Parallel Algorithm for Multiplying Integer Polynomials and Integers

Andrzej Chmielowiec

Abstract This chapter aims to develop and analyze an effective parallel algorithm for multiplying integer polynomials and integers. Multiplying integer polynomials is of fundamental importance when generating parameters for public key cryptosystems, whereas their effective implementation translates directly into the speed of such algorithms in practical applications. The algorithm has been designed specifically to accelerate the process of generating modular polynomials, but due to its good numerical properties it may surely be used to multiply integers. The basic idea behind this new method was to adapt it to parallel computing. Nowadays, it is a very important property, as it allows us to fully exploit the computing power offered by modern processors. The combination of the Chinese Remainder Theorem and the Fast Fourier Transform made it possible to develop a highly effective multiplication method. Under certain conditions our integer polynomial multiplication method is asymptotically faster than the algorithm based on Fast Fourier Transform when applied to multiply both: polynomials and their coefficients. Undoubtedly, this result is the major theoretical conclusion of this chapter.

Keywords CRT · Fast multiplication · FFT multiplication · Integer multiplication · Multiplication algorithm · Parallel multiplication · Polynomial multiplication

Polish National Science Centre grant N N516478340.

A. Chmielowiec (✉)
Institute of Fundamental Technological Research, Polish Academy of Sciences,
Pawinskiego 5B, 02-106 Warszawa, Poland
e-mail: achmielo@ippt.gov.pl; andrzej.chmielowiec@cmmsigma.eu

G.-C. Yang et al. (eds.), *IAENG Transactions on Engineering Technologies*,
Lecture Notes in Electrical Engineering 229, DOI: 10.1007/978-94-007-6190-2_46,

1 Introduction

In 1971 Schönhage and Strassen [15] proposed a new algorithm for large integer multiplication. Since that time, methods based on the Fast Fourier Transform (FFT) have been continuously developed and upgraded. Now we have many multiplication algorithms which are based on the FFT. They are used to multiply integers [4, 13] or power series [10, 14, 17, 18]. Some of them are architecture independent and some are dedicated to a specific processor. The algorithms serve as black boxes which guarantee the asymptotic complexity of the methods using them. However, practical implementation often works in the case of such numbers for which it is ineffective to apply a fast multiplication method. The determination of modular polynomials is a good illustration of this problem. The latest methods for generating classic modular polynomials were developed by Charles, Lauter [1] and Enge [6]. Moreover, Müller [12] proposed another family of modular polynomials which may also be used in the process of counting points on an elliptic curve. The Müller's polynomials are characterized by a reduced number of non-zero coefficients and lower absolute values of coefficients, compared to classic modular polynomials. All the aforesaid authors give the computational complexity of algorithms used to determine modular polynomials based on the assumption that both polynomials and their coefficients are multiplied with the use of the Fast Fourier Transform. The complexity of such a multiplication algorithm is

$$O((n \log n)(k \log k)),$$

where n is the degree of the polynomial, and k is the number of bits of the largest coefficient. However, the application of an asymptotically fast algorithm to multiply numbers becomes effective only when the numbers are of considerable length. According to Garcia's report [7], fast implementation of multiplication in GMP (GNU Multiple Precision Arithmetic Library) becomes as effective as classic multiplication algorithms only for numbers of at least $2^{17} = 131072$ bits. That is why it would be worth to develop a multiplication algorithm which operates fast for polynomials with relatively small coefficients. In order to achieve that, we decided to use the Chinese Remainder Theorem (CRT). This chapter presents results described in our WCE'12 article [2] and extends it to show that the proposed method can also be used to implement fast parallel integer multiplication. In general our idea fits into the scheme proposed in the work [8].

The chapter is organized as follows.

In Sect. 2 for completeness we briefly recall the general idea of the Fast Fourier Transform. The FFT may be implemented in many forms and a choice of proper implementation depends on the problem we want to solve and the processor we are using.

In Sect. 3 we show in detail how to use the CRT to distribute polynomial arithmetic between many processors. Our new method is very simple both in concept and implementation. It does not need any communication between processors, which is

an additional advantage. This algorithm may use any implementation of the FFT. Particularly it may be used with parallel FFT, which reduces the total time of computation.

In Sect. 4 we present numerical results of our 32-bit implementation based on OpenMP parallel programming standard. We compare the proposed method with algorithm based on the FFT over large finite field.

In Sect. 5 we show how fast the 64-bit implementation dedicated and optimized for x86-64 processors is. We compare this implementation to GMP integer multiplication algorithm.

To summarize, to multiply polynomials developer combines two independent techniques to achieve the best performance from a machine or processor:

1. distribution of computations between smaller domains being polynomial rings (apply CRT),
2. optimization of FFT operations within these smaller domains.

