT instead of 1-D FFT increases with the number of variables and the number of cores in use. The cut-off criteria and a detail performance evaluation are reported in [16]. Consider now the following map from the monomials of $\mathbb{K}[x_1, \ldots, x_n]$ to those of $\mathbb{K}[x_1]$:

$$x_1^{e_1} x_2^{e_2} x_3^{e_3} \cdots x_n^{e_n} \quad \longmapsto \quad x_1^{e_1+\alpha_2 e_2+\alpha_3 e_3+\cdots+\alpha_n e_n}$$

where $\alpha_2 = d_1 + d'_1 + 1$, $\alpha_3 = \alpha_2(d_2 + d'_2 + 1)$, $\ldots$, $\alpha_n = \alpha_{n-1}(d_{n-1} + d'_{n-1} + 1)$. This induces a polynomial ring homomorphism Ψ (called *the Kronecker substitution*) from $\mathbb{K}[x_1, \ldots, x_n]$ to $\mathbb{K}[x_1]$, hence a map satisfying $\Psi(a + b) = \Psi(a) + \Psi(b)$ and $\Psi(ab) = \Psi(a)\Psi(b)$, for all polynomials a, b. Moreover, one can check that fg is the only pre-image of $\Psi(f)\Psi(g)$. This latter polynomial has degree

$$\delta_n := (d_1 + d'_1 + 1) \cdots (d_n + d'_n + 1) - 1.$$

It follows from this construction that one can reduce multivariate multiplication to univariate one. If $\mathbb{K}$ admits primitive s-th roots of unity for $\delta_n < s = 2^\ell$ for some ℓ, then using the FFT-based multiplication, one can compute fg in at most $\frac{9}{2}\lg(s)s + 3s$ operations in $\mathbb{K}$. Using the TFT approach, this upper bound becomes $\frac{9}{2}(\lg(s) + 1)(\delta_n + 1) + 3s$.

Multivariate multiplications based on multi-dimensional FFT and Kronecker's substitution have similar sequential running time. However, the latter approach has at least two drawbacks. First, the field $\mathbb{K}$ may not admit primitive s-th roots of unity. Recall that primitive s-th roots of unity exist in $\mathbb{K}$ if and only if s divides $p - 1$, see [7] . Secondly, as mentioned above, it is desirable to achieve efficient parallel multiplication without assuming that 1-D FFTs are performed in a parallel fashion.

3. Main Results

The specific features of 1-D FFTs over finite fields, see Section 2.1, lead us to the following hypothesis. We assume throughout this paper that we have at our disposal a **black box** computing the DFT at a primitive 2^ℓ-th root of unity (when $\mathbb{K}$ admits such value) of any vector of size s in the range $(2^{\ell-1}, 2^\ell]$ in time $O(s\lg(s))$. However, we do not make any assumptions about the algorithm and its implementation. In particular, we do not assume that this implementation is a parallel one. As mentioned in the introduction, we do not seek a very fine-grained parallelism for our polynomial multiplication since it is meant to be a core-routine in higher-level parallel code. Therefore, we rely on the row-column multi-dimensional FFT to create concurrent execution in the multiplication algorithm presented in Section 2.2.

This strategy has its own challenges. Suppose that the dimension x_1 has a small size s_1, say in the order of units, whereas the dimension x_2 has a size s_2 in the thousands. This implies that a lot of small FFTs have to be performed along x_1 while only a few large FFTs can be run simultaneously along x_2. Therefore, along x_1, the parallelization overhead may dominate, reducing severely the benefits of concurrent runs. Meanwhile, along x_2, the measured speedup factor may simply be too small by lack of parallelism.

We formalize this remark in Section 3.1 where we give a lower bound for the parallel running time of the algorithm of Section 2.2. Then, in Section 3.2 we give an upper bound for the number of cache misses of the same algorithm. We observe that these lower and upper bounds reach a "local" maximum and minimum respectively, when the number n of variables equals 2 and the dimension sizes of the 2-D FFT are equal. Therefore, the algorithm of Section 2.2 performs very well in terms of parallelism and cache complexity on bivariate polynomial input when the partial degrees of the product are equal. For this reason, we introduce in Section 3.3 the concept of *balanced bivariate multiplication.*

In Section 3.4, we claim that dense multivariate multiplication can be efficiently reduced to balanced bivariate multiplication. "Efficiently" means here that the overheads of the reduction are in general much less than the performance gains. Sections 4 to 6 formally establish this reduction and demonstrate its performances.

3.1. *Parallel running time estimates*

Let us consider the parallel running time of the algorithm of Section 2.2 with the multi-threaded programming model of [6]. Under the assumption that 1-D FFT may be run sequentially, the following estimate holds for the span of ***Step 1***:

$$\frac{9}{2}\left(s_1 \lg(s_1) + \cdots + s_n \lg(s_n)\right) \tag{4}$$

if the parallelization of the for-loops are omitted, for instance in the PRAM model. Since we rely on the *fork-join* multi-threaded model of [6], we shall take these overheads into account as follows. Following the way a `cilk_for` loop is implemented in the `cilk++` language [9], we assume that the span of a for-loop of the form

for $i = 1 \cdots n$ **do** BODY(i); **end for**;

is upper bounded by $O(\lg(n) + S)$ where S is the maximum span of `BODY(i)` for `i` in the range $1 \cdots n$. Consequently, the term $\sum_{i=0}^{i=n} \lg(s/s_i) = (n-1)\lg(s)$ should be added to the expression in (4). However, when all s_i are large enough, say greater than n, which is a reasonable assumption in practice, this additional term can be neglected. Therefore, the parallelism (i.e. theoretical speedup) of ***Step 1*** equals

$$\frac{s_1 \cdots s_n \lg(s_1 \cdots s_n)}{s_1 \lg(s_1) + \cdots + s_n \lg(s_n)} \tag{5}$$

which is lower bounded by $s/\max_{i=1\cdots n}(s_i)$. Similar estimates can be given for ***Step 3*** while the costs of ***Step 2*** can be neglected comparing to the others. Observe that for a fixed n and a fixed s, the quantity $\max_{i=1\cdots n}(s_i)$ is lower bounded by $s^{1/n}$. Therefore, under these conditions the lower bound $s/\max_{i=1\cdots n}(s_i)$ is maximized when each dimension size is equal to $s^{1/n}$; in this case the parallelism is given by

$$s/(s^{1/n}). \tag{6}$$

These calculations suggest that the dimension sizes $s_1, \ldots, s_n$ should be equal in order to maximize parallelism.

3.2. *Cache complexity estimates*

We now turn to cache complexity estimates, using the theoretical model introduced in [5]. We focus on ***Step 1*** again. Let L be the size of a cache line. We assume that the cache size is large enough such that all data involved by P concurrent runs of 1-D FFT (where P is the number of processors) fit in cache. This is justified in the experimentation with our *balanced bivariate multiplication* (see Section 6) where our 16-core machine has 4MB of L2 unified cache and each FFT vector has at most size 128KB. We also assume that our n-D FFT is performed without data transposition by loading directly from main memory to cache the vectors on which 1-D FFT is run. This technique is used in the implementation of the FFTW [4]. Therefore, cache misses arise essentially when reading data before performing a 1-D FFT. For a vector of size s_i the number of cache misses is at most $s_i/L+1$. Thus the number of cache misses at ***Step 1*** and ***Step 3*** fits in

$$O(\Sigma_{i=1\cdots n}\,(\Pi_{j\neq i}s_j)(s_i/L+1)).$$

At ***Step 2***, this number is within $O(\frac{s}{L}+1)$. Hence, if $Q(s_1,\ldots,s_n)$ denotes the total number of cache misses for the whole algorithm, we obtain

$$Q(s_1,\ldots,s_n)\leq cs\frac{n+1}{L}+cs(\frac{1}{s_1}+\cdots+\frac{1}{s_n}) \tag{8}$$

for some constant c. As in Section 3.1, let us consider $s=s_1\cdots s_n$ to be fixed. The following is easy to prove:

$$\frac{n}{s^{1/n}}\leq\frac{1}{s_1}+\cdots+\frac{1}{s_n}.$$

Moreover this latter inequality is an equality when each s_i equals $s^{1/n}$. Noting $\frac{n+1}{n}\leq 2$ for $n\geq 1$ we deduce:

$$Q(s_1,\ldots,s_n)\leq ncs(\frac{2}{L}+\frac{1}{s^{1/n}}) \tag{10}$$

when $s_i=s^{1/n}$ holds for all i. This suggests to minimize n, thus setting $n=2$. Therefore, for a fixed s, the upper bound of (8) reaches a local minimum at $n=2$ and $s_1=s_2=\sqrt{s}$.

3.3. *Balanced bivariate multiplication*

The analysis of Sections 3.1 and 3.2 suggests that, for a fixed n, the algorithm of Section 2.2 is nearly optimum in terms of parallelism and cache complexity when $s_1=\cdots=s_n$, that is, when the partial degrees of the product are equal. This yields the definition and proposition below.

Definition 1. *The pair of polynomials $f,g\in\mathbb{K}[x_1,\ldots,x_n]$ is* balanced *if all the partial degrees of their product are equal, that is, if $d_1+d'_1=d_i+d'_i$ holds for all $2\leq i\leq n$.*

Proposition 1. *Under our assumption of 1-D FFT black box, for two multivariate polynomials f, g, the theoretical speedup of the algorithm in Section 2.2 is $(s_1 \cdots s_n \lg(s_1 \cdots s_n))/(s_1 \lg(s_1) + \cdots + s_n \lg(s_n))$ and its cache complexity is within $O(s\frac{n+1}{L} + s(\frac{1}{s_1} + \cdots + \frac{1}{s_n}))$. For fixed s and n, these bounds are respectively maximized and minimized when the pair f, g is balanced. In addition, when s and n vary, these bounds reach respectively a local maximum and a local minimum whenever $n = 2$ and $s_1 = s_2 = \sqrt{s}$ both hold.*

We present here experimental results which confirm the above analysis. Figure 1 provides speedup factors of our program for multiplying different balanced pairs of bivariate polynomials. The number associated with each curve is the common partial degree of the input. This illustrates the good performances of the algorithm of Section 2.2 for such input. For the partial degree 8191, our implementation reaches a speedup factor of 14 on 16 cores. Figure 2 provides speedup factors of different pairs which are either non-bivariate or non-balanced. The performances reported there are clearly much less satisfactory than what could be observed on Figure 1. Note that these poor results are obtained on both non-balanced bivariate and balanced non-bivariate input.

We also use VTune [10] to evaluate the instruction efficiency and cache efficiency of our multiplication code run with balanced bivariate input polynomials and compared to non-bivariate and/or non-balanced input polynomials. More precisely, we study a pair of bivariate polynomials with partial degrees both equal to 8191. Then, the non-balanced bivariate and balanced non-bivariate input pairs of polynomials are designed so that for each pair the size of the product is similar.

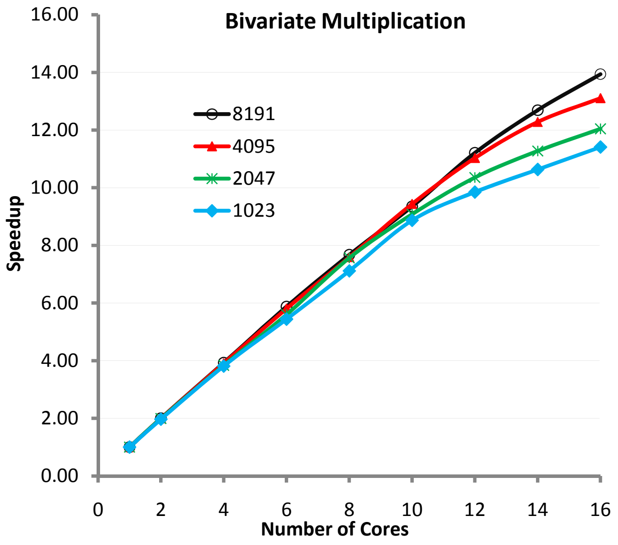

Fig. 1. Speedup of bivariate multiplication on balanced input on an Intel Xeon multi-core.

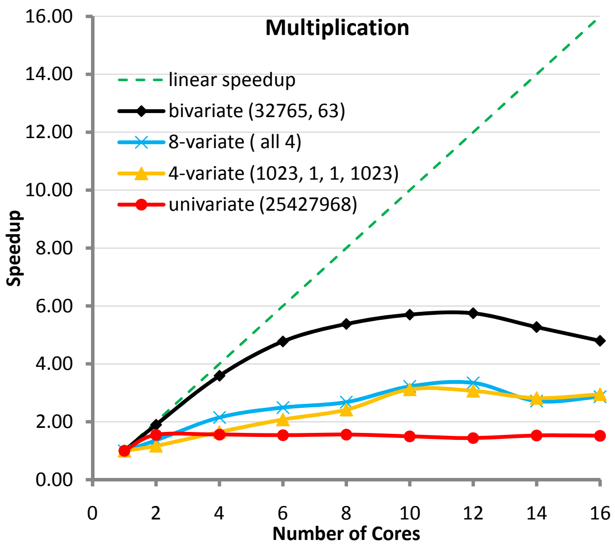

Fig. 2. Speedup of multiplication for non-bivariate or non-balanced input on an Intel Xeon multi-core.

Table 1. Features of the problems.

No.	Type	Size of input polynomial (int)	Product size (int)
1	balanced bivariate	8191×8191	268402689
2	un-balanced bivariate	259575×258	268401067
3	balanced 4-variate	63×63×63×63	260144641
4	balanced 8-variate	5×5×5×5×5×5×5×5	214358881

Table 2. Performance evaluation by VTune.

No.	INST_ RETIRED. ANY$\times 10^9$	Clocks per Instruction Retired	L2 Cache Miss Rate $(\times 10^{-3})$	Modified Data Sharing Ratio $(\times 10^{-3})$	Time on 8 Cores (s)
1	659.555	0.810	0.333	0.078	16.15
2	713.882	0.890	0.735	0.192	19.52
3	714.153	0.854	1.096	0.635	22.44
4	1331.340	1.418	1.177	0.576	72.99

Thus, all these problems are comparable in terms of work. The features of these problems are summarized in Table 1.

Table 2 lists a selection of events and ratios reported by VTune on each run for computing the product of an input pair. Due to the availability of VTune in our laboratory, these measurements are done on a 8-core machine with 8 GB memory. Each processor is an Intel Xeon X5460 @3.16GHz and has 6144 KB of L2 cache.

The number of instructions retired for the balanced bivariate case is twice less than the balanced 8-variate case. The clocks per instruction retired for the balanced bivariate case is also the least among all of them. This indicates the good instruction efficiency of our balanced bivariate multiplication. The L2 cache miss ratio for the balanced bivariate case is as low as 0.0003, while the others are 2 to 4 times higher. The very small modified data sharing ratios (less than 0.0001) for the balanced bivariate case imply that, chances of threads racing on using and modifying data laid in one cache line are very low. However, it is 2 to 7 times higher for the non-balanced bivariate or balanced 4-variate and 8-variate cases. This shows the superior cache efficiency of the balanced bivariate multiplication. Consequently, the timing on 8 cores for the balanced bivariate case is the best, about 4.5 times faster than the balanced 8-variate case. These experimental results are coherent with our previous theoretical analysis.

3.4. *Reduction to balanced bivariate multiplication*

The results of Sections 3.1 to 3.3 indicate that a reduction to bivariate multiplication with balanced input could improve the performance of multivariate multiplication based on FFT techniques. This is, indeed, possible and we achieve this reduction through the rest of the paper.

In Section 4 we describe a first fundamental technique, that we call *contraction*. This generalization of Kronecker's substitution allows us to turn a n-variate multiplication (for $n > 2$) into a bivariate multiplication without any overheads in terms of sequential running time. This technique provides performance improvements on many practical cases.

In Section 5, we study how univariate polynomial multiplication can be performed efficiently via bivariate multiplication based on 2-D TFT. This technique, that we call *extension*, has several motivations. First, under our assumption of 1-D FFT black-box (which may be a sequential program) this trick creates concurrent execution for FFT-based univariate multiplication. Secondly, when the base field $\mathbb{K}$ does not possess primitive roots of unity of sufficiently large orders for performing a Cooley-Tukey radix-2 FFT, this trick can reduce the computations to a case where this latter algorithm can be applied. Finally, this technique of extension, together with that of contraction, is the basis of dense multivariate multiplication via *balanced bivariate multiplication*, presented in Section 6.

Figure 3 gives the timing of a 4-variate multiplication with non-balanced input via 4-D TFT method vs balanced 2-D TFT method. Even on 1 core, the balanced 2-D TFT method is 2.5 times faster. The work for the two methods is essentially the same. However, our balanced 2-D TFT method has better cache efficiency. Moreover, it scales well on 16 cores. As a result, the total "net speedup" using balanced 2-D TFT method instead of the direct 4-D TFT method reaches 31 on 16 cores.

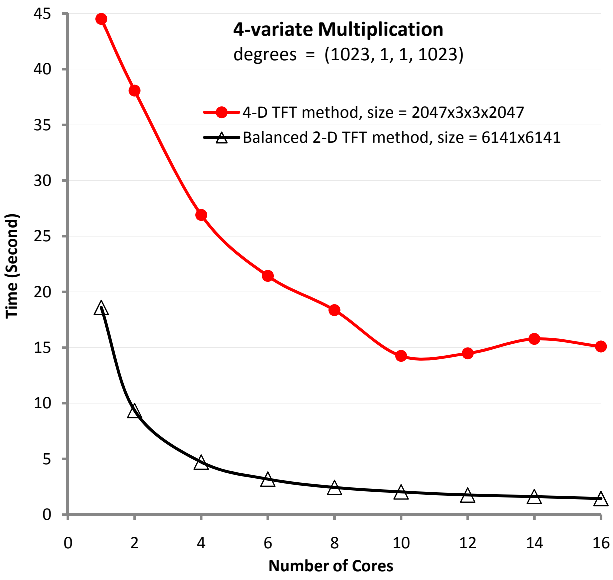

Fig. 3. Timing of 4-variate multiplication with unbalanced input via 4-D TFT vs balanced 2-D TFT methods on an Intel Xeon multi-core.

4. Contraction

Before introducing the concept of contraction in Definition 3, we specify in Definition 2 how polynomials are represented in our implementation. Proposition 2 states that contraction can be performed essentially "at no cost" with this representation. The experimental results reported at the end of this section illustrate the benefits of contraction. The notations are those introduced in Section 2.

Definition 2. *Let $\ell_1, \ldots, \ell_n$ be positive integers such that $\ell_i > d_i$ holds for all $1 \leq i \leq n$. Define $\underline{\ell} := (\ell_1, \ldots, \ell_n)$. We call $\underline{\ell}$-*recursive dense representation *(RDR, for short) of f any one-dimensional array F of size $\ell := \ell_1 \cdots \ell_n$ and with integer indices in the range $0 \cdots (\ell - 1)$ such that the following two conditions hold.*

(i) the coefficient in f of the monomial $x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n}$, for $0 \leq e_i \leq d_i$, is in the slot $F[j]$ of F with index $j = e_1 + \ell_1 e_2 + \ell_1 \ell_2 e_3 + \cdots + (\ell_1 \cdots \ell_{n-1}) e_n$,

(i) the coefficient $F[j]$ is 0 whenever the index j equals $e_1 + \ell_1 e_2 + \ell_1 \ell_2 e_3 + \cdots + (\ell_1 \cdots \ell_{n-1}) e_n$ where $0 \leq e_i < \ell_i$ for $1 \leq i \leq n$ and $d_i < e_i$ holds for some i; such a coefficient $F[j]$ is called a padding zero.

Remark 1. *When $n = 1$, an $\underline{\ell}$-RDR of f is given by any vector F of size at least $d_1 + 1$ where $F[i]$ is the coefficient of x_1^i in f when $0 \leq i \leq d_1$ and 0 otherwise. Consider now $n > 1$ and let $\ell_1, \ldots, \ell_n$ be positive integers such that $\ell_i > d_i$ holds for all $1 \leq i \leq n$. For all $0 \leq i \leq d_n$, let $c_i \in \mathbb{K}[x_1, \ldots, x_{n-1}]$ be the coefficient of f regarded as a univariate polynomial in x_n and C_i be an $(\ell_1, \ldots, \ell_{n-1})$-RDR of c_i.*

Let $Z_{d_n+1}, \ldots, Z_{\ell_n-1}$ be zero-vectors, all of size $\ell_1 \cdots \ell_{n-1}$. Then an $\underline{\ell}$-RDR of f is obtained by concatenating $C_0, C_1, \ldots, C_{d_n}, Z_{d_n+1}, \ldots, Z_{\ell_n-1}$ in this order. This fact justifies the term recursive dense representation.

Recall how the product fg can be computed in parallel via n-dimensional FFT in the context of our implementation. During ***Step 1*** and ***Step 3*** of the algorithm of Section 2.2, n-dimensional FFT's are performed by computing in parallel one-dimensional FFT's along x_i, for $i = 1, \ldots, n$ successively. As pointed out in Section 3, this approach suffers from the following bottleneck. In practice (and in particular when solving systems of polynomial equations) the partial degree d_1 is likely to be large whereas $d_2, \ldots, d_n$ are likely to be as small as 1 or 2. This implies that the n-dimensional FFT approach will compute a lot of "small 1-D FFTs" (whereas such 1-D FFTs of short vectors are not worth the game) and only a few "large 1-D FFTs" concurrently (leading to poor parallelism). To deal with this "unbalanced work" the techniques developed in this paper aim at transforming the polynomials f and g into bivariate ones in a way that they can be efficiently multiplied by a 2-D FFT approach. In this section, we accomplish a first step toward this goal using the notion of *contraction*.

Definition 3. *Let $\ell_1, \ldots, \ell_n$ be positive integers such that $\ell_i > d_i$ holds for all $1 \leq i \leq n$. Let m be an integer satisfying $1 \leq m < n$. Define $\alpha_1 = 1$, $\alpha_2 = \ell_1$, $\alpha_3 = \ell_1\ell_2$, $\ldots$, $\alpha_m = \ell_1\ell_2 \cdots \ell_{m-1}$, $\alpha_{m+1} = 1$, $\alpha_{m+2} = \ell_{m+1}$, $\ldots$, $\alpha_n = \ell_{m+1}\ell_{m+2} \cdots \ell_{n-1}$. Then, we set $\underline{\alpha} := (\alpha_1, \ldots, \alpha_n)$. Consider the following map from the monomials of $\mathbb{K}[x_1, \ldots, x_n]$ to those of $\mathbb{K}[x_1, x_{m+1}]$:*

$$x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n} \quad \longmapsto \quad x_1^{c_1} x_{m+1}^{c_2}$$

*where $c_1 = \alpha_1 e_1 + \cdots + \alpha_m e_m$ and $c_2 = \alpha_{m+1} e_{m+1} + \alpha_{m+2} e_{m+2} + \cdots + \alpha_n e_n$. This induces a polynomial ring homomorphism $\Psi_{\underline{\alpha}}$, from $\mathbb{K}[x_1, \ldots, x_n]$ to $\mathbb{K}[x_1, x_{m+1}]$, that we call $\underline{\alpha}$-*contraction*. Hence, this map satisfies $\Psi_{\underline{\alpha}}(ab) = \Psi_{\underline{\alpha}}(a)\Psi_{\underline{\alpha}}(b)$, for all polynomials a, b.*

Proposition 2. *With the notations of Definition 3, define $t_1 = \ell_1\ell_2 \cdots \ell_m$, $t_2 = \ell_{m+1} \cdots \ell_n$ and $\underline{t} = (t_1, t_2)$. Let F be an one-dimensional array of size $t_1 t_2 = \ell_1 \cdots \ell_n$. If F is an ℓ-RDR of f, then F is also a $\underline{t}$-RDR of $\Psi_{\underline{\alpha}}(f)$. Conversely, if F is a $\underline{t}$-RDR of $\Psi_{\underline{\alpha}}(f)$, then it is an ℓ-RDR of f.*

Proposition 2 follows easily from the *recursive structure* of RDR's, as pointed out in Remark 1. We explain now how we make use of contraction for computing the product fg. We define $\ell_i := d_i + d'_i + 1$, for all i. Then, we set $\underline{\ell} := (\ell_1, \ldots, \ell_n)$. Let F and G be $\underline{\ell}$-RDR of f and g respectively. We choose an integer m such that the "distance" given by $|\ell_1 \cdots \ell_m - \ell_{m+1} \cdots \ell_n|$ is minimum. With these values for $\underline{\ell}$ and m, consider the $\underline{\alpha}$-contraction of Definition 3. Then $\Psi_{\underline{\alpha}}(f)$ and $\Psi_{\underline{\alpha}}(g)$ are two bivariate polynomials in $\mathbb{K}[x_1, x_m]$. By the choice of ℓ_i's, the polynomial fg is the only pre-image of $\Psi_{\underline{\alpha}}(f) \times \Psi_{\underline{\alpha}}(g)$ under $\Psi_{\underline{\alpha}}$. Therefore, this map can be used to compute fg via a 2-D FFT approach. Moreover, it follows from Proposition 2 that this change of representation is made at no cost! In addition, by the choice of m, the degrees of $\Psi_{\underline{\alpha}}(fg)$ w.r.t. x_1 and x_{m+1} are as close to each other as possible.

Let us compare the work, the parallelism and the cache complexity of the multiplication based on n-D FFT with the multiplication based on contraction and 2-D FFT approach. To keep the discussion simple, let us assume that we can choose $s_i = \ell_i$ for all i. (Recall that s_i is the size of our 1-D FFT input vectors along x_i.) This is actually realistic if all 1-D FFTs are computed by TFT, which is the case in our implementation. It follows from Proposition 2 and Expression (1) that the work is unchanged. The inequality

$$\frac{3}{L} + (\frac{1}{s_1 \cdots s_{m-1}} + \frac{1}{s_m \cdots s_n}) \leq \frac{n+1}{L} + (\frac{1}{s_1} + \cdots + \frac{1}{s_n})$$

combined with Expression (8) suggests that contraction is likely to reduce cache misses. As discussed in Section 3.1, “contracting dimensions” will keep enough theoretical speedup while reducing parallel overhead.

Experimental results. In our experimentation illustrated in Figure 4, we study the case of multiplying two 4-variate polynomials f and g. Their partial degrees d_2, d_3, d'_2, d'_3 are all equal to 1 while $d_1 = d'_1$ and $d_4 = d'_4$ vary in the range $1024 \cdots 2047$. These degree patterns are typical in computing normal forms based on the algorithm in [12].

On 1 core, we compare three methods for computing the product fg: direct 4-D TFT, 1-D TFT via Kronecker's substitution (see Section 2.2) and our contraction to 2-D method. We should observe first that the Kronecker substitution method fails

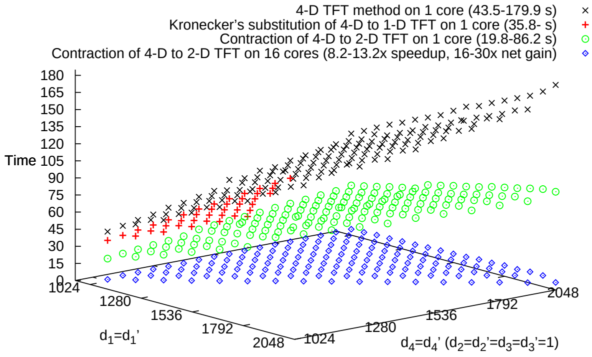

Fig. 4. Timing (s) for 4-variate multiplication by direct 4-D TFT on 1 core vs Kronecker's substitution on 1 core vs contraction from 4-D to 2-D TFT on 1 core and 16 cores on an Intel Xeon multi-core.

on most input due to the fact that $\mathbb{K}$ does not have primitive roots of unity of sufficiently large order; in those cases our contraction method clearly outperforms the direct 4-D TFT method, which was expected, based on our complexity estimates. When the Kronecker substitution method does not fail, our contraction method is clearly the most efficient technique. The fact that the contraction outperforms Kronecker's substitution in this case can probably be explained by cache complexity arguments, see the discussion on FFT computation in [5]. The results also show that the contraction method scales very well on 16 cores, reaching a speedup factor between 8.2 to 13.2 for this large range of problems. The net speedup on 16 cores w.r.t. the direct 4-D method on 1 core is between 16 and 30.

5. Extension

Let $f, g \in \mathbb{K}[x_1]$ be two non-constant univariate polynomials with coefficients in the prime field $\mathbb{K} = Z/pZ$ and with respective degrees d_f, d_g. Let $b \geq 2$ be an integer. We consider the map Φ_b from $\mathbb{K}[x_1]$ to $\mathbb{K}[x_1, x_2]$ that replaces any monomial x_1^a by $x_1^u x_2^v$, where u, v are the remainder and quotient in the Euclidean division of a by b, and that leaves all coefficients unchanged. More formally Φ_b is the canonical ring homomorphism from $\mathbb{K}[x_1]$ to the residue class ring $\mathbb{K}[x_1, x_2]/\langle x_1^b - x_2 \rangle$.

We determine a value for b such that the product fg can be obtained by computing $\Phi_b(f)\Phi_b(g)$ using a number of operations in $\mathbb{K}$ which, essentially, is at most twice the number of operations for a direct computation in $\mathbb{K}[x_1]$. Moreover, we impose that the pair $\Phi_b(f), \Phi_b(g)$ is balanced, or nearly balanced. Indeed, the cache complexity upper bound given by Expression (8) is minimized in this case.

To this end, we need some notation. Let s_1, s_2 be positive integers and F, G, H be (s_1, s_2)-RDR's of $\Phi_b(f)$, $\Phi_b(g)$ and $\Phi_b(f)\Phi_b(g)$. According to the complexity results of Section 3, we shall determine b, s_1, s_2 such that both $s := s_1 s_2$ and $|s_1 - s_2|$ are as small as possible in order to reduce work and cache complexity, and improve parallelism as well. Let q_f, r_f (resp. q_g, r_g) be the quotient and remainder in the Euclidean division of d_f (resp. d_g) by b. Since $\Phi_b(f)\Phi_b(g)$ will be calculated by 2-D TFTs over $\mathbb{K}$, this polynomial product will be obtained as an element of $\mathbb{K}[x_1, x_2]$ (not one of $\mathbb{K}[x_1, x_2]/\langle x_1^b - x_2 \rangle$). Hence the degrees of $\Phi_b(f)\Phi_b(g)$ w.r.t. x_1 and x_2 will be at most $2b - 2$ and $q_f + q_g$, respectively. Thus we should set

$$s_1 = 2b - 1 \quad \text{and} \quad s_2 = q_f + q_g + 1. \tag{13}$$

Roughly speaking, the RDR's F, G of f and g contain at least 50% of padding zeros. More precisely, the size (i.e. number of slots) of each of the arrays F, G, H is

$$s_1 s_2 = 2b(q_f + q_g) + (s_1 - s_2) + 1$$

Elementary calculations show that the value of $r_f + r_g$, which ranges from 0 to $2(b - 1)$, has negligible impact on the value of b. This is due to the fact that b is essentially $\sqrt{1 + (d_f + d_g)/2}$. Thus we can impose $r_f = r_g = 0$. For this assumption, Lemma 1 in Section 6 states that it is always possible to choose b such that the absolute value $|s_1 - s_2|$ is at most equal to 2. This implies the following:

$$s \leq 2(d_f + d_g) + 3. \tag{15}$$

Since the size of the univariate polynomial fg is $d_f + d_g + 1$, this b "essentially" realizes our objectives of increasing the data size at most by a factor of 2, while ensuring that our 2-D TFT's will operate on (nearly) square 2-D arrays.

Once the bivariate product $\Phi_b(f)\Phi_b(g)$ is computed, one task remains: converting this polynomial to the univariate polynomial fg. This operation is non-trivial since x_2 stands for x_1^b meanwhile the degree of $\Phi_b(f)\Phi_b(g)$ w.r.t. x_1 can be larger than b, but at most equal to $2b-2$. Elementary algebraic manipulations lead to the pseudo-code below which constructs a (d_f+d_g+1)-RDR of fg from H, the (s_1, s_2)-RDR of $\Phi_b(f)\Phi_b(g)$. Recall that s_1 and s_2 have been set to $2b-1$ and $q_f + q_g + 1$ respectively. Define $d := d_f + d_g$ and $q := q_f + q_g$. This procedure clearly runs in $O(d)$ operations in $\mathbb{K}$. Finally, we obtain Proposition 3.

```
**for** $u := 0 \cdots (b-1)$ **do** $U[u] := H[u]$; **end for**;
**for** $w := 1 \cdots q$ **do**
    $X := w\,b$;
    $Z := w(2b-1)$;
    $Y := (w-1)(2b-1) + b$;
    **for** $u := 0 \cdots (b-2)$ **do**
        $U[X+u] := H[Y+u] + H[Z+u]$;
    **end for**;
    $U[X+(b-1)] := H[Z+(b-1)]$;
**end for**;
$X := (q+1)b$;
$Z := d - X$;
$Y := q(2b-1) + b$;
**for** $u := 0 \cdots Z$ **do** $U[X+u] := H[Y+u]$; **end for**;
```

Proposition 3. *Let $f, g \in \mathbb{K}[x_1]$ have respective positive degrees d_f, d_g. Let $d = d_f + d_g$. Then, one can compute from f, g a pair of bivariate polynomials $h, k \in \mathbb{K}[x_1, x_2]$ within $O(d)$ bit operations, such that the product fg can be recovered from the product hk within $O(d)$ operations in $\mathbb{K}$, and such that there exist integers s_1, s_2 satisfying the following four constraints:* $\deg(h, x_1) + \deg(k, x_1) < s_1$, $\deg(h, x_2) + \deg(k, x_2) < s_2$, $s_1 s_2 \leq 2(d_f + d_g) + 3$ *and* $|s_1 - s_2| \leq 2$.

We stress the fact that, the construction that has led to Proposition 3 permits to efficiently multiply univariate polynomials via 2-D TFT without requiring that 1-D TFTs are computed in a parallel fashion. Moreover this strategy allows us to take advantage of FFT techniques even if $\mathbb{K}$ does not admit primitive roots of unity of sufficiently large order for using the radix 2 Cooley Tukey algorithm.

Experimental results. We compare the timings of univariate polynomial multiplications based on direct 1-D TFT and our extension method to 2-D, for input

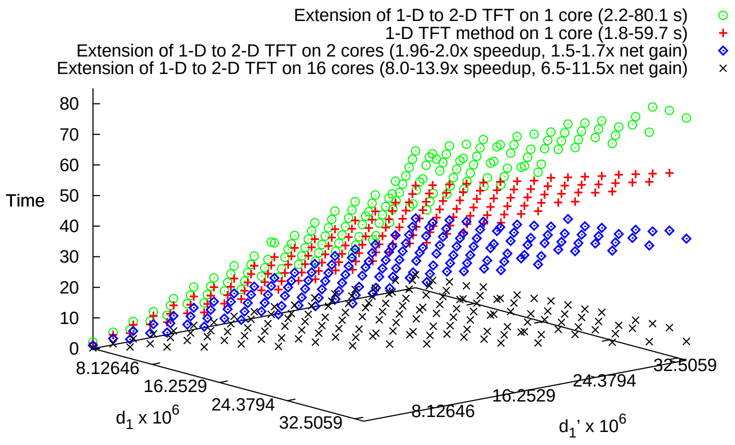

Fig. 5. Univariate multiplication timing (s) via extension to 2-D TFT on 1, 8, 16 cores vs direct 1-D TFT on an Intel Xeon multi-core.

degree ranging between 8126460 and 32505900. The results are reported on Figure 5. On 1 core, our extension method is slower than the direct 1-D TFT method for about 30%. This is not a surprise since we know that the extension from 1-D to 2-D can increase the size s of the product by (at most) a factor of 2. On 2 cores, the extension method provides a speedup factor between 1.5 and 1.7 with respect to the direct 1-D TFT method. On 16 cores, this gain ranges between 6.5 and 11.5. These data also show that extending univariate polynomials to balanced bivariate pairs can create substantial parallelism even when 1-D FFTs are executed sequentially.

6. Balanced Multiplication

We turn now to the question of performing multivariate polynomial multiplication efficiently via bivariate multiplication, based on 2-D TFTs applied to (nearly) balanced pairs. We call *balanced multiplication* the resulting strategy. As in Sections 2 and 4, let $f, g \in \mathbb{K}[x_1, \ldots, x_n]$ be two multivariate polynomials with coefficients in the prime field $\mathbb{K} = Z/pZ$ and with ordered variables $x_1 < \cdots < x_n$. For each i, let d_i and d'_i be the degree with x_i of f and g respectively.

One approach for computing the product fg via bivariate multiplication would be to convert the polynomials f and g to univariate ones via Kronecker's substitution and then, to apply the techniques of *extension* developed in Section 5. This would have the following drawback. RDR's of the images of f and g by Kronecker's substitution must have padding zeros so as to recover the product fg. The extension technique of Section 5 requires also the introduction of padding zeros. A naive combination of these two transformations would introduce far too many padding zeros. We actually checked experimentally that this approach was unsuccessful.

In this section, we develop a "short cut" which combines extension and contraction in a single transformation. In order to focus on the main ideas, we assume first that the input polynomials are in *Shape Lemma position*, see Definition 4. This assumption is actually a practical observation (formalized by the so-called *Shape Lemma* [1]) for the polynomials describing the symbolic solutions of polynomial systems with finitely many solutions. Remark 2 states how to relax this assumption.

Definition 4. *The pair $f, g \in \mathbb{K}[x_1, \ldots, x_n]$ is in* Shape Lemma position *if $(d_2 + 1) \cdots (d_n + 1) < d_1 + 1$ and $({d'}_2 + 1) \cdots ({d'}_n + 1) < {d'}_1 + 1$ both hold.*

This assumption suggests to extend the variable x_1 to two variables x_1, y such that f, g can be turned via a contraction $\Psi_{\underline{\alpha}}$ from $\mathbb{K}[x_1, y, x_2, \ldots, x_n]$ to $\mathbb{K}[x_1, y]$ into a balanced pair of bivariate polynomials.

For an integer $b \geq 2$, consider the map Φ_b that replaces any monomial x_1^a of $\mathbb{K}[x_1, x_2, \ldots, x_n]$ by $x_1^u y^v$ in $\mathbb{K}[x_1, y, x_2, \ldots, x_n]$, where u, v are the remainder and quotient in the Euclidean division of a by b, and that leaves all coefficients and other monomials unchanged. More formally Φ_b is the canonical ring homomorphism from $\mathbb{K}[x_1, x_2, \ldots, x_n]$ to the residue class ring $\mathbb{K}[x_1, y, \ldots, x_n]/\langle x_1^b - y\rangle$.

We shall determine b such that after contracting the variables $y, x_2, \ldots, x_n$ onto y in the polynomials $\Phi_b(f)\Phi_b(g)$, the resulting bivariate polynomials h and k form a balanced pair. The construction is similar to that of Section 5. Let s_1, s_2 be positive integers and H, K be (s_1, s_2)-RDR's of h and k. Define $\sigma := (d_2 + {d'}_2 + 1) \cdots (d_n + {d'}_n + 1)$. Let q_f and q_g be the quotients in the Euclidean division by b of $d_f := d_1$ and $d_g := {d'}_1$ respectively. Following the reasoning of Section 5, we set $s_1 = 2b - 1$ and $s_2 = (q_f + q_g + 1)\sigma$ and we aim at determining b such that both $s := s_1 s_2$ and $|s_1 - s_2|$ are as small as possible, in order to reduce work and cache complexity, and improve speedup factors. Since σ is regarded as small (comparing to d_1 and ${d'}_1$), Lemma 1 hereafter provides us with a candidate b. Our experimental results confirm that this choice achieves our goals.

Remark 2. *To transform a pair f, g into a pair in Shape Lemma position within $O(s)$ bit operations, we proceed as follows. We re-order the variables such that there exists an index j satisfying $1 \leq j < n$, $(d_1 + 1) \cdots (d_j + 1) \geq (d_{j+1} + 1) \cdots (d_n + 1)$ and $({d'}_1 + 1) \cdots ({d'}_j + 1) \geq ({d'}_{j+1} + 1) \cdots ({d'}_n + 1)$. Then, contract $x_1, \ldots, x_j$ to x_1. In rare cases, such a variable ordering may not exist and one can use Kronecker's substitution followed by extension.*

Experimental results. We study the performance of our balanced multiplication method for 4-variate polynomial input. All partial degrees $d_2, d_3, d_4, d'_2, d'_3, d'_4$ are set to 2 while d_1 and ${d'}_1$ range between 32768 and 65536. Figure 6 and 7 illustrates our timing results on the Intel Xeon 16-core with two levels of cache and the AMD Opteron 32-core with three levels of cache described in the introduction.

On the Xeon multi-core, we first compare our balanced multiplication with the one through Kronecker's substitution on 1 core. The latter approach performs

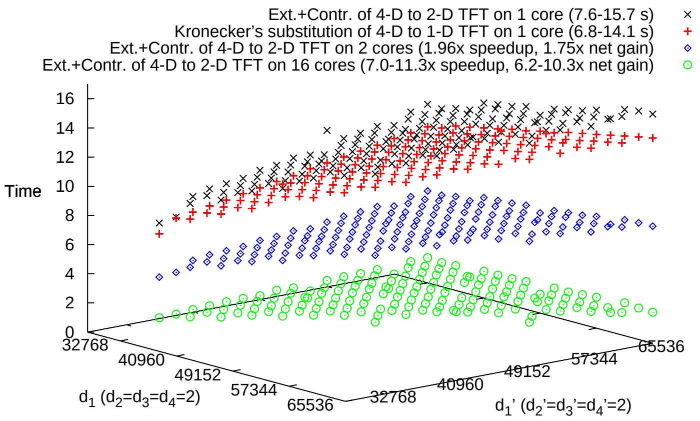

Fig. 6. 4-variate multiplication timing (s) via balanced multiplication on 1, 2, 16 cores vs Kronecker's substitution to 1-D TFT on an Intel Xeon multi-core.

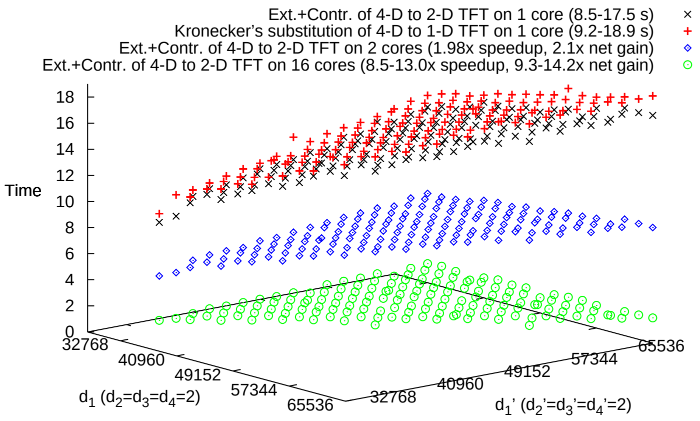

Fig. 7. 4-variate multiplication timing (s) via balanced multiplication on 1, 2, 16 cores vs Kronecker's substitution to 1-D TFT on an AMD Opteron multi-core.

slightly better than ours; indeed our method has a higher algebraic complexity, even though it improves on cache complexity. On 2 cores our method reaches a speedup gain of 1.75 w.r.t Kronecker's substitution and a speedup factor of 1.96 w.r.t. itself on 1 core. On 16 cores, these maxima become 10.3 and 11.3. For comparison with the direct 4-D TFT approach, see Figure 3 in Section 3.

On the Opteron machine, our balanced multiplication on 1 core is 5 to 10% faster than Kronecker's substitution (to 1-D) method. This indicates that our balanced multiplication is more cache friendly than Kronecker's substitution method on this Opteron multi-core. Using 16 cores, our balanced multiplication obtains speedup factors up to 13.5 w.r.t the timing on 1 core. The timing of our balanced multiplication using 16 cores on the Opteron machine is even faster by about 5 to 20% than that of using 16 cores on the Xeon multi-core, even though its CPU is slower (2.2 GHz vs 2.4GHz). This again demonstrates the cache efficiency of our balanced multiplication on multi-cores with multi-level memory hierarchies.

Lemma 1. *With $d_f, d_g, b, q_f, q_g, r_f, r_g$ as above, given a positive integer σ, define $t_1 := (2b-1)$ and $t_2 := (q_f + q_g + 1)\sigma$. There exists at least one integer b such that we have*

$$-1 \leq t_1 - t_2 \leq 2\sigma.$$

In particular, for $\sigma = 1$, the inequality $|s_1 - s_2| \leq 2$ can be achieved. If, in addition, $d_f = d_g$ is satisfied, there exists b such that $|s_1 - s_2| \leq 1$ holds.

Proof. We solve for b (as positive integer) the quadratic equation $2b^2 - \sigma b - (d_f + d_g)\sigma = 0$, which means $s_1 = s_2$, assuming $r_f = r_g = 0$. Its discriminant is

$$\Delta := (\sigma + 1)^2 + 8(d_f + d_g)\sigma.$$

Let k be the positive integer satisfying $k^2 \leq \Delta < (k+1)^2$. We define:

$$b_i := \frac{\sigma + k + i}{4}.$$

For $i = -1, 0, 1, 2$, elementary calculations bring the respective inequalities: $-2 \leq t_1 - t_2 \leq 2\sigma$, $-1 \leq t_1 - t_2 \leq 2\sigma$, $-1 \leq t_1 - t_2 \leq 2\sigma$ and $1 \leq t_1 - t_2 \leq 2\sigma$. For the case where $\sigma = 1$ and $d_f = d_g$, we have $|s_1 - s_2| \leq 1$ for either $b = k'$ or $b = k' + 1$ with ${k'}^2 \leq d_f < (k'+1)^2$. $\square$

7. Concluding Remarks

We have presented strategies for the implementation of dense polynomial multiplication on multi-core architectures. We have focused on polynomial multiplication over finite fields based on FFT techniques since this is a fundamental operation for symbolic computation. The techniques that we have developed for this operation are highly efficient in terms of parallelism and cache complexity. Our results are both theoretical and practical. We are not aware of similar work in the context of symbolic computation.

The design of our techniques has mainly two motivations. First, we aim at supporting higher-level parallel algorithms for solving systems of non-linear equations. Therefore, our multiplication must perform efficiently in terms of sequential running time, parallelism and cache complexity, on any possible input degree patterns, insisting on those which put code efficiency to challenge.

Secondly, we have integrated the specificities of 1-D FFT computations over finite fields in the context of symbolic computation with polynomials. On one hand, these 1-D FFTs are applied to vectors which are large enough such that the base field may not contain the appropriate primitive roots of unity for a radix 2 Cooley Tukey algorithm. On the other hand, the length of these vectors is not large enough for making efficient use of parallel code for 1-D FFT.

As a consequence of these constraints, we have assumed that 1-D FFTs in our implementation could be computed by a black box program, possibly a sequential one. Therefore, we had to take advantage of the row-column algorithm for multi-dimensional FFT computations. Our theoretical analysis has shown that balanced bivariate multiplication, as defined in Section 3.3, is a good kernel.

Based on this observation, we have developed two fundamental techniques, contraction and extension, in order to efficiently reduce any dense multivariate polynomial multiplication to this kernel.

Our experimental results demonstrate that these techniques can substantially improve performances with respect to multiplication based on a direct (and potentially un-balanced) multidimensional FFT. Moreover, they can lead to efficient parallel code for univariate multiplication, despite of our 1-D FFT black box assumption. We believe that symbolic computation software packages such as Maple, Magma, NTL can greatly benefit from our work.

Acknowledgments

This work was supported in part by NSERC and the MITACS NCE of Canada, and NSF under Grants 0540248, 0615215, 0541209, and 0621511. We are very grateful for the help of Professor Charles E. Leiserson, Dr. Matteo Frigo and all other members of SuperTech Group at CSAIL MIT and Cilk Arts. We would also like to thank Professor Zhiyi Huang for fruitful discussions on efficient memory management. In addition, sincere thanks to the reviewers for their constructive comments.

References

[1] E. Becker, T. Mora, M. G. Marinari and C. Traverso. The shape of the shape lemma. *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC) '94*, pages 129–133, 1994.
[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *IEEE Conference on Foundations of Computer Science '94*, 1994.
[3] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. of Comp.*, 19:297–301, 1965.
[4] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.
[5] M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran. Cache-oblivious algorithms. *Annual Symposium on Foundations of Computer Science' 99*, pages 285–297.
[6] M. Frigo, C. E. Leiserson and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN*, 1998.
[7] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge Univ. Press, 1999.

[8] J. Hoeven. The Truncated Fourier Transform and applications. *Proc. of ISSAC'04*, pages 290–296, 2004.
[9] Cilk Arts Inc. Cilk++. http://www.cilk.com/.
[10] Intel Corporation. VTune Performance Analyzer. http://www.intel.com/.
[11] X. Li, M. Moreno Maza, R. Rasheed and É. Schost. The modpn library: Bringing fast polynomial arithmetic into Maple. *Proc. of Milestones of Computer Algebra '08*.
[12] X. Li, M. Moreno Maza and É. Schost. Fast arithmetic for triangular sets: From theory to practice. *Proc. of ISSAC'07*, pages 269–276, 2007.
[13] LinBox Project. http://www.linalg.org/.
[14] MAGMA Project. http://magma.maths.usyd.edu.au/magma/.
[15] P. L. Montgomery. Modular multiplication without trial division. *Math. of Comp.*, 44(170):519–521, 1985.
[16] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. *High Performance Computing Symposium (HPCS) 2009*, D. J. K. Mewhort et al. (Eds.), LNCS 5976, pp. 378–399. Springer-Verlag Berlin, Heidelberg 2010.
[17] NTL. *The Number Theory Library.* http://www.shoup.net/ntl.