FFT-based Dense Polynomial Arithmetic on Multi-cores

Marc Moreno Maza[1] and Yuzhen Xie[2]

[1] Ontario Research Centre for Computer Algebra
University of Western Ontario, London, Canada
moreno@csd.uwo.ca
[2] Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, USA
yxie@csail.mit.edu

Abstract. We report efficient implementation techniques for FFT-based dense multivariate polynomial arithmetic over finite fields, targeting multi-cores. We have extended a preliminary study dedicated to polynomial multiplication and obtained a complete set of efficient parallel routines in Cilk++ for polynomial arithmetic such as normal form computation. Since bivariate multiplication applied to balanced data is a good kernel for these routines, we provide an in-depth study on the performance and the cut-off criteria of our different implementations for this operation. We also show that, not only optimized parallel multiplication can improve the performance of higher-level algorithms such as normal form computation but also this composition is necessary for parallel normal form computation to reach peak performance on a variety of problems that we have tested.

Keywords: parallel polynomial arithmetic, parallel polynomial multiplication, parallel normal form, parallel multi-dimensional FFT/TFT, Cilk++, multi-core

1 Introduction

Polynomial Arithmetic is at the core of every computer algebra system (CAS) such as AXIOM, MAGMA, MAPLE, MATHEMATICA, NTL and REDUCE, and has an essential impact on the performance of these software packages. Today, the ubiquity of hardware acceleration technologies (multi-cores, graphics processing units, ...) makes the development of *basic polynomial algebra subroutines (BPAS)* necessary in order to support CAS, akin to the BLAS in numerical linear algebra.

The work presented in this paper aims at contributing to this effort. In fact, and up to our knowledge, this is the first report on the parallelization of dense polynomial arithmetic, over finite fields and targeting multi-cores. All symbolic calculations on univariate and multivariate polynomials can be reduced to computing with polynomials over finite fields (such as the prime field Z/pZ for a prime number p) via the so-called *modular techniques*. Moreover, most symbolic calculations tend to densify intermediate expressions even when the input and output polynomials are sparse. See Chapter 5 in [10] for an extensive presentation of these ideas, which explain why we focus primarily on dense polynomials over finite fields.

Such polynomials are well suited for the use of asymptotically fast algorithms based on FFT techniques. Note that some features of FFT techniques are specific to finite fields, see Section 2.1 for details. In this context polynomial multiplication plays a central role, and many basic operations on polynomials such as division can be efficiently reduced to multiplication. This observation has motivated our preliminary study [19] dedicated to FFT-based dense polynomial multiplication. We have shown that *balanced input data* can maximize parallel speedup and minimize cache complexity for bivariate multiplication. We say that a pair of multivariate polynomials is *balanced* if the partial degrees of their product are equal (or very close). However, unbalanced input data, which are common in symbolic computation, are challenging. We have provided efficient techniques to reduce multivariate (and univariate) multiplication to *balanced bivariate multiplication*. Our implementation in `Cilk++` [3] demonstrates good speedup on multi-cores. Sections 2.2 to 2.4 summarize the context and the results of [19].

In order to obtain a solid foundation library for basic polynomial algebra subroutines over finite fields and targeting multi-cores, at least two essential problems need to be handled and are addressed in this paper. In Sections 1.1 and 1.2, we describe these problems and present our solutions, which are detailed through Sections 3 to 6. Section 1.3 describes our experimentation framework. Finally, in Section 7 we summarize our results and discuss the outcome of this research.

1.1 Optimizing Balanced Bivariate Multiplication

FFT-based bivariate multiplication can be achieved by a variety of algorithms and implementation techniques. Since balanced bivariate multiplication is the kernel to which we are reducing multivariate multiplication, we need to determine the most appropriate algorithm and implementation techniques for the input patterns of practical interest. This problem of *cut-off criteria* is essential in scientific computing. For instance, in coding matrix multiplication, one is faced with choosing among Strassen multiplication, classical multiplication and others; see [12] for details.

Determining cut-off criteria is even more challenging in the context of multi-core programming where both input data patterns and number of cores need to be taken into account. This makes it necessary to combine theoretical and empirical analysis. The former cannot provide precise criteria due to simplification hypotheses but helps narrowing the pattern ranges of the latter. Once the empirical results are obtained, the theoretical analysis can also help understanding them.

In this work, we consider two implementations of bivariate multiplication. One is based on Cooley-Tukey FFT; often we simply call it FFT. The other is based on Truncated Fourier Transform (TFT), see Sections 2.3. The theoretical analysis provides a simple cut-off criterion between our two algorithms when run serially. Section 4, after a description of our implementation, provides experimental results on 1, 8, 12 and 16 cores on a 16-core machine. We obtain simple cut-off criteria in each case and for several degree patterns of practical interest. These results are important since for certain degree ranges and for certain numbers of cores TFT substantially outperforms FFT while FFT is faster in the other cases. Taking advantage of these features can speedup not only multiplication but also the operations that rely on it.

1.2 Efficient Parallel Computation of Normal Forms

All basic operations on polynomials, such as division, can be reduced to multiplication. For multivariate polynomials over finite fields, two basic operations are of high interest: *exact division* and *normal form computations*. Note that polynomial addition is important too but does not bring any particular implementation challenges.

Given two multivariate polynomials f and g, we say that g *divides* f exactly if there exists a polynomial q such that $f = qg$ holds. Testing whether g divides f exactly and computing q when this holds can be done using FFT-techniques similar to those used for multiplication in Section 2.3. Hence we do not insist on this operation since no major additional implementation issues have to be handled with respect to multiplication.

However, computing normal forms (as defined in Section 2.5) brings new challenges. Indeed, complexity estimates show that the serial and parallel times of these computations (using the multi-threaded programming model of [9]) are exponential in the number of variables, see Section 5. Moreover, the number of synchronization points in a parallel program computing normal forms is also exponential in the number of variables. In addition, at each synchronization point the number of threads which need to join grows with the input data size (precisely with their partial degrees). Consequently, the parallel overhead is potentially large. A first attempt for parallelizing the serial normal form algorithms of [17] reached limited success as reported in [14].

In this work, we investigate how our parallel multiplication code could be efficiently composed with a parallel normal form implementation. This has the potential to increase parallel speedup factors but also parallel overhead.

Once again we approach this problem by combining theoretical and empirical analysis. In the former, some parallel overhead (the one for parallelizing a `for` loop) is taken into account, but not all. For instance, those coming from synchronization points are neglected. In Section 5, this theoretical analysis suggests that parallel multiplication can improve the parallelism of parallel normal form computation. In Section 6, our experimentation not only confirms this insight but shows that parallel multiplication is necessary for parallel normal form computation to reach good speedup factors on all input patterns that we have tested. These results are important since normal form computations represent the dominant cost in many higher-level algorithms, such as those for solving systems of polynomial equations, which is our driving application.

1.3 Experimentation Framework

The techniques proposed in this paper are implemented in the Cilk++ language [3], which extends C++ to the realm of multi-core programming based on the multi-threaded model realized in [9]. The Cilk++ language is also equipped with a provably efficient parallel scheduler by work-stealing [2]. We use the serial C routines for 1-D FFT and 1-D TFT from the `modpn` library [16]. Our integer arithmetic modulo a prime number relies also on the efficient functions from `modpn`, in particular the improved Montgomery trick [18], presented in [17]. All our benchmarks are carried out on a 16-core machine with 16 GB memory and 4096 KB L2 cache. All the processors are Intel Xeon E7340 @ 2.40GHz.

2 Background

Throughout this paper $\mathbb{K}$ designates the finite field Z/pZ with p elements, where $p > 2$ is a prime number. All polynomials considered hereafter are multivariate with coefficients in $\mathbb{K}$ and with n ordered variables $x_1 < \cdots < x_n$. The set of all such polynomials is denoted by $\mathbb{K}[x_1, \ldots, x_n]$.

The purpose of this section is to describe algorithms for two basic operations in $\mathbb{K}[x_1, \ldots, x_n]$: *multiplication* and *normal form computation*. These algorithms are based on FFT techniques. We start by stressing the specificities of performing FFTs over finite fields, in particular the use of the *Truncated Fourier Transform* (TFT). In the context of polynomial system solving, which is our driving application, this leads to what we call the *1-D FFT black box assumption*.

2.1 FFTs over Finite Fields and the Truncated Fourier Transform

Using the Cooley-Tukey algorithm [6] (and its extensions such as Bluestein's algorithm) 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 $\mathbb{K}$, three 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 [10, Chapter 14] for details. As a consequence the FFT of a vector of size s over the finite field $\mathbb{K}$ 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) [11]. 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 [17]. 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 (which is needed for running the Cooley-Tukey algorithm on a vector of size s) exists in $\mathbb{K}$ if and only if s divides $p - 1$. Consider two univariate polynomials f, g over $\mathbb{K}$ and let d be the degree of the product fg. It follows that fg can be computed by evaluation and interpolation based on the radix 2 Cooley-Tukey Algorithm (see the algorithm of Section 2.3 with $n = 1$) if and only if some power of 2 greater than d divides $p - 1$. When this holds, computing fg amounts to:

- $\frac{9}{2} \lg(s) s + 3s$ operations in $\mathbb{K}$ using the Cooley-Tukey Algorithm,
- $\frac{9}{2}(\lg(s) + 1)(d + 1) + 3s$ 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 [10, Chapter 8], which introduces "virtual primitive roots of unity". However, this increases the running time to $O(s \lg(s) \lg(\lg(s)))$ scalar operations.

Last but not least, when solving systems of algebraic equations (which is our driving application), partial degrees of multivariate polynomials (including degrees of univariate polynomials) rarely go beyond the million. This implies that, in our context, the lengths of the vectors to which 1-D FFT need to be applied are generally not large enough for making efficient use of parallel code for 1-D FFT.

2.2 The 1-D FFT Black Box Assumption

The discussion of the previous section, in particular its last paragraph, suggests the following hypothesis. We assume throughout this paper that we have at our disposal a **black box** computing the DFT at a 2^ℓ-primitive 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. Therefore, we rely on the row-column multi-dimensional FFT to create concurrent execution in the algorithm presented in Section 2.3.

2.3 Multivariate Multiplication

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

***Step 1*:** Evaluate f and g at each point of the n-dimensional grid $((\omega_1^{e_1}, \ldots, \omega_n^{e_n}), 0 \leq e_1 < s_1, \ldots, 0 \leq e_n < s_n)$ 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.

The above procedure amounts to:

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

operations in $\mathbb{K}$, where $s = s_1 \cdots s_n$. If our 1-D FFT black box relies on TFT rather than the Cooley-Tukey algorithm, the above estimate becomes:

$$\frac{9}{2} \sum_{i=1}^{n} (\prod_{j \neq i} s_j)(d_i + d'_i + 1)(\lg(s_i) + 1) \quad + \quad (n+1) \prod_{i=1}^{n} (d_i + d'_i + 1). \tag{2}$$

2.4 Balanced Bivariate Multiplication

In [19], the authors give a cache complexity estimate of the algorithm of Section 2.3 under the assumption of 1-D FFT black box. Using the theoretical model introduced in [8], and denoting by L the size of a cache line, they have obtained the following

upper bound $c\,s\frac{n+1}{L} + c\,s(\frac{1}{s_1} + \cdots + \frac{1}{s_n})$ for some constant $c > 0$ on the number of cache misses. This suggests the following definition. The pair of polynomials f, g is said *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 $2 \leq i \leq n$. Indeed, for fixed s and n, this bound is minimized when the pair f, g is balanced; moreover it reaches a local minimum at $n = 2$ and $s_1 = s_2 = \sqrt{s}$. Experimentation reported in [19] confirms the good performance of *balanced bivariate multiplication*, that is, bivariate multiplication with balanced input. Based on these results, the authors have developed techniques in order to efficiently reduce any dense multivariate polynomial multiplication to balanced bivariate multiplication.

2.5 Normal Form Computation

Let $f, g_1, \ldots, g_n \in \mathbb{K}[x_1, \ldots, x_n]$ be polynomials. Recall that variables are ordered as $x_1 < \cdots < x_n$. We assume that the set $\{g_1, \ldots, g_n\}$ satisfies the following properties:

(i) for all $1 \leq i \leq n$ the polynomial g_i is non-constant and its largest variable is x_i,
(ii) for all $1 \leq i \leq n$ the leading coefficient of g_i w.r.t. x_i is 1,
(iii) for all $2 \leq i \leq n$ and all $1 \leq j < i$ the degree of g_i in x_j is less than the degree of g_j in x_j, that is, $\deg(g_i, x_j) < \deg(g_j, x_j)$.

Such a set is called a *reduced monic triangular set*. The adjectives triangular, monic and reduced describe respectively the above properties (i), (ii) and (iii). We will denote by δ_i the degree of g_i in x_i and by δ the product $\delta_1 \cdots \delta_n$. For instance, with $n = 2$, $g_1 = x_1^2 + 1$ and $g_2 = x_2^3 + x_1$, the set $\{g_1, g_2\}$ is a reduced monic triangular set. Observe that this notion is dependent on the variable ordering. In our example, the set $\{g_1, g_2\}$ would no longer be triangular for the ordering $x_2 < x_1$.

Reduced monic triangular sets are special cases of Gröbner bases [7] and enjoy many algorithmic important properties. We are interested here in the following one. There exists a **unique** polynomial $r \in \mathbb{K}[x_1, \ldots, x_n]$ such that the following hold:

(iv) either $r = 0$ holds or for all $1 \leq i \leq n$ the degree of r in x_i is less than δ_i,
(v) f is congruent to r modulo $\{g_1, \ldots, g_n\}$, that is, there exist polynomials $q_1, \ldots, q_n \in \mathbb{K}[x_1, \ldots, x_n]$ such that we have $f = r + q_1 g_1 + \cdots + q_n g_n$.

Such a polynomial r is called the *normal form* of f w.r.t. $\{g_1, \ldots, g_n\}$. Consider n, g_1, g_2 as above and $f = x_2^3 x_1 + x_2 x_1^2$. Then $r = -x_2 + 1$ is the normal form of f w.r.t. $\{g_1, g_2\}$. Indeed we have $f = r + q_1 g_1 + q_2 g_2$ with $q_2 = x_1$ and $q_1 = x_2 - 1$; moreover we have $\deg(r, x_1) < \delta_1$ and $\deg(r, x_2) < \delta_2$.

In broad terms the polynomial r is obtained after simplifying f w.r.t. $\{g_1, \ldots, g_n\}$. It is, indeed, what the command `simplify` computes in computer algebra systems such as MAPLE, when this command is applied to f and $\{g_1, \ldots, g_n\}$. This is an essential operation in symbolic computation and the above result states that r is uniquely defined as long as it satisfies (iv) and (v). One natural way for computing r is as follows:

(a) Initialize r_{n+1} to be f.
(b) For i successively equal to $n, n-1, \ldots, 2, 1$ compute r_i as the remainder in the Euclidean division of r_{i+1} by g_i, regarding these polynomials as univariate in x_i.

(c) Return r_1.

In our example we set $r_3 = f$ and compute r_2 the remainder of r_3 by g_2 which is $r_2 = x_2x_1^2 - x_1^2$. Then we compute r_1 the remainder of r_2 by g_1 which is $-x_2 + 1$.

This procedure suffers from intermediate expression swell. (This cannot be seen on very simple example, of course.) One can check that the degree in x_1 of the successive remainders $r_n, \ldots, r_2$ may dramatically increase until r_2 is finally divided by g_1. Consider $g_n = x_n^2 - x_1^2 - 1, g_{n-1} = x_{n-1}^2 - x_1^2 - 1, \ldots, g_2 = x_2^2 - x_1^2 - 1, g_1 = x_1^2 + 1$ and $f = x_2^4 \cdots x_{n-1}^4 x_n^4$. We will obtain $r_2 = (x_1^2 + 1)^{2n-2}$ whereas r is simply 0.

This phenomenon is better controlled by an algorithm proposed by Li, Moreno Maza and Schost in [17]. This latter procedure relies on Cook-Sieveking-Kung's "fast division trick" [5, 20, 13] which reduces an Euclidean division to two multiplications. The algorithm of [17] proceeds by induction on the number of variables and its pseudo-code is shown below. The base case, that is $n = 1$, is given by the procedure $\mathsf{NormalForm}_1$ which is in fact Cook-Sieveking-Kung's fast division. The general case is given by the $\mathsf{NormalForm}_i$ procedure for $2 \leq i \leq n$ where the input polynomial f is assumed to be in $\mathbb{K}[x_1, \ldots, x_i]$ and $\{g_1, \ldots, g_i\}$ is a reduced monic triangular set in $\mathbb{K}[x_1, \ldots, x_i]$. A few comments are needed about these two procedures.

- $\mathrm{Rev}(g_i)$ designates the *reversal* of the polynomial g_i. that is, the polynomial obtained from g_i by reversing the order of its coefficients; for instance $\mathrm{Rev}(g_1) = 3x_1^2 + 2x_1 + 1$ for $g_1 = x_1^2 + 2x_1 + 3$.
- $\mathrm{Rev}(g_i)^{-1} \mod g_1, \ldots, g_{i-1}, x_i^{\deg(f,x_i)-\deg(g_i,x_i)+1}$ is the inverse of $\mathrm{Rev}(g_i)$ modulo the reduced monic triangular set $\{g_1, \ldots, g_{i-1}, x_i^{\deg(f,x_i)-\deg(g_i,x_i)+1}\}$; this can be computed via *symbolic Newton Iteration*, see [10, Chapter 8].
- In practice the quantities $S_1, \ldots, S_n$ are pre-computed and stored before calling $\mathsf{NormalForm}_i$. Therefore, the computations of these quantities are not taken into account in any complexity analysis of these procedures.
- $\mathrm{map}(\mathsf{NormalForm}_{i-1}, \mathrm{Coeffs}(f, x_i), \{g_1, \ldots, g_{i-1}\})$ is the polynomial in x_i obtained from f by replacing each coefficient of f in x_i with its normal form w.r.t. $\{g_1, \ldots, g_{i-1}\}$.

$\mathsf{NormalForm}_1(f, \{g_1\})$
1 $S_1 := \mathrm{Rev}(g_1)^{-1} \mod x_1^{\deg(f,x_1)-\deg(g_1,x_1)+1}$
2 $D := \mathrm{Rev}(f)S_1 \mod x_1^{\deg(f,x_1)-\deg(g_1,x_1)+1}$
3 $D := g_1 \mathrm{Rev}(D)$
4 **return** $f - D$

$\mathsf{NormalForm}_i(f, \{g_1, \ldots, g_i\})$
1 $f := \mathrm{map}(\mathsf{NormalForm}_{i-1}, \mathrm{Coeffs}(f, x_i), \{g_1, \ldots, g_{i-1}\})$
2 $S_i := \mathrm{Rev}(g_i)^{-1} \mod g_1, \ldots, g_{i-1}, x_i^{\deg(f,x_i)-\deg(g_i,x_i)+1}$
3 $D := \mathrm{Rev}(f)S_i \mod x_i^{\deg(f,x_i)-\deg(g_i,x_i)+1}$
4 $D := \mathrm{map}(\mathsf{NormalForm}_{i-1}, \mathrm{Coeffs}(D, x_i), \{g_1, \ldots, g_{i-1}\})$
5 $D := g_i \mathrm{Rev}(D)$
6 $D := \mathrm{map}(\mathsf{NormalForm}_{i-1}, \mathrm{Coeffs}(D, x_i), \{g_1, \ldots, g_{i-1}\})$
7 **return** $f - D$

Observe that performing $\mathsf{NormalForm}_1$ simply amounts to two multiplications. More generally, $\mathsf{NormalForm}_i$ requires two multiplications and $3(\delta_i + 1)$ recursive calls to $\mathsf{NormalForm}_{i-1}$. In [17], the authors have shown that $\mathsf{NormalForm}_n(f, \{g_1, \ldots, g_n\})$ runs in $O(4^n \delta \lg(\delta) \lg(\lg(\delta)))$ operations in $\mathbb{K}$.

3 Cutoff Analysis for Dense Bivariate Multiplication

As mentioned in the introduction, the work in [19] identified balanced bivariate multiplication as a good kernel for dense multivariate multiplication. Bivariate multiplication based on FFT techniques, and under the 1-D FFT black box assumption, can be done via either 2-D FFT or 2-D TFT. More precisely, the necessary 1-D FFTs of the algorithm of Section 2.3 can be performed via either the Cooley-Tukey algorithm or TFT. In order to optimize our balanced bivariate multiplication code, we need to determine when to use these different 1-D FFT routines. This section offers a first answer based on algebraic complexity analysis meanwhile Section 4 will provide experimental results.

Recall that we aim at applying bivariate multiplication to balanced pairs. In practice, as mentioned in [19], using "nearly balanced" pairs is often sufficient. Hence, with the notations of Section 2, we can assume that $d_1 + d'_1$ and $d_2 + d'_2$ are of the same order of magnitude. Moreover, it is often the case that d_i and d'_i are quite close, for all $1 \leq i \leq n$. For instance, in normal form computations, we have $d_i \leq 2d'_i - 2$ for all $1 \leq i \leq n$ (up to exchanging the role of f and g). Therefore, in both the experimental analysis of Section 4 and in the complexity analysis of the present section, we can assume that all partial degrees d_1, d'_1, d_2, d'_2 are of the same order. To keep experimentation and estimates manageable, we will assume that they all belong to a range $[2^k, 2^{k+1})$ for some $k \geq 2$. In the case of our complexity analysis, we will further assume that all d_1, d'_1, d_2, d'_2 are equal (or close) to a value d in such a range.

We call *degree cut-off FFT vs TFT* a value $d \in [2^k, 2^{k+1})$ such that the work (or algebraic complexity) of bivariate multiplication based on 2-D TFT is less than the one of bivariate multiplication based on 2-D FFT. Our objective in the sequel of this section is to determine the smallest possible cut-off for a given value of k.

To this end, we have developed a MAPLE package (available upon request) that manipulates polynomials with rational number coefficients and with k and 2^k as variables. We denote by $\mathbb{Q}[k, 2^k]$ the set of these objects. It satisfies all the usual algebraic rules on such expressions plus other operations targeting complexity analysis. For instance, our package computes symbolic logarithms of appropriate elements of $\mathbb{Q}[k, 2^k]$ and performs asymptotic majorations (i.e. majorations that hold for k big enough).

The table below gives the work for the algorithm of Section 2.3 when 1-D FFTs are performed by TFT. Note that for all d in the range $[2^k, 2^{k+1})$ the work of the Cooley-Tukey bivariate multiplication is $48 \times 4^k(3k + 7)$.

Determining degree cut-off's (for FFT vs TFT) implies solving inequalities of the form $p > 0$ for $p \in \mathbb{Q}[k, 2^k]$. We achieve this by using standard techniques of real function analysis and we omit the details here.

For different values d of the form $2^k + c_1 2^{k-1} + \cdots + c_7 2^{k-7}$ where each $c_1, \ldots, c_7$ is either 0 or 1, we have compared the work of our bivariate multiplication based on either FFT or TFT. The above table lists some of our findings. These results suggest that for

Table 1. Work of TFT-based bivariate multiplication

d	Work
2^k	$3(2^{k+1}+1)^2(7+3k)$
$2^k + 2^{k-1}$	$81\ k\ 4^k + 270\ 4^k + 54\ k\ 2^k + 180\ 2^k + 9k + 30$
$2^k + 2^{k-1} + 2^{k-2}$	$\frac{441}{4}\ k\ 4^k + \frac{735}{2}\ 4^k + 63\ k\ 2^k + 210\ 2^k + 9k + 30$
$2^k + 2^{k-1} + 2^{k-2} + 2^{k-3}$	$\frac{2025}{16}\ k\ 4^k + \frac{3375}{2}\ 4^k + \frac{135}{2}\ k\ 2^k + 225\ 2^k + 9k + 30$

Table 2. Degree cut-off estimate

$(c_1, c_2, c_3, c_4, c_5, c_6, c_7)$	Range for which this is a cut-off
$(1,1,1,0,0,0,0)$	$3 \leq k \leq 5$
$(1,1,1,0,1,0,0)$	$5 \leq k \leq 7$
$(1,1,1,0,1,1,0)$	$6 \leq k \leq 9$
$(1,1,1,0,1,1,1)$	$7 \leq k \leq 11$
$(1,1,1,1,0,0,0)$	$11 \leq k \leq 13$
$(1,1,1,1,0,1,0)$	$14 \leq k \leq 18$
$(1,1,1,1,1,0,0)$	$19 \leq k \leq 28$

every range $[2^k, 2^{k+1})$ that occur in practice (see Section 4) a sharp (or minimal) degree cut-off is around $2^k + 2^{k-1} + 2^{k-2} + 2^{k-3}$. Our experimental results lead in fact to $2^k+2^{k-1}+2^{k-2}$ on 1 core, which seems to us coherent. Indeed our complexity analysis does not take several important factors such as memory management overhead and etc.

4 Efficient Implementation for Balanced Bivariate Multiplication

In this section we present our implementation techniques for FFT-based polynomial multiplication on multi-cores. These techniques avoid unnecessary calculations and reduce memory movement, Although we focus on balanced bivariate multiplication, our explanation covers to the general case, which we had to consider anyway. Indeed, optimizing our code in the general multivariate case was necessary to make a fair performance evaluation of our balanced bivariate multiplication and our reduction to it. We evaluate the performance of our implementation using VTune [4] and Cilkscreen [3]. We further compare the performance and determine experimentally the cut-off between TFT- and FFT-based bivariate multiplication for a large range of polynomials.

4.1 Implementation Techniques

As in Section 2, let $f \in \mathbb{K}[x_1, \cdots, x_n]$ be a multivariate polynomial with degree d_i in x_i, for all $1 \leq i \leq n$. We represent f in a *dense recursive manner* w.r.t. the variable order $x_1 < \ldots < x_n$. This encoding, which is similar to an $n-$dimensional matrix in row-major layout, is defined as follows:

– The coefficients of f are stored in a contiguous one-dimensional array B.
– The coefficient of the term $x_1^{e_1} \ldots x_n^{e_n}$ is indexed by $\ell_1 \cdots \ell_{n-1} e_n + \ell_1 \cdots \ell_{n-2} e_{n-1} + \cdots + \ell_1 e_2 + e_1$ in B, where $\ell_i = d_i + 1$ for all $1 \leq i \leq n$.

The parallelization of the multiplication algorithm in Section 2.3 takes advantage of the ease-of-use of the parallel constructs in `Cilk++`. For instance, when evaluating a polynomial by means of a n-dimensional FFT, the 1-D FFTs computed along the i-th dimension (for each $1 \leq i \leq n$) are parallelized by a `cilk_for` loop. Similarly, *Step 2* of the multiplication algorithm is performed by a `cilk_for`. A special care is needed, however, for handling the large data sets and frequent memory access that this algorithm may involve. We described below the challenges and our solutions.

Data transposition. A number of $n-1$ data transpositions is needed when performing the row-column n-dimensional algorithm. Data transposition is purely memory-bound and can be a bottleneck. For the 2-dimensional case, we use the cache-efficient code provided by Matteo Frigo. It employs a divide-conquer approach presented in [8] which fits the base case into the cache of the targeted machine. For the multi-dimensional case, we divide the problem into multiple 2-dimensional transpositions where Matteo Frigo's cache-efficient code can be applied. Consider for instance a trivariate polynomial representation, with dimension sizes of s_1, s_2, s_3 and where the variable ordering has to be changed from $x_1 < x_2 < x_3$ to $x_3 < x_2 < x_1$. First we exchange x_1 and x_2 by means of s_3 number of 2-dimensional transpositions of size $s_1 s_2$. This gives the order of $x_2 < x_1 < x_3$. To exchange x_2 and x_3, we view variables x_2 and x_1 as one variable and group their coefficients vector into one of size $\ell = s_2 s_1$. Then, one 2-dimensional transposition of size ℓs_3 will suffice. The resulting variable order is now $x_3 < x_2 < x_1$.

Avoiding unnecessary calculations and reducing memory movement. We first discuss the implementation of the algorithm of Section 2.3 when the 1-D FFTs are performed by the (radix 2) Cooley-Tukey algorithm. We allocate two workspaces A and B each with size $s = s_1 \cdots s_n$ for the evaluation of f and g respectively. To prepare for the interpolation of the product $f\,g$, the coefficient data of f and g are scattered to the appropriate positions of A and B. One can make these data movements first and then evaluate the polynomials. We use a more cache-efficient way instead. On the evaluation of the first variable of f, we copy in parallel each of the $(d_2+1) \cdots (d_n+1)$ number of coefficient vectors of size d_1+1 from f to the corresponding vector of size s_1 in A and continue the evaluation of this vector in A by a 1-D FFT in place. This method improves the data-locality. It also saves from not doing FFTs on $(s_2 \cdots s_n) - (d_2+1) \cdots (d_n+1)$ number of size s_1 vectors of zeros. We proceed similarly with g.

For the TFT version of the algorithm of Section 2.3, we have realized two implementations. One implementation is similar to the one above and we call it "in-place". Each data transposition has to be done for the size s, but the number of 1-D FFTs in an evaluation or interpolation are bounded by $ps = (d_1 + d'_1 + 1) \cdots (d_n + d'_n + 1)$, the size of the product. Our second implementation is "out-of-place". We only allocate one extra workspace C of size ps. C is used for the evaluation of g. We will use the space of the product for the evaluation of f, and the result will be in the right place. Savings can be gained on the evaluation of the first variable in the same way as above. The difference is as follows. To evaluate a vector v_i of size $d_i + d'_i + 1$ in variable x_i,

we allocate a temporary vector t_i of size s_i as a workspace and copy the data in v_i to s_i and then perform 1-D TFT in t_i; t_i is freed after use.

Our benchmark shows that the "out-of-place" TFT-based method is more efficient for balanced problems (that is when the partial degrees of the product are equal); moreover the performance of this approach becomes even better when the number of cores increases. However, for problems with unequal partial degrees in the product, the "in-place" TFT-based method works better. We will study in a future work the causes of this behavior.

4.2 Performance Evaluation

We use VTune [4] and Cilkscreen [3] to evaluate the performance of our implementation. We measure the instruction, cache and parallel efficiency on 8 processors for the multiplication of bivariate polynomials with partial degrees in the range of $[2047, 4096]$.

Table 3 lists a selection of events and ratios reported by VTune for FFT-based and TFT-based methods respectively. 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 Intel Xeon X5460 @3.16GHz and has 6144 KB of L2 cache.

For all the tested problems by either FFT or TFT method, their clocks per instructions retired (CPI) is around 0.8. Their L2 cache miss rates are below 0.0008. The very small modified data sharing ratios (less than 0.00025) imply that, chances of threads racing on using and modifying data laid in one cache line are very low. However, TFT-based method use about three times less number of instructions retired than FFT-based for problems which are worst cases for FFT, such as $(2048, 2048)$, $(2048, 4096)$ and $(4096, 4096)$. These account for the better timing of TFT-based method for a certain range of degrees, shown in the figures of next section.

Table 3. Performance evaluation by VTune for TFT-based and FFT-based method

Method	d_1 d_2	INST_RETIRED ($\times 10^9$)	Clocks per instructions retired (CPI)	L2 cache miss rate ($\times 10^{-3}$)	Modified data sharing ratio ($\times 10^{-3}$)	Time on 8 cores (s)
TFT-based	2047 2047	44	0.794	0.423	0.215	0.86
	2048 2048	52	0.752	0.364	0.163	1.01
	2047 4095	89	0.871	0.687	0.181	2.14
	2048 4096	106	0.822	0.574	0.136	2.49
	4095 4095	179	0.781	0.359	0.141	3.72
	4096 4096	217	0.752	0.309	0.115	4.35
FFT-based	2047 2047	38	0.751	0.448	0.106	0.74
	2048 2048	145	0.652	0.378	0.073	2.87
	2047 4095	79	0.849	0.745	0.122	1.94
	2048 4096	305	0.765	0.698	0.094	7.64
	4095 4095	160	0.751	0.418	0.074	3.15
	4096 4096	622	0.665	0.353	0.060	12.42

We use Cilkscreen to estimate the parallelism of these running instances by measuring their *work* and *span*; recall that we rely on the multi-threaded parallelism model introduced in [2]. The data measured by Cilkscreen are summarized in Table 4 . The parallel overhead appears very low and the burdened parallelism is nearly equal to the expected parallelism. The speedup factors that we obtain, reported in the next section, are as good as the estimated ones.

Table 4. Performance evaluation by Cilkscreen for TFT-based and FFT-based method

Method	d_1 d_2	Work ($\times 10^9$)	Span/ Burdened span ($\times 10^9$)	Parallelism/ Burdened parallelism	Speedup estimates 4P	8P	16P
TFT-based	2047 2047	45	0.613/0.614	74.18/74.02	3.69-4	6.77-8	11.63-16
	2048 2048	53	0.615/0.616	86.35/86.17	3.74-4	6.96-8	12.22-16
	2047 4095	109	0.118/0.118	92.69/92.58	3.79-4	7.09-8	12.54-16
	2048 4096	125	1.184/1.185	105.41/105.27	3.80-4	7.19-8	12.88-16
	4095 4095	193	2.431/2.433	79.29/79.24	3.71-4	6.86-8	11.89-16
	4096 4096	223	2.436/2.437	91.68/91.63	3.76-4	7.03-8	12.43-16
FFT-based	2047 2047	40	0.612/0.613	65.05/64.92	3.64-4	6.59-8	11.08-16
	2048 2048	155	0.619/0.620	250.91/250.39	3.80-4	7.50-8	14.55-16
	2047 4095	98	1.179/1.180	82.82/82.72	3.77-4	6.99-8	12.23-16
	2048 4096	383	1.190/1.191	321.75/321.34	3.80-4	7.60-8	14.82-16
	4095 4095	169	2.429/2.431	69.39/69.35	3.66-4	6.68-8	11.35-16
	4096 4096	392	2 .355/2.356	166.30/166.19	3.80-4	7.47-8	13.87-16

4.3 Cut-off between TFT- and FFT-based Methods

We compare the performances of the FFT- and TFT-based bivariate multiplication. More precisely and as discussed in Section 3, we consider that all partial degrees d_1, d_2, d'_1, d'_2 are between two consecutive powers of 2 and we would like to determine for which degree patterns the TFT approach outperforms the FFT one. We study the following three degree ranges: $[256, 512)$, $[1024, 2048)$ and $[4096, 8192)$. In general, we use the "in-place" implementations of FFT- and TFT-based methods. When the partial degrees of the product are equal, the "out-of-place" TFT-based method is used.

Table 5. Sizes and cut-offs of three sets of problems

No.	Input degree range	Product size range	Size cut-off on 1 core	8 cores	12 cores	16 cores
1	256-511	263169-1046529	786596	814012	861569	
2	1023-2047	4198401-16769025	12545728	13127496	14499265	16433645
3	4095-8191	67125249-268402689	202660762	207958209	227873850	257370624

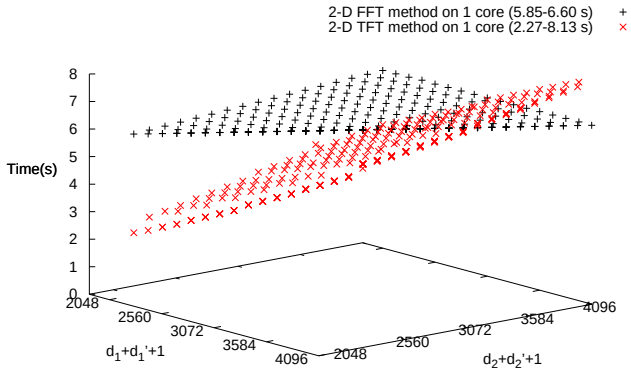

Fig. 1. Timing of bivariate multiplication for input degree range of $[1024, 2048)$ on 1 core.

Table 6. Cut-off details between TFT-based and FFT-based bivariate multiplication

No.	#cores	TFT-based Method				FFT-based Method			
		Time (s)	Speedup factor	Faster portion	Time ratio TFT/FFT	Time (s)	Speedup factor	Faster portion	Time ratio FFT/TFT
1	1	0.120-0.419		75%	2.55-1.0	0.305-0.338		25%	1.0-1.20
	8	0.020-0.065	5.5-6.7	78%	2.50-1.0	0.050-0.055	6.0-6.3	22%	1.0-1.17
	12	0.015-0.048	5.5-9.0	82%	2.73-1.0	0.038-0.044	7.4-8.2	18%	1.0-1.15
2	1	2.27-8.13	2.58-1.0	75%		5.85-6.60		25%	1.0-1.20
	8	0.309-1.08	6.8-7.6	78%	2.61-1.0	0.806-0.902	7.2-7.3	22%	1.0-1.16
	12	0.224-0.779	8.2-11.0	86%	2.77-1.0	0.613-0.707	9.3-9.8	14%	1.0-1.09
	16	0.183-0.668	7.8-14.1	98%	3.18-1.0	0.588-0.661	9.6-10.8	2%	1.0-1.02
3	1	42.2-154.3		76%	2.63-1.0	110.9-123.2		24%	1.0-1.20
	8	5.52-20.07	6.8-7.8	77%	2.69-1.0	14.82-16.57	7.4-7.6	23%	1.0-1.17
	12	3.75-14.10	9.0-11.4	85%	2.92-1.0	10.96-12.72	9.9-10.3	15%	1.0-1.03
	16	3.09-11.36	9.5-14.9	96%	3.12-1.0	9.55-11.02	11.0-12.0	4%	1.0-1.04

The sizes and the cut-offs on 1, 8, 12 and 16 cores for the three sets of problems are summarized in Table 5. Table 6 lists the timings, the speedup factors, the percentages of the size range and the ratio by which TFT or FFT is superior for the three sets of problems on 1, 8, 12 and 16 cores. To provide an insight view of the benchmarks, we display the complete timing results and their cut-off regression w.r.t. the size of the product for the range of $[1024, 2048)$ on 1, 8 and 16 cores in Figures 1 to 6.

Figures 1 to 6 reveal clearly the different performances of FFT- and TFT-based methods for the problems in set 2 with partial degrees in the range of $[1024, 2048)$, which is $[2^{10}, 2^{11})$. Here, the timing of FFT-based method is about the same for all the

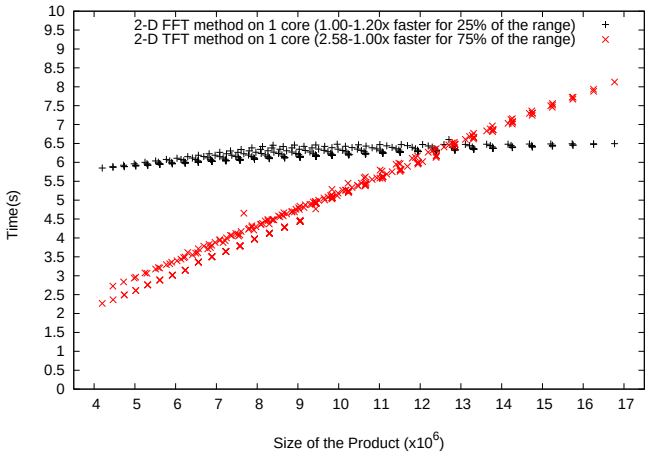

Fig. 2. Size cut-off for input degree range of $[1024, 2048)$ on 1 core.

problems, but the timing of TFT-based method is correlated to the size of the partial degrees. This result agrees with our complexity analysis reported in Expressions (1) and (2) in Section 2.3. Expression (1) indicates that the work for multiplying any pair of n-variate polynomials with partial degrees d_i and d'_i in a range of $[2^k, 2^{k+1})$ by FFT-based method is constant, and determined by the value of s, which is $2^{n(k+2)}$. This reflects the well-known *staircase* phenomenon of FFT. Meanwhile, the work of TFT-based method grows linearly with $\prod_{i=1}^{n}(d_i + d'_i + 1)$, as indicated by Expression (2). Therefore, in a certain lower range of $[2^k, 2^{k+1})$, TFT-based method performs better.

Overall, both FFT- and TFT-based methods show good speedup factors on 8 to 16 cores, with peak performance for the latter. The cut-off criteria are similar for all degree ranges, independent of the magnitude of the degrees. In a degree range of $[2^k, 2^{k+1})$, the percentage of problems for which the TFT-based method outperforms FFT's increases with the number of cores. On one core, the TFT-based method is better for the first 75% of the sizes by at most a factor of 2.6. This is coherent to the theoretical analysis result in Section 3, taking into account the memory management overhead in our implementation. On 16 cores, the TFT-based method is superior for up to 98% of the problem range by a maximum factor of 3.2. The mechanism that favors the TFT-based method on multi-cores will be studied further.

5 Parallelism Estimates for Normal Form Computations

The recursive structure of the procedure $\mathsf{NormalForm}_n$ in Section 2.5 offers opportunities for concurrent execution. Moreover, this procedure relies on multivariate multiplication and we can hope to increase parallelism by relying on our parallel multiplication. Estimating this extra parallel speedup factor is a fundamental problem, as discussed in the introduction. This section offers a first answer based on complexity analysis meanwhile Section 6 will provide experimental results.

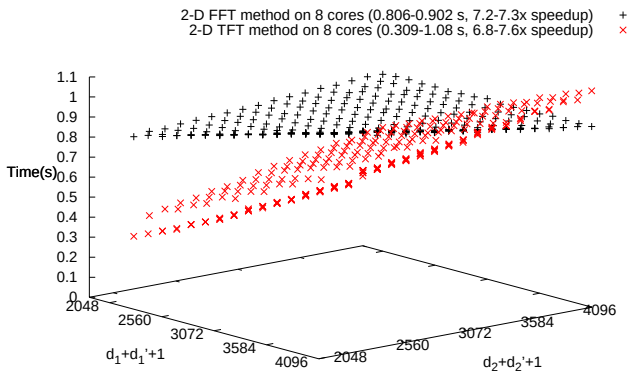

Fig. 3. Timing of bivariate multiplication for input degree range of $[1024, 2048)$ on 8 cores.

We first consider the span (or parallel running time) of the algorithms in Section 2 by means of the multi-threaded programming model of [9]. This model, however, does not explicitly cover parallel for-loops, which are needed for both multiplication and normal form computations.

Following the way a `cilk_for` loop is implemented in the `cilk++` language [3], we assume that the span of a for-loop of the form

```
for i from 1 to n do BODY(i); end for;
```

is bounded by $O(\lg(n)S)$ where S is the maximum span of `BODY(i)` for `i` in the range `1..n`. Consequently the span of a nested for-loop

```
for j in 1..m do
  for i from 1 to n do BODY(i); end for;
end for;
```

is bounded by $O(\lg(n)\lg(m)S)$.

Defining $s = s_1 \cdots s_n$ and $\ell = \prod_{i=1}^{n} \lg(s_i)$, it is easy to check that the span of the multiplication algorithm of Section 2.3 is

$$3\sum_{i=1}^{n}(\prod_{j\neq i}\lg(s_j))s_i\lg(s_i) + 3\prod_{i=1}^{n}\lg(s_i) = 3\ell\left(\sum_{i=1}^{n} s_i + 1\right) \tag{3}$$

operations in $\mathbb{K}$, when the Cooley-Tukey algorithm is used for 1-D FFTs. This estimates becomes

$$3\sum_{i=1}^{n}\frac{\ell}{\lg(s_i)}(d_i + d'_i + 1)(\lg(s_i) + 1) \quad + \quad 3\prod_{i=1}^{n}\lg(d_i + d'_i + 1), \tag{4}$$

which is the same order of magnitude as

$$3\ell\sum_{i=1}^{n}(d_i + d'_i + 1) \quad + \quad 3\prod_{i=1}^{n}\lg(d_i + d'_i + 1), \tag{5}$$

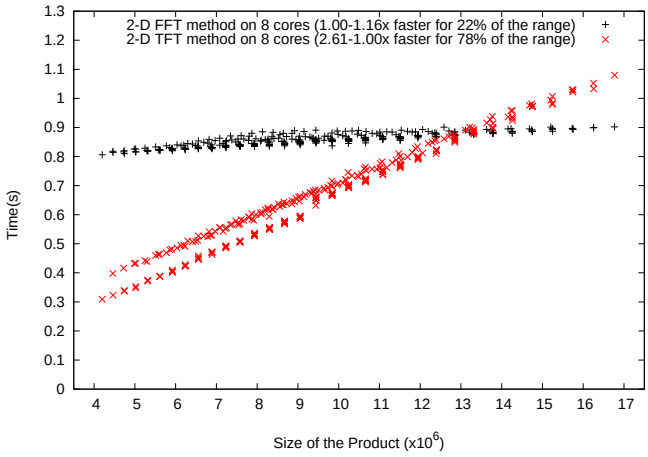

Fig. 4. Size cut-off for input degree range of $[1024, 2048)$ on 8 cores.

when all the s_i become large.

We turn now to the span estimates for the procedure $\mathsf{NormalForm}_i$ when applied to f and $\{g_1, \ldots, g_i\}$ where $g_1, \ldots, g_i$ is a reduced monic triangular set of $\mathbb{K}[x_1, \ldots, x_i]$ (see Section 2) and f is a polynomial of $\mathbb{K}[x_1, \ldots, x_i]$. In practice, the partial degree of f w.r.t x_i is at most $2\delta_i - 2$, for all $1 \leq i \leq n$. Indeed, the polynomial f is often the product of two polynomials a and b which are reduced w.r.t. the reduced monic triangular set $\{g_1, \ldots, g_i\}$, that is, which satisfy $\deg(a, x_j) < \delta_j$ and $\deg(b, x_j) < \delta_j$ for all $1 \leq j \leq i$.

Define $\underline{\delta}_i = (\delta_1, \ldots, \delta_i)$. Let us denote by $\mathsf{W}_\mathsf{M}(\underline{\delta}_i)$ and $\mathsf{S}_\mathsf{M}(\underline{\delta}_i)$ the work and span of a multiplication algorithm applied to h and g_i where h satisfies $\deg(h, x_j) < \delta_j$ for $1 \leq j < i$ and $\deg(h, x_i) \leq 2\delta_i - 2$. Let also $\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_i)$ be the span of $\mathsf{NormalForm}_i$ applied to f and $\{g_1, \ldots, g_i\}$. If the procedure $\mathsf{NormalForm}_i$ is run with a serial multiplication, then we have:

$$\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_i) = 3\,\ell_i\,\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_{i-1}) + 2\,\mathsf{W}_\mathsf{M}(\underline{\delta}_i) + \ell_i \tag{6}$$

where $\ell_i = \prod_{j=1}^{i} \lg(\delta_j)$. Similarly, if the procedure $\mathsf{NormalForm}_i$ is run with a parallel multiplication, we obtain:

$$\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_i) = 3\,\ell_i\,\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_{i-1}) + 2\,\mathsf{S}_\mathsf{M}(\underline{\delta}_i) + \ell_i. \tag{7}$$

Neglecting logarithmic factors and denoting by d the maximum of δ_i for all $1 \leq i \leq n$, the span $\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_i) \in O(3^n d^n)$ if a serial multiplication is used, otherwise $\mathsf{S}_{\mathrm{NF}}(\underline{\delta}_i) \in O(3^n d)$ if a parallel multiplication is used. Since the work $\mathsf{W}_{\mathrm{NF}}(\underline{\delta}_i) \in O(4^n d^n)$ (again neglecting logarithmic factors) this implies that work, span and parallelism (i.e. the ratio of work divided by span) are all exponential in the number of variables. This suggests that obtaining efficient parallel implementation of the procedure $\mathsf{NormalForm}_i$ is interesting but also challenging.

In Tables 7 and 8, the span of $\mathsf{NormalForm}_i$ is computed for $i = 1, 2, 3$ and $\delta_1 = \cdots = \delta_i = d$ with $d \in \{2^k, 2^k + 2^{k-1}\}$. For $i = 1$, the spans of $\mathsf{NormalForm}_i$ with

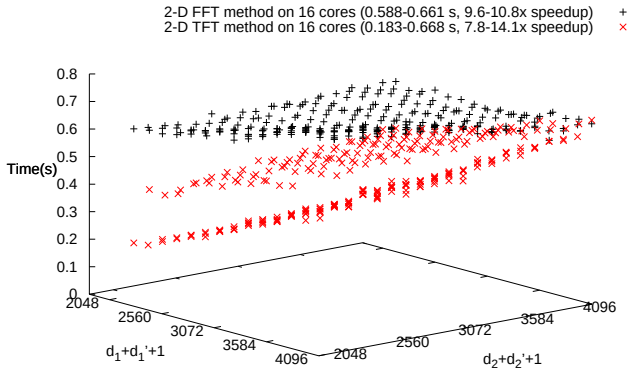

Fig. 5. Timing of bivariate multiplication for input degree range of $[1024, 2048)$ on 16 cores.

Table 7. Span of TFT-based normal form for $\underline{\delta}_i = (2^k, \ldots, 2^k)$.

i	With serial multiplication	With parallel multiplication
1	$18\ k\ 2^k + 44\ 2^k + 10k + 22$	$12\ k\ 2^k + 24\ 2^k + 11\ k + 20$
2	$72\ k\ 4^k + 168\ 4^k + \mathrm{o}(4^k)$	$60\ k^2\ 2^k + 168\ k^2\ 2^k + 96\ 2^k + \mathrm{o}(2^k)$
3	$216\ k\ 8^k + 496\ 8^k + \mathrm{o}(8^k)$	$216\ k^3\ 2^k + 720\ k^2\ 2^k + 720\ k\ 2^k + 288\ 2^k + \mathrm{o}(2^k)$

or without parallel multiplication are essentially the same. Indeed we assume that 1-D FFTs are run serially. The slight gain is explained by the fact in **Step 1** of the algorithm of Section 2.5. One can evaluate the two input polynomials concurrently.

For $i = 2$, the gain obtained from the use of a parallel multiplication is asymptotically in the order of $\Theta(2^k/k)$. For $i = 3$, this becomes $\Theta(4^k/k^2)$. This suggests that a parallel multiplication code (even under the 1-D FFT black box assumption) can speedup substantially a parallel code for $\mathsf{NormalForm}_i$ with $i \geq 2$.

In Table 9, we provide the span of $\mathsf{NormalForm}_i$ for another degree pattern, namely for $\delta_i = 2^k$ and $\delta_{i-1} = \cdots = \delta_i = 1$. This configuration is actually the general one for

Table 8. Span of TFT-based normal form for $\underline{\delta}_i = (2^k + 2^{k-1}, \ldots, 2^k + 2^{k-1})$.

i	With serial multiplication	With parallel multiplication
1	$27\ k\ 2^k + 93\ 2^k + 10k + 32$	$18\ k\ 2^k + 54\ 2^k + 11\ k + 27$
2	$162\ k\ 4^k + 540\ 4^k + \mathrm{o}(4^k)$	$90\ k^2\ 2^k + 396\ k^2\ 2^k + 378\ 2^k + \mathrm{o}(2^k)$
3	$729\ k\ 8^k + 2403\ 8^k + \mathrm{o}(8^k)$	$324\ k^3\ 2^k + 1836\ k^2\ 2^k + 3186\ k\ 2^k + 1782\ 2^k + \mathrm{o}(2^k)$

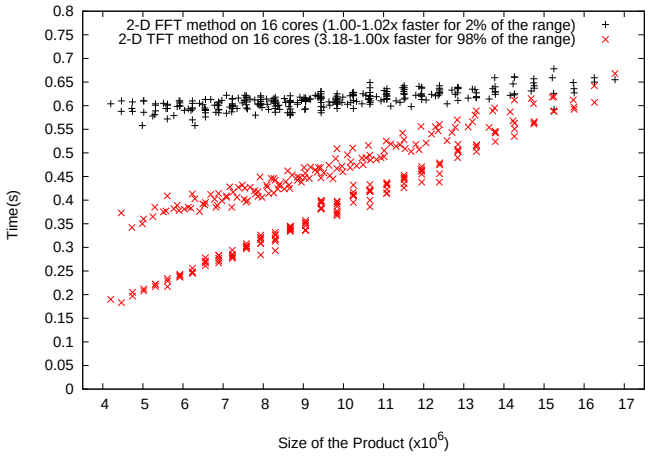

Fig. 6. Size cut-off for input degree range of $[1024, 2048)$ on 16 cores.

the polynomials describing the symbolic solutions of polynomial systems with finitely many solutions. We call it *Shape Lemma* after the landmark paper [1] where this degree pattern was formally studied and from which the terminology is derived.

In Table 10, we provide the limit of the parallelism of $\mathsf{NormalForm}_i$ when k goes to $+\infty$ (that is the ratio between work and span) for the same degree patterns as in Table 9.

Table 9. Span of TFT-based normal form for $\underline{\delta}_i = (2^k, 1, \ldots, 1)$.

i	With serial multiplication	With parallel multiplication
2	$144\ k\ 2^k + 642\ 2^k + 76\ k + 321$	$72\ k\ 2^k + 144\ 2^k + 160\ k + 312$
4	$4896\ k\ 2^k + 45028\ 2^k + 2488\ k + 22514$	$1296\ k\ 2^k + 2592\ 2^k + 6304\ k + 12528$
8	$3456576\ k\ 2^k + 71229768\ 2^k + \mathrm{o}(2^k)$	$209952\ k\ 2^k + 419904\ 2^k + \mathrm{o}(2^k)$

Table 9 suggests that for *Shape Lemma* degree patterns and for a fixed number of variables, the extra speedup factor provided by a parallel multiplication (w.r.t. a serial one) is upper bounded by a constant. This does not imply that using parallel multiplication in a parallel normal form for *Shape Lemma* degree patterns would be of limited practical interest. Table 10 suggests that a parallel multiplication can indeed increase the parallelism of $\mathsf{NormalForm}_i$, in particular when the number of variables is large.

These complexity estimates do not take into account parallel overhead. In the case of $\mathsf{NormalForm}_i$, those are potentially large. Indeed, after Steps 1, 4 and 6 of $\mathsf{NormalForm}_i$, there is a synchronization point for $(\delta_i + 1)$ threads, namely the $(\delta_i + 1)$ recursive calls to $\mathsf{NormalForm}_{i-1}$. Observe also that the number of synchronization points of

Table 10. Parallelism estimates of TFT-based normal form for $\underline{\delta}_i = (2^k, 1, \ldots, 1)$.

i	With serial multiplication	With parallel multiplication
2	$13/8 \simeq 2$	$13/4 \simeq 3$
4	$1157/272 \simeq 4$	$1157/72 \simeq 16$
8	$5462197/192032 \simeq 29$	$5462197/11664 \simeq 469$

$\mathsf{NormalForm}_i$ is 3^{i-1}. This puts a lot of "burden" on the parallelism of this procedure. Section 6 will tell how much can really be achieved in practice.

6 Efficient Parallel Computation of Normal Forms

We present now our experimental results for efficient parallelization of normal forms. Our key techniques include reducing multivariate multiplication to bivariate and composing the parallelism of bivariate multiplications with that at the level of the normal form algorithm, as discussed in Section 5. We study three typical degree patterns, illustrated in Figures 7, 8 and 9. All the benchmarks show the important role of parallel bivariate multiplication in improving the performance of normal form computations.

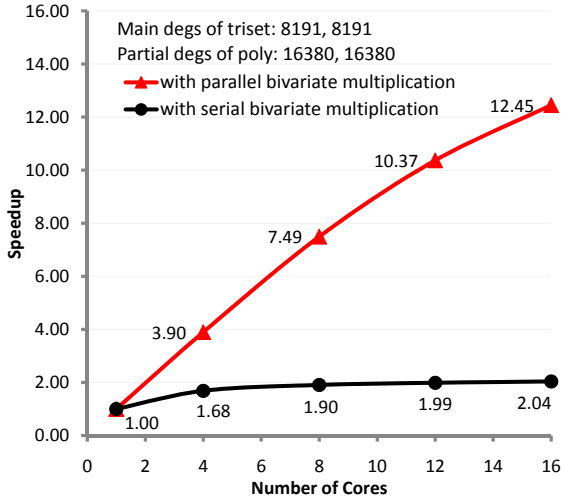

Fig. 7. Normal form computation of a large bivariate problem.

Figure 7 displays the speedup factors for computing the normal form of a bivariate problem with serial bivariate multiplication and with parallel bivariate multiplication. The main degrees of the polynomials of a triangular set are 8191 and 8191, and the partial degrees of the polynomial under simplification are 16380 and 16380. The speedup

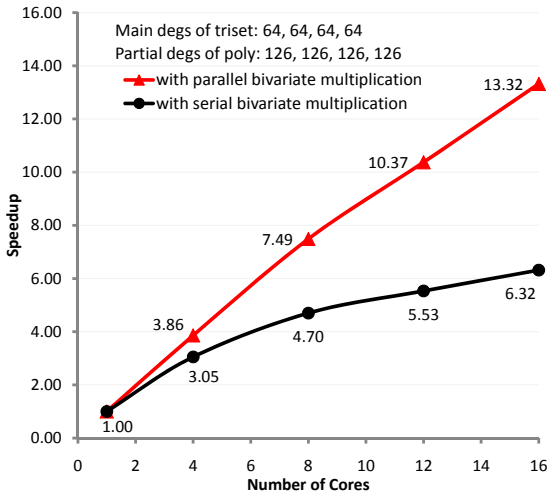

Fig. 8. Normal form computation of a medium-sized 4-variate problem.

of normal form computation without parallel multiplication is very poor, about 2.0 on 16 cores. The parallelization of the bivariate multiplications helps improving the performance significantly, by a factor of 6.

Figure 8 demonstrates the 2.0 times of improvement contributed by the parallel bivariate multiplications involved in the normal form computation of a medium-sized 4-variate problem, where all the main degrees of the triangular set are equal to 64, and all the partial degrees of the polynomial to be reduced are 126. The 8-variate problem, described in Figure 9 with main degree pattern of "$1024, 2, 2, 2, 2, 2, 2, 2$", shows a speedup of 10.75 on 16 cores without parallel bivariate multiplication. Composed with parallel bivariate multiplication it can achieve a more satisfactory speedup of 13.4.

These results show that, when the number of variables is small, say 2, the parallelism of our normal form routine can be small. However, if the input polynomial degrees are large enough, parallel multiplication can increase the overall parallelism substantially. In the Shape Lemma case, when the number of variables is large, say 8, our normal form routine already possesses a high parallelism. Hence, even though parallel multiplication cannot help as much as in the previous case, the combination of the two parallel code brings again high performance.

7 Concluding Remarks

We have reported implementation strategies for FFT-based dense polynomial arithmetic targeting multi-cores. We have extended our preliminary study [19] dedicated to multiplication leading to a complete set of efficient routines for polynomial arithmetic operations, including normal form computations.

Since balanced bivariate multiplication is the kernel to which most of these routines reduce, we have conducted an in-depth study on the implementation techniques for this

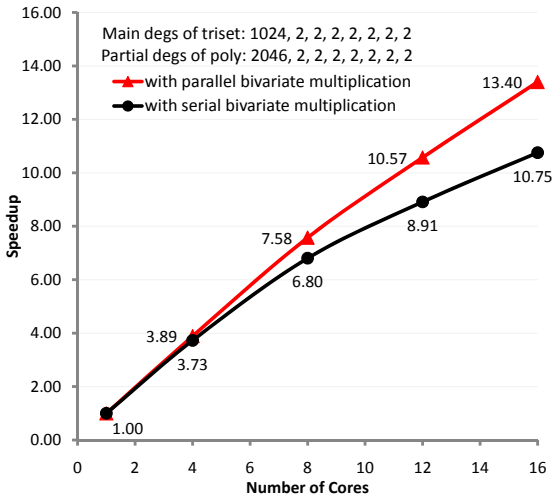

Fig. 9. Normal form computation of an irregular 8-variate problem.

operation. Our performance analysis by VTune and Cilkscreen show that our implementations have good instruction and cache efficiency, and good parallelism as well. In particular we have determined cut-off criteria between two variants of this balanced bivariate multiplication based respectively on Cooley-Tukey FFT and the Truncated Fourier Transform on multi-cores. The cut-off criteria are similar for all degree ranges that we have tested. However, for a fixed degree range of the form $[2^k, 2^{k+1})$, the percentage of problems for which TFT-based method outperforms FFT's increases with the number of cores.

We have explained why the parallelization of normal form computation is challenging and also of great importance in symbolic computation. We have shown that, not only efficient parallel multiplication can improve the performance of parallel normal form computation, but also that this composition is necessary for parallel normal form computation to reach peak performance on all input patterns that we have tested.

For both problems of optimizing balanced bivariate multiplication and performing efficient parallel computation of normal forms, we have combined theoretical and empirical analyses. The former could not provide a precise answer due to simplification hypotheses but helped narrowing the pattern ranges for the latter analysis.

Nevertheless, we would like to obtain more "realistic" results through complexity analysis. In the context of this study, this means being able to better take parallel overhead into account: A subject of future research.

Another future work is the development of higher-level algorithms on top of the basic polynomial algebra subroutines (BPAS) presented in this paper. Our driving application is the solving of polynomial systems symbolically. Our next step toward this goal is the parallelization of polynomial GCDs modulo regular chains, following the work of [15].

Acknowledgements. This work was supported by NSERC and the MITACS NCE of Canada, and NSF under Grants 0540248, 0615215, 0541209, and 0621511. We are also 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.

References

1. E. Becker, T. Mora, M. G. Marinari, and C. Traverso. The shape of the shape lemma. In *Proc. of ISSAC'1994*, pages 129–133, NY, USA, 1994. ACM Press.
2. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *IEEE FOCS94*, 1994.
3. Cilk Arts. Cilk++. http://www.cilk.com/.
4. Intel Company. Intel VTune Performance Analyzer 9.1 for Linux . http://www.intel.com/.
5. S. Cook. *On the minimum computation time of function*. PhD thesis, Harvard Univ., 1966.
6. J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.
7. D. Cox, J. Little, and D. O'Shea. *Using Algebraic Geometry*. Graduate Text in Mathematics, 185. Springer-Verlag, New York, 1998.
8. 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, 1999.
9. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN*, 1998.
10. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge Univ. Press, 1999.
11. J. van der Hoeven. Truncated Fourier transform. In *Proc. ISSAC'04*. ACM Press, 2004.
12. S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson. Implementation of strassen's algorithm for matrix multiplication. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 1996. IEEE Computer Society.
13. H. T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22:341–348, 1974.
14. X. Li and M. Moreno Maza. Multithreaded parallel implementation of arithmetic operations modulo a triangular set. In *Proc. PASCO'07*, pages 53–59, NY, USA, 2006. ACM Press.
15. X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *Proc. of ISSAC'09*, pages 239–246. ACM Press, 2009.
16. X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. In *MICA'08*, 2008.
17. X. Li, M. Moreno Maza, and É Schost. Fast arithmetic for triangular sets: From theory to practice. In *Proc. ISSAC'07*, pages 269–276, NY, USA, 2007. ACM Press.
18. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
19. M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multicores. In *Proc. PDCAT'09*, Hiroshima, Japan, 2009.
20. M. Sieveking. An algorithm for division of powerseries. *Computing*, 10:153–156, 1972.