The whole idea is illustrated on the following scheme.

$$\begin{array}{ccccccc} & & \mathbb{F}_{p_1}[X] & \xrightarrow{FFT} & \mathbb{F}_{p_1}[X] & & \\ & & \vdots & & \vdots & & \\ \mathbb{Z}[X] & \xrightarrow{CRT} & \mathbb{F}_{p_i}[X] & \xrightarrow{FFT} & \mathbb{F}_{p_i}[X] & \xrightarrow{CRT^{-1}} & \mathbb{Z}[X] \\ & & \vdots & & \vdots & & \\ & & \mathbb{F}_{p_k}[X] & \xrightarrow{FFT} & \mathbb{F}_{p_k}[X] & & \end{array}$$

It means that multiplications in $\mathbb{Z}[X]$ can be distributed between k independent rings $\mathbb{F}_{p_i}[X]$ and each such multiplication can be done independently in parallel.

2 Fast Fourier Transform and its Implementations

A Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform. The basic idea of DFT is to represent polynomials as sequences of values rather than sequences of coefficients. Computing DTF of n values using the definition takes $O(n^2)$ arithmetic operations, while FFT can compute the same result in only $O(n \log n)$ operations. This is the reason why the Fast Fourier Transform plays a very important role in efficient computations and is considered in many publications. Some of them give a general description of the FFT [3, 5, 9, 11], others contain details about very fast implementations [10, 16–18]. In our numerical experiments in the last section a classic algorithm of FFT has been used. However for practical purposes we suggest application of the *cache-friendly truncated FFT*

recently developed [10]. This new FFT method reduces the computational cost and is optimized against modern processor architecture.

3 Using Chinese Remainder Theorem to Distribute Computations Between Many Processors

In the rest of this chapter we will assume that the largest absolute value of polynomial coefficients is less than B. To multiply integer polynomials we have to find a family of finite fields $\mathbb{F}_{p_i}$ in which computations will be done. It is clear that the product $\prod_{i=1}^{k} p_i$ should be large enough to eliminate modular reduction during the multiplication process.

Definition 1.1 *Let* $f(X) = f_{n-1}X^{n-1} + \cdots + f_1X + f_0 \in \mathbb{Z}[X]$ *and* $M \in \mathbb{Z}$. *We define* $f(X) \bmod M$ *as follows*

$$f(X) \bmod M = (f_{n-1} \bmod M)X^{n-1} + \ldots + (f_0 \bmod M),$$

where

$$f_i \bmod M \in \left\{ \left\lfloor \frac{-M+1}{2} \right\rfloor, \ldots, -1, 0, 1, \ldots, \left\lfloor \frac{M-1}{2} \right\rfloor \right\}.$$

Lemma 1.2 *Let* $f(X) = f_{n-1}X^{n-1} + \cdots + f_1X + f_0$, $g(X) = g_{n-1}X^{n-1} + \cdots + g_1X + g_0$ *be polynomials with integer coefficients such that* $|f_i| < B$ *and* $|g_i| < B$. *If integer* M *satisfies the following condition*

$$2nB^2 < M$$

then $f(X)g(X) \bmod M = f(X)g(X)$.

Proof: If $f(X)g(X) = h(X) = h_{2n-2}X^{2n-2} + \cdots + h_1X + h_0$ then

$$\begin{aligned} h(X) &= \left(\sum_{i=0}^{n-1} f_i X^i\right)\left(\sum_{j=0}^{n-1} g_j X^j\right) \\ &= \sum_{i=0}^{n-1}\sum_{j=0}^{i} f_j g_{i-j} X^i + \sum_{i=1}^{n-1}\sum_{j=0}^{n-1-i} f_{i+j} g_{n-1-j} X^{n-1+i} \\ &= \sum_{i=0}^{n-1} X^i \sum_{j=0}^{i} f_j g_{i-j} + \sum_{i=1}^{n-1} X^{n-1+i} \sum_{j=0}^{n-1-i} f_{i+j} g_{n-1-j} \end{aligned}$$

Based on the assumption that $|f_i| < B$ and $|g_i| < B$ we have

1. for all i from 0 to $n-1$ we have

$$|h_i| = \left|\sum_{j=0}^{i} f_j g_{i-j}\right| \leq \sum_{j=0}^{i} |f_j||g_{i-j}| < \sum_{j=0}^{i} B^2 = (i+1)B^2,$$

2. for all i from 1 to $n-1$ we have

$$|h_{n-1+i}| = \left|\sum_{j=0}^{n-1-i} f_{i+j} g_{n-1-j}\right| \leq \sum_{j=0}^{n-1-i} |f_{i+j}||g_{n-1-j}| < \sum_{j=0}^{n-1-i} B^2 = (n-i)B^2.$$

