Multithreaded Parallel Implementation of Arithmetic Operations Modulo a Triangular Set

Xin Li
ORCCA, UWO
London, Ontario, Canada
xli96@csd.uwo.ca

Marc Moreno Maza
ORCCA, UWO
London, Ontario, Canada
moreno@csd.uwo.ca

ABSTRACT

We discuss the parallelization of arithmetic operations on polynomials modulo a triangular set. We focus on parallel normal form computations since this is a core subroutine in many high-level algorithms, such as triangular decompositions of polynomial systems.

When computing modulo a triangular set, multivariate polynomials are regarded recursively as univariate ones, which leads to several implementation challenges when one targets highly efficient code. We rely on an algorithm proposed in [17] which addresses some of these issues.

We propose a two-level parallel scheme. First, we make use of parallel multidimensional Fast Fourier Transform in order to perform multivariate polynomial multiplication. Secondly, we extract parallelism from the structure of the sequential normal form algorithm of [17]. We have realized a multithreaded implementation. We report on different strategies for the management of tasks and threads.

Categories and Subject Descriptors:
I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation – *Algebraic Algorithms*

General Terms:
Algorithms, Performance, Design, Experimentation, Languages, Theory.

Keywords:
Parallelization, Algorithms, High-performance, Normal form, polynomials.

1. INTRODUCTION

Arithmetic operations on polynomials modulo a triangular set are essential in many areas of symbolic computation. They are the core routines for higher-level procedures, such as computing polynomial GCDs and resultants, solving systems of algebraic and differential equations. In practice, they can impact the performances of a computer algebra software in a dramatic manner. The speed-up factor can be obtained by using fast arithmetic techniques as in [6, 17], by optimizing memory traffic as in demonstrated in [12] or by combining different code levels, as illustrated in [16, 18].

With the increasing availability of parallel architectures, we are interested in the following questions. How to achieve efficient parallel implementations of these polynomial operations? How will they impact the performances of a parallel polynomial system solver. This paper deals with the first question, the second one being the motivation of future work.

When computing modulo a triangular set, multivariate polynomials are regarded recursively as univariate ones. Thus, such polynomials can be represented using tree structures. One can also flatten these trees into a vector, for instance by using Kronecker's substitution. This recursive vision naturally suggests a recursive implementation of arithmetic operations. This is, actually, a challenge when one targets highly efficient code. Indeed, different intermediate coefficient rings are involved, which implies different implementations of ring operations, such as multiplication, and thus different cut-off between classical and faster algorithms. Moreover, asymptotically fast algorithms, such as FFT-based algorithms, are clearly easier to implement and optimize when the actual coefficient ring is a prime field rather than a polynomial ring over a finite field.

In order to overcome this difficulty, we rely on the algorithms introduced in [17]. In broad words, given a triangular set $\{T_1(x_1), T_2(x_1, x_2)\}$ with coefficients in $F_p = \mathbb{Z}/p\mathbb{Z}$, this algorithm allows us to efficiently implement the multiplication in $(F_p[x_1]/\langle T_1 \rangle)[x_2]]/\langle T_2 \rangle$ by avoiding plain division in $F_p[x_1, x_2]$ and relying only on univariate operations in $F_p[x_1]$ and multiplication in $F_p[x_1, x_2]$, for which high-performance sequential algorithms and code are available.

Turning now to the parallelization of these arithmetic operations modulo a triangular set, it is desirable to follow the path open in [17], for the same reasons as in the sequential case. Moreover, multiplication in $F_p[x_1, x_2]$ can be achieved by parallel FFT-based techniques, which is a well studied subject. In Section 2 we discuss two natural parallelizations of the algorithms in [17] and in Section 3 we propose two implementation strategies based on multithreaded parallelism. In Section 4, we report on our experimentations.

The targeted application of this work is the implementation of modular methods in algorithms for computing triangular decompositions such as [5]. Our medium/fine-grained parallel arithmetic modulo a triangular set should combine in a favourable manner with the coarse-grained approach for parallel and modular computations of triangular decompositions presented in [20].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASCO'07, July 27–28, 2007, London, Ontario, Canada.

Copyright 2007 ACM 978-1-59593-741-4/07/0007 ...$5.00.

We know that, in these circumstances, the polynomials that we are computing with are very dense, which makes load balancing easier to achieve than in the sparse case. The experimentation reported in [19] support this claim. Therefore, this density assumption has guided our implementation strategies. We leave for future work the challenging case of sparse polynomials modulo a triangular set. This is a need for computations such as those presented in [7], where the number of variables can be in the order of several hundreds.

Computations in algebraic number fields or in $GF(2^k)$ are very important in symbolic computation and cryptography, and can be performed with triangular sets. Parallel algorithms and implementations of these computations are reported in [13, 9, 2]. In this paper, the base field is F_p and our techniques rely heavily on this assumption. However, by means of modular methods, our work could apply to computations in algebraic number fields.

Other multithreaded symbolic computations are reported in [14, 11], considering polynomial GCDs and resultants. This is another area of interaction for our work, which could again benefit to a parallel solver such as the one of [20].

2. ALGORITHMS

Let $L_0 = \mathbb{K}$ be a commutative ring with a 1. Let B be a univariate polynomial in $\mathbb{K}[x]$, non-constant, monic and with degree $d > 1$. We aim at computing modulo B the product $A \in \mathbb{K}[x]$ of two polynomials reduced w.r.t. B, that is, with degree less than d. So, for simplicity, let us assume that A has degree $2d-2$.

The quotient Q and the remainder R in the division of A by B can be computed as follows, using the trick of Cook-Sieveking-Kung [4, 22, 15]. We summarize this trick and refer to [8] for details. Let B^{-1} be the inverse of the reversal of B modulo x^{d-1}. Let $\overline{Q}$ be the product $\overline{A}B^{-1}$ computed modulo x^{d-1}, where $\overline{A}$ is the reversal of A. Then Q is the reversal of $\overline{Q}$ and we have $R = A - BQ$.

Consider now $\mathcal{T} = (T_1, \ldots, T_s)$ a set of non-constant polynomials in $\mathbb{K}[x_1, \ldots, x_s]$. Let d_i be the degree of T_i w.r.t. x_i, for all i. We say that T is a *triangular set* if for all i, the polynomial T_i lies in $\mathbb{K}[x_1, \ldots, x_i]$, is monic in x_i and is reduced with respect to $T_1, \ldots, T_{i-1}$, that is, for all $j = 1, \ldots, i-1$ the degree of T_i w.r.t. x_j is less than of d_j.

Let $1 \le i \le s$ and let $P \in \mathbb{K}[x_1, \ldots, x_s]$. The *normal form* of P w.r.t. $T_1, \ldots, T_i$, denoted by $\mathrm{NF}_i(P)$, is the the unique polynomial $R \in \mathbb{K}[x_1, \ldots, x_s]$ which is reduced w.r.t. $T_1, \ldots, T_i$, and congruent to P modulo the ideal $\langle T_1, \ldots, T_i \rangle$. Moreover, we define $\mathrm{NF}_0(P) = P$.

For $i = 1, \ldots, s$ we define $L_i = \mathbb{K}[x_1, \ldots, x_i]/\langle T_1, \ldots, T_i \rangle$, the residue class ring of $\mathbb{K}[x_1, \ldots, x_i]$ modulo $\langle T_1, \ldots, T_i \rangle$.

Our main goal is to implement arithmetic operations in all L_i, leading to normal form computations for polynomials in $\mathbb{K}[x_1, \ldots, x_s]$ modulo $\langle T_1, \ldots, T_i \rangle$. We summarize the algorithm of [17]. We assume that, for all $1 \le i \le s$, the inverse T_i^{-1} of the reversal of T_i in $L_{i-1}[x_i]/\langle x_i^{d_i-1} \rangle$ has been precomputed. Let $P \in \mathbb{K}[x_1, \ldots, x_s]$ be such that the degree of P w.r.t. x_i is at most $2d_i - 2$ for all $1 \le i \le s$. Then we compute $\mathrm{NF}_s(P)$ as follows:

Step 1 Let $P' := \mathrm{NF}_{s-1}(P)$.

Step 2 Let $\overline{P'}$ be the reversal of P' in $L_{s-1}[x_s]$. Let $\overline{P'} := \overline{P'} \bmod x_s^{d_s-1}$ and let $\overline{Q} := \overline{P'} T_s^{-1} \bmod x_s^{d_s-1}$.

Step 3 Let $\overline{Q} := \mathrm{NF}_{s-1}(\overline{Q})$.

Step 4 Let Q be the reversal of $\overline{Q}$ in $L_{s-1}[x_s]$. Let $R := P - QT_s$.

Step 5 Return $\mathrm{NF}_{s-1}(R)$.

For a polynomial F in $\mathbb{K}[x_1, \ldots, x_s]$, with positive degree w.r.t. x_s, we compute $\mathrm{NF}_{s-1}(F)$ as a "map" on its coefficients w.r.t. x_s.

We parallelize the computation of $\mathrm{NF}_s(P)$ at two levels. First, for degrees large enough, we perform the products in **Step 2** and **Step 4** by means of a parallel multi-dimensional FFT algorithm, see Section 3.1. From now on, let us regard these products as atomic operations. Secondly, we focus on the calls to the NF_{s-1} function performed at **Step 1**, **Step 3** and **Step 5**. Let G be the *task graph* or *instruction stream DAG* [3] associated with $\mathrm{NF}_s(P)$. One can use either a *depth-first* traversal or a *bottom-up level-by-level* traversal for G, leading to the two parallel schemes detailed in Section 3.3. Note that our task graph G is not a fork-join graph and the special techniques developed for this kind of task graphs, see for instance [23], do not apply here.

In fact, the structure of the algorithm implies several "global synchronisations". More precisely, before starting each of **Step 2**, **Step 3**, **Step 4** and **Step 5**, all threaded computations of the previous step must be completed. These constraints make the parallelization of our normal form computations more challenging than for more standard "divide & conquer" algorithms. See also [21] on this topic.

3. IMPLEMENTATION

3.1 Multidimensional FFT

The standard multidimensional FFT algorithm [1] as shown in the pseudo-code **MultiFFT** below has been implemented in C in our packages. In the standard approach, the input polynomials $f_1, f_2 \in \mathbb{K}[x_1, x_2, \ldots, x_s]$ will be evaluated, for all $i = 1 \cdots s$, by N/N_i Discrete Fourier Transforms (DFTs) on dimension i in turn, where N_i is the FFT size on dimension i, and $N = \prod_{i=1}^{s} N_i$.

MultiFFT:

Input: $f_1, f_2 \in \mathbb{K}[x_1, x_2, \ldots, x_s]$.

Output: $f_1 f_2 \in \mathbb{K}[x_1, x_2, \ldots, x_s]$.

1 **for** i **from** 1 **to** s **do**
2 **Eval**(f_1, i)
3 **Transpose**(f_1, i, 1)
4 **Eval**(f_2, i)
5 **Transpose**(f_2, i, 1)
6 $f_1 f_2$ = **PairwiseMul** (f_1, f_2)
7 **for** i **from** s **to** 1 **do**
8 **Interp**($f_1 f_2$, i)
9 **Transpose**($f_1 f_2$, i, 1)
12 **return** $f_1 f_2$

In the pseudo-code, **Eval** stands for the DFT computations on the dimension i. Before applying **Eval** on dimension i, the coefficient arrays of the input polynomials will be transposed (transposing data on dimension i to dimension 1) for preserving the good locality of memory reference; **Transpose** stands for this operation. Likewise, during the interpolation we also need to transpose the coefficient arrays

backward. And the interpolation, presented by **Interp**, is conducted by the inverse DFTs (IDFTs).

Multidimensional FFT is a very nice application to parallelize on a SMP architecture, since the "small" DFTs/IDFTs performed on a given dimension have no data dependency to each other. Therefore, instead of computing these "small" DFTs/IDFTs one by one in a sequential setting, we create multiple threads and each of the threads will be in charge of an amount of "small" DFTs/IDFTs' computations. Since all the "small" DFTs/IDFTS have similar amount of workloads in a dense polynomial application, each thread will be in charge of a similar number of "small" DFTS/IDFTS.

In fact, when implementing multivariate polynomial multiplication in our sequential mode, we used two approaches. One is the above mentioned multidimensional FFT, the other is based on Kronecker's substitution. In this latter method, two input multivariate polynomials are mapped to univariate ones. Then, univariate FFT can be used to compute the polynomial multiplication. This implies that parallelizing Kronecker's substitution based FFT multiplication is actually, parallelizing a univariate FFT. We didn't try this direction based on three reasons. First, the multidimensional FFT is much easier to parallelize as we described before. Second, we implemented Truncated Fourier transform (TFT) [10], and replaced multidimensional FFT by multidimensional TFT in our package. This brings us a significant improvement of performance comparing to Kronecker's substitution method as reported in Section 4. Moreover, the multidimensional TFT has the same code structure as the multidimensional FFT, thus is easy to implement. Third, multidimensional FFT/TFT is more cache friendly comparing to Kronecker's substitution method for certain range of input [17].

Therefore, on a multi-processor architecture we prefer multidimensional FFT to Kronecker's substitution method. In addition, matrix transposition in multidimensional FFT also can be parallelized. We leave it as a future work, since the computation time of matrix transposition is generally a small portion of the whole computation.

3.2 Two Traversal Methods for Normal Form

By using the names defined in our pseudo-code in this section, we describe the *normal form* operation as follows. The *normal form* operation consists of two major operations **UniFastMod** and **NormalForm**. **NormalForm** is the "main" function which recursively reduces the coefficients of the input polynomial $f \in \mathbb{K}[x_1, x_2, \cdots, x_s]$. TS is brief for the given triangular set, and s is the number of variables.

In addition, we have following definition of operations for all pseudo-code in this section.

- $\mathrm{rev}_n(f)$ returns ${x_s}^n f(\frac{1}{x_s})$, where x_s is the main variable of f and $n \geq deg(f)$.
- $deg(f)$ returns the degree of f.
- $degree(f,\ i)$ returns the partial degree of f in x_i.
- $coef\ (f,\ i)$ fetches the i-th coefficient of f.
- $coef\ (f,\ i,\ s)$

  This operation only applies to a dense multivariate polynomial f who is encoded in an one dimensional array. we define this operation as follows. For the input polynomial f, we use a data representation based on Kronecker substitution [6, 17]. Namely, a dense multivariate polynomial will be encoded in an one dimensional array. The Kronecker map $U(f)$ is an array of element of $\mathbb{K}$.

$$
\begin{array}{ll}
U: & (x_1,\ x_2, \cdots,\ x_s) \longmapsto (x_1,\ {x_1}^{\delta_2}, \cdots,\ {x_1}^{\delta_s}) \\
where & \delta_1 = 1, \\
 & \delta_i = \ \prod_{j=1}^{i-1} (degree(f,\ j)\ +\ 1)
\end{array}
\tag{1}
$$

Thus, $coef\ (f,\ i,\ s)$ returns the i-th slot of $U(f)$ regarded as an array where each slot has size of δ_s.

```
NormalForm (f, TS, s)

Input: f ∈ K[x_1, x_2, ···, x_s], TS = {T_1, T_2, ···, T_s},with
       T_i is monic.

Output: The normal form of f w.r.t. TS.

1 if (s == 0) return f
2 d = deg(f)
3 RC (f, 0, d, TS, s − 1)
4 f =UniFastMod(f, TS, s)
```

```
UniFastMod (f, TS, s)

1  n ⟵ deg f
2  m ⟵ deg T_s
3  if n < m then
4    q ⟵ 0
5    r ⟵ f
6  else
7    q ⟵ rev_n(f) T_s^{-1}   mod  x^{n-m+1}
8    q ⟵ rev_{n-m}(q)
9    RC(q, 0, n − m, TS, s − 1)
10   w = T_s q
11   RC(w, 0, n − m, TS, s − 1)
12   r ⟵ f − w
13 return r
```

Each reduction step is performed by calling **UniFastMod**, namely a fast univariate division in $L_{s-1}[x_s]$. The function **RC** means to reduce each coefficient of a polynomial by calling **NormalForm** and it is an in-place operation.

```
RC (f, start, end, TS, s)

1 for i from start to end do
2   coef (f, i)=NormalForm (coef (f, i), TS, s)
```

As we mentioned above, a multivariate polynomial can be encoded by a tree structure. When reducing its coefficients, we need to have a tree traversal. The nested recursion in **NormalForm** performs a depth-first tree traversal. The other way is what we called "bottom-up level-by-level" (BULL) traversal. The pseudo functions **RS**, **NormalForm2**, **UniFastMod2**, and **RC2** describe the computational steps for this method.

```
NormalForm2 (f, TS, s)

1 if (s == 0) return f
2 size = ∏_{j=1}^{s} (degree(f, j) + 1)
3 i = 2
4 while (i≤s) do
5    ss = size / ∏_{j=1}^{i} (degree(f, j) + 1)
6    RS (f, 0, ss − 1, TS, i)
7    i = i + 1
```

```
RS (f, start, end, TS, s)

1 for i from start to end do
2    coef(f, i, s)=UniFastMod2(coef(f, i, s), TS, s)
```

In brief, we suppose the input multivariate polynomial f is encoded in an one dimensional array by the Kronecker map U. The *size* of the array is $\prod_{j=1}^{s}(degree(f,\ j)\ +\ 1)$. We start the reduction steps at level 1. That is we view the given array as an array with $size\ /\ (degree(f,\ 1)\ +\ 1)$ slots. Each slot has size of $degree(f,\ 1)\ +\ 1$. Each slot actually is encoding an univariate polynomial in L_1. Then we reduce all slots by calling **UniFastMod2**. Then we continue the reduction steps on level 2, 3, $\cdots$, i, $\cdots$, s.

On level i, the given array is viewed as an array with $size\ /\ \prod_{j=1}^{i}(degree(f,\ j)\ +\ 1)$ slots. Each slot has size $\prod_{j=1}^{i}(degree(f,\ j)\ +\ 1)$. We iteratively conduct the reduction steps from level 1 to level s by calling function **RS**. In this way, we compute a *normal form* in a BULL traversal.

```
UniFastMod2 (f, TS, s)

1  n ⟵ deg f
2  m ⟵ deg T_s
3  if n < m then
4    q ⟵ 0
5    r ⟵ f
6  else
7    q ⟵ rev_n(f) T_s^{-1}  mod  x^{n-m+1}
8    q ⟵ rev_{n-m}(q)
9    RC2(q, 0, n − m, TS, s − 1)
10   w = T_s q
11   RC2(w, 0, n − m, TS, s − 1)
12   r ⟵ f − w
13 return r
```

```
RC2 (f, start, end, TS, s)

1 for i from start to end do
2    coef(f, i)=NormalForm2(coef(f, i), TS, s)
```

3.3 Parallelizing Normal Form

Both approaches based on either depth-first or bottom-up level-by-level tree traversal are nice applications to parallelize. In our setting, we suppose input polynomial are dense, thus the workload of each coefficient reduction is close. We describe our parallelization strategies as follows.

3.3.1 Parallelism in Depth-first Method

```
NormalForm_Para (f, TS, s)

1 if (s == 0) return
2 d = deg(f)
3 for i from 0 to d do
4    Task=NormalForm_Para(coef(f, i),TS,s-1)
5    CreateThread (Task)
6 DumpThreadPool()
7 f =UniFastMod(f, TS, s − 1)
```

```
CreateThread (Task)

1 Creat a thread for Task in thread_pool
2 if thread_pool is full.
3    DumpThreadPool(thread_pool)
```

```
DumpThreadPool(thread_pool)

1 Force all threads in thread_pool to finish.
```

In the depth-first method we cursively create a thread for each coefficient reduction which we called a "**task**". All threads will live in a **thread_pool**. When the **thread_pool** is full. We will force all threads to finish up before inserting a new one. To force all threads to finish, we use the function **DumpThreadPool**. Function **NormalForm_Para** in above pseudo-code is the parallelized version of the depth-first multivariate reduction.

3.3.2 Parallelism in Bottom-up Level-by-level Method

```
NormalForm2_Para_1 (f, TS, s)

1  if (s == 0) return f
2  size = ∏_{j=1}^{s} (degree(f, j) + 1)
3  i = 2
4  while (i≤s) do
5    ss = size / ∏_{j=1}^{i} (degree(f, j) + 1)
     // suppose NoOfCPU divides ss.
6    q = ss / NoOfCPU
7    for j from 0 to NoOfCPU-1 repeat
8       Task = RS (f, jq, (j + 1)q, TS, i)
9       CreateThread ( Task )
10   i = i + 1
11   DumpThreadPool()
```

```
NormalForm2_Para_2 (f, TS, s)

1  Create C threads and put them into sleep.
2  if (s == 0) return f
3  size = ∏_{j=1}^{s} (degree(f, j) + 1)
4  i = 2
5  while (i≤s) do
6    ss = size / ∏_{j=1}^{i} (degree(f, j) + 1)
     // suppose NoOfCPU divides ss.
7    q = ss / NoOfCPU
8    for j from 0 to NoOfCPU-1 repeat
9       Task = RS (f, jq, (j + 1)q, TS, i)
10      Wake up a thread to handle Task.
11   i = i + 1
12 Finish and terminate all threads.
```

In the BULL traversal, we have two slightly different sub-methods. One is that at each level we create C threads to handle all reductions on this level in parallel, where C is a constant. Then, we wait them to finish and destroy these C threads before go to next level. Therefore, the total number of threads being created is parametrized by the number of variables of the input. This sub-method is presented by function **NormalForm2_Para_1** in above pseudo-code. In the other sub-method, we will create a fixed number of threads and put them into sleep at the beginning. Then we start the BULL traversal. When there is a reduction on demanding, we will push it onto a task queue and send a signal to wake up some thread. The waken thread will go to fetch a task from the task queue and handle it immediately. If there are multiple tasks have been pushed on the task queue, multiple threads will be waken up and run in parallel. After finishing a task, the thread will go back to sleep or continue to handle another task. This sub-method is presented by function **NormalForm2_Para_2**.

The first sub-method is very easy to implement. But the overhead of creating and destroying many threads maybe burdensome in large input cases. The second sub-method takes a little more coding effort for tasks management and threads synchronization. But it is advantageous by avoiding the potential overhead happened in the first sub-method.

We used `pthread` library to implement the parallelization. We tested the performance on a AMD 4 processor machine. We observed a factor of 3.5 speed-up when the input size is sufficiently large. The experimentation results are reported in Section 4.

4. BENCHMARKS

In Section 3, several parallelization strategies have been described. We provide benchmark results for these methods. The tested operation is the multivariate polynomial multiplication modulo a triangular set and the tested strategies are summarized in below list.

0	Sequential algorithm.
1	Depth-first traversal with a thread pool.
2	BULL traversal with a thread pool.
3	BULL traversal with sleep/wake-up threads.

We conducted our benchmark on a AMD Opteron 850 4-Processor machine with CPU MHZ 2391.537 and cache size 1024 KB for each processor. The input dense polynomials are randomly generated. The benchmark data can well reflect the performance in real world computation.

We benchmarked 2, 3 and 4 variable cases. We observe a factor of $2 \sim 3$ speed-up in those examples. Here, we only report the data we collected from the 4 variable example. In this example, we fixed the partial degrees in x_3 and x_4 at 4 – the number of processors. Then by increasing partial degrees in x_1 and x_2, we obtain a timing surface for each methods listed in above table. Namely, Figure 1 is the benchmark between the sequential method and the Depth-first traversal parallelization method with a global thread pool. Figure 2 is the benchmark between sequential method and the BULL traversal parallelization method with a global thread pool. Figure 3 is the benchmark between sequential method and the BULL traversal parallelization method with threads sleep/wake-up strategy. And Table 1, 2, 3 and 4 are the selected data point from Figure 1, 2, 3 and 4 respectively.

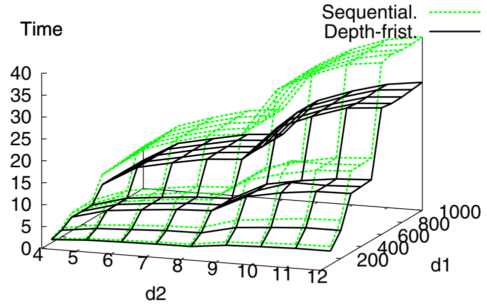

Figure 1: method 0 vs. method 1

Table 1: Selected data points from Figure 1

d_4	d_3	d_2	d_1	method 0 (sec)	method 1 (sec)
4	4	4	100	0.926028	0.736449
4	4	6	300	8.104279	6.015184
4	4	8	500	9.642438	7.084307
4	4	10	800	35.232581	25.746897
4	4	12	1000	39.521405	29.216119

According to the benchmark result, the depth-first method does not improve the performance by big factors w.r.t to the number of processors. The main reason is that when the coarser grain parallelization is well balanced and processors have been well utilized, it's insensible to keep generating finer grain sub-threads recursively for the sub-tasks, especially when the sub-tasks are small. On the other hand, the bottom-up level-by-level approach has a factor of $2 \sim 3$ speed up based on the input size accordingly. The examples with larger degrees have better speed-up than the smaller ones. The main reason for this is that the overhead generated by threads and tasks management is still not negligible for smaller input.

For the comparison between methods 2 and 3, we observe that method 2 outperforms method 3 for smaller input. The main reason is that in methods 3 the overhead of managing task queues and synchronizing signals is more expensive than the one in method 2. When the input is small, the overhead has bigger impact on the overall computational time. Whereas, method 3 will only generate fixed number of threads. Thus, the scheduling becomes much simpler. The overhead of creating /destroying threads in the middle steps has been avoided as well. Thus, for larger input method 3 outperforms method 2 according to our results, though the gap is not big.

Figure 4 shows an improved version of method 3. The speed-up is yielded by replacing all Fast Fourier Transform by Truncated Fourier Transform (TFT). Although this improvement seems unrelated to parallelism, the better multiple cache behavior deserves to be counted in. Namely, TFT requires less memory to store the intermediate results than FFT. There are larger chances that these results will be kept in cache and used in later computation steps on the same processor.

Above benchmarks only show a factor of $2 \sim 3$ speed up

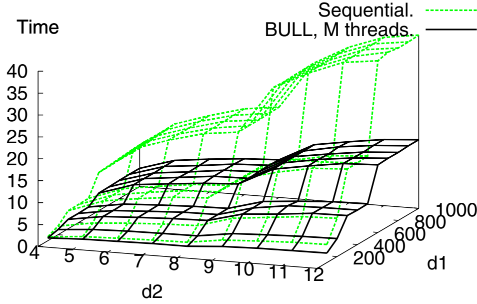

Figure 2: method 0 vs. method 2

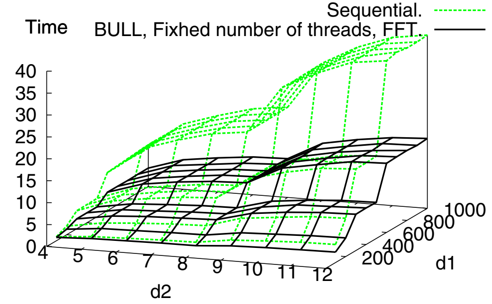

Figure 3: method 0 vs. method 3

Table 2: Selected data points from Figure 2

d_4	d_3	d_2	d_1	method 0 (sec)	method 2 (sec)
4	4	4	100	0.926028	0.659218
4	4	6	300	8.104279	3.844373
4	4	8	500	9.642438	4.391355
4	4	10	800	35.232581	13.915399
4	4	12	1000	39.521405	15.650396

Table 3: Selected data points from Figure 3

d_4	d_3	d_2	d_1	method 0 (sec)	method 3 (sec)
4	4	4	100	0.926028	0.778774
4	4	6	300	8.104279	4.031646
4	4	8	500	9.642438	4.531477
4	4	10	800	35.232581	13.335127
4	4	12	1000	39.521405	14.952662

on a 4 processor machine. This is not a satisfying result with considering that polynomials in our applications are dense ones. Dense polynomial computations usually provide a good opportunity for work-load balance. However, we have identified the major bottle-neck that impedes the perform in our benchmark examples. Recall that in previous benchmarks we set the partial degrees of x_4 and x_3 as a constant number 4. This leads a situation that in some of the sub-algorithms such as *Coefficient Reduction*, there is no enough work-load to be scheduled evenly to all 4 processors by our current scheduling method. Therefore, we increase the degrees of x_3 and x_4 to be 8. Then, we observe a factor $3.2 \sim 3.3$ speed-up between method 1 and method 3. In Table 5 we list a few timing points from the new benchmark result.

When we increase the partial degrees of x_3 and x_4 to be 16, 24, 32, $\cdots$. we ab serve a factor of $3.4 \sim 3.6$ speed-up between method 1 and method 3 (see Table 6).

To summarize, for the larger examples, especially when we increase the partial degrees of x_3 and x_4 in 4-variable case, the performance is reasonably better. By profiling information, we know the top level division in BULL method is often a dominant factor. Thus, increasing the degrees of top level variables to some extend with respect to the number of processors allows a more balanced work-load assignment thus a better performance.

Although, our experiments are conducted on a 4 processor machine. We believes that our approach will scale on larger parallel SMP system. Actually, the number of threads in application has been parametrized such that it can be easily adjusted according to the number of processors or other cut-offs.

5. CONCLUSION

In conclusion, we studied multithreaded versions of multivariate polynomial arithmetic modulo a triangular set. In this report, we focused on the normal form operation. We obtain parallelism from two procedures: a multidimensional FFT algorithm and our normal form algorithm. Due to the intrinsic data-dependency inside these operations, we observe a factor of 2∼3 speed up on a 4 processor machine.

One major issue remains: detecting cut-offs between the different possible strategies. This is a highly complicated task. A cut-off in our application is parameterized by the type of architectures, the number of processors, the number of variables of the input, and the shape of the given triangular set, etc. This motivates us to design an automatic cut-off detection framework for our application in near future. Now we are working on automatic cut-off detecting. We are designing algorithms which will detect the cutoffs between different parallel methods at compile time.

6. REFERENCES

[1] R. Al Na'Mneh, W.D. Pan, and R. Adhami Communication efficient adaptive matrix transpose algorithm for FFT on symmetric multiprocessors. In *Proc. SSST'05 the Thirty-Seventh Southeastern Symposium on System Theor*, pages 312–315. IEEE Computer Society, 2005.

[2] J.-C. Bajard and G. A. Jullien. Parallel Montgomery multiplication in $GF(2^k)$ using trinomial residue arithmetic. In *Proc. ARITH'05, the 17th IEEE Symposium on Computer Arithmetic*, pages 164–171. IEEE Computer Society, 2005.

[3] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc.*

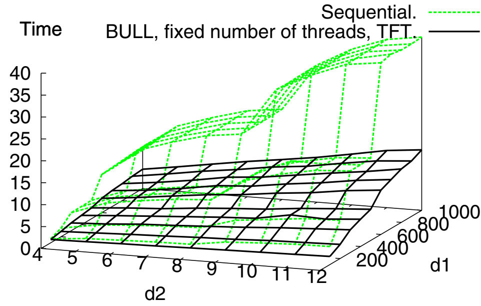

Figure 4: method 0 vs. method 3 with TFT implementation.

Table 4: Selected data points from Figure 4

d_4	d_3	d_2	d_1	method 0 (sec)	TFT (sec)
4	4	4	100	0.926028	0.755583
4	4	6	300	8.104279	2.732532
4	4	8	500	9.642438	4.831472
4	4	10	800	35.232581	10.011660
4	4	12	1000	39.521405	13.816763

Table 5: Larger benchmark 1.

d_4	d_3	d_2	d_1	method 0 (sec)	method 3 (sec)
8	8	8	100	13.770629	4.321261
8	8	8	300	96.117776	18.458235
8	8	8	1000	132.304345	39.757645
8	8	8	1600	277.367573	82.651414

Table 6: Larger benchmark 2.

d_4	d_3	d_2	d_1	method 0 (sec)	method 3 (sec)
16	16	16	16	15.303748	4.567856
16	16	24	24	56.612566	16.479111
16	16	32	32	63.762428	18.359758
16	16	40	40	236.199680	67.175220
16	16	48	48	252.753472	71.237213
16	16	56	56	265.966837	74.979127

SPAA'96, the eighth annual ACM symposium on Parallel algorithms and architectures, pages 297–308, Padua, Italy, ACM Press, 1996.

[4] S. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966.

[5] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *Proc. ISSAC'05*, pages 108–115. ACM Press, 2005.

[6] 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*, pages 93–100, ACM Press, 2006.

[7] M.V. Foursov and M. Moreno Maza. On computer-assisted classification of coupled integrable equations. *J. Symb. Comp.*, 33:647–660, 2002.

[8] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[9] A. Halbutogullari and C. K. Koc. Parallel multiplication in $GF(2^k)$ using polynomial residue arithmetic. *Designs, Codes and Cryptography*, 20(2):155–173, 2000.

[10] J. van der Hoeven. The truncated Fourier transform and applications. In J. Gutierrez, editor, *Proc. ISSAC 2004*, pages 290–296, ACM Press, 2004.

[11] H. Hong and H. W. Loidl. Parallel computation of modular multivariate polynomial resultants on a shared memory machine. In B. Buchberger and J. Volkert, editors, *Proc. of CONPAR 94, Springer LNCS 854.*, pages 325–336. Springer Verlag, 1994.

[12] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical Taylor shift by 1. In *ISSAC'05*, pages 200–207. ACM Press, 2005.

[13] Hyun-Sung Kim, Hee-Joo Park, and Sung-Ho Hwang. Parallel Modular Multiplication Algorithm in Residue Number System. Chapter in *Parallel Processing and Applied Mathematics*, volume 3019/2004 of *Lecture Notes in Computer Science*, Springer, 2004.

[14] W. Küchlin. On the multi-threaded computation of integral polynomial greatest common divisors. In *Proc. of ISSAC '91*, pages 333–342. ACM Press, 1991.

[15] H. T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22:341–348, 1974.

[16] 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, pages 12–23. Springer, 2006.

[17] X. Li, M. Moreno Maza, and É Schost. Fast arithmetic for triangular sets: From theory to practice. In *Proc. ISSAC'07*, ACM Press, 2007.

[18] X. Li, M. Moreno Maza, and É. Schost. On the virtues of generic programming for symbolic computation. In *Proc. CASA'07*, Lecture Notes in Computer Science vol. 4488, pages. 251258, Springer-Verlag Berlin Heidelberg 2007.

[19] M. Moreno Maza, B. Stephenson, S. M. Watt, and Y. Xie. Multiprocessed parallelism support in Aldor on SMPs and multicores. In *Proc. PASCO'07*, ACM Press, 2007.

[20] M. Moreno Maza and Y. Xie. Component-level parallelization of triangular decompositions. In *Proc. PASCO'07*, ACM Press, 2007.

[21] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu and M. Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs In *Proc. PLDI'07*, ACM Press, 2007.

[22] M. Sieveking. An algorithm for division of power series. *Computing*, 10:153–156, 1972.

[23] S. Varette and J.-L. Roch. Probabilistic Certification of Divide & Conquer Algorithms on Global Computing Platforms. Application to Fault-Tolerant Exact Matrix-Vector Product In *Proc. PASCO'07*, ACM Press, 2007.