It means that $|h_i| < nB^2$ for all i from 0 to $2n-2$. If $M > 2nB^2$, then all coefficients (represented as in Definition 0) of $f(X)$, $g(X)$ and $h(X)$ can be represented in residue system modulo M without reduction. This leads to the formula $f(X)g(X) \bmod M = f(X)g(X)$ and ends proof. □

Theorem 1.3 *Let $f(X) = f_{n-1}X^{n-1} + \cdots + f_1 X + f_0$, $g(X) = g_{n-1}X^{n-1} + \cdots + g_1 X + g_0$ be polynomials with integer coefficients such that $|f_i| < B$ and $|g_i| < B$. If prime numbers p_i satisfy the following conditions:*

- $p_i \neq p_j$,
- $M = \prod_{i=1}^{k} p_i$,
- $2nB^2 < \prod p_i = M$,
- $p_i = 2^{m+1} r_i + 1$ *for some* $2^{m+1} \geq 2n$ *and* $r_i \in \mathbb{Z}$,

then

$$f(X)g(X) = f(X)g(X) \bmod M = (f(X) \bmod M)(g(X) \bmod M) \bmod M$$

and fields $\mathbb{F}_{p_i}$ can be used to parallel multiplication of polynomials f and g with FFT method.

Proof: Since operation mod M is a natural homomorphism of $\mathbb{Z}$ then we have

$$(f(X) \bmod M)(g(X) \bmod M) \bmod M = f(X)g(X) \bmod M$$

Based on Lemma 1.2 we achieve the second equality

$$f(X)g(X) \bmod M = f(X)g(X).$$

It means that the multiplication of $g(X)$, $f(X) \in \mathbb{Z}[X]$ gives the same result as the multiplication of $g(X) \bmod M$, $f(X) \bmod M \in (\mathbb{Z}/M\mathbb{Z})[X]$ if elements of ring $\mathbb{Z}/M\mathbb{Z}$ are represented by $\{-\frac{M-1}{2}, \ldots, -1, 0, 1, \ldots, \frac{M-1}{2}\}$. But M is a product of different primes p_i and the Chinese Remainder Theorem implies the following isomorphism:

$$\mathbb{Z}/M\mathbb{Z} \simeq \mathbb{F}_{p_1} \times \cdots \times \mathbb{F}_{p_k}.$$

It is clear that the above isomorphism can be extended to isomorphism of polynomial rings, more precisely we have:

$$(\mathbb{Z}/M\mathbb{Z})[X] \simeq \mathbb{F}_{p_1}[X] \times \cdots \times \mathbb{F}_{p_k}[X].$$

It means that multiplications in $(\mathbb{Z}/M\mathbb{Z})[X]$ can be distributed between k independent rings $\mathbb{F}_{p_i}[X]$ and each such multiplication can be done independently in parallel. Moreover all prime numbers $p_i = 2^{m+1}r_i + 1$ were chosen in the way to be well suited for FFT because each field $\mathbb{F}_{p_i}$ contains primitive root of unity of degree 2^{m+1}. □

In practice it is quite easy to find primes satisfying the assumptions of Theorem 1.3. For example, there exist 56 primes of the form

$$p_i = r_i \cdot 2^{22} + 1,$$

where $512 < r_i < 1024$. This set of primes allows us to multiply polynomials for which

- $\deg f + \deg g < 2^{22}$,
- $\max\{|f_i|, |g_i|\} \leq 2^{871}$.

If we want to use the proposed algorithm to multiply polynomials with larger degrees and coefficients then we can use 64-bit primes. For example the following set

$$\mathcal{P}_{64} = \{p_i : p_i \in \mathcal{P}, p_i = r_i \cdot 2^{32} + 2^{63} + 1, 1 < r_i < 2^{31}\}$$

can be used to multiply polynomials for which

- $\deg f + \deg g < 2^{32}$,
- $\max\{|f_i|, |g_i|\} < 2^{6099510377}$.

Suppose now that we have k prime numbers p_i that have the same bit length and satisfy the conditions described in Theorem 1.3. We have the following theorem:

Theorem 1.4 *If $\lfloor \log_2(p_i) \rfloor = \lfloor \log_2(p_j) \rfloor$ and formal power series have precision n, then the multiplication algorithm described in Theorem 1.3 consists of*

$$c_1 k^2 n + kn(2 + 3\log(n)) + c_2 k^2 n$$

multiplications in $\mathbb{F}_{p_i}$. Where c_1, c_2 are some constants.

Proof: Since $\lfloor \log_2(p_i) \rfloor = \lfloor \log_2(p_j) \rfloor$ for each i, j, then we can assume that the cost of multiplication in every $\mathbb{F}_{p_i}$ is the same. Single FFT multiplication consists of three basic steps:

1. Reduction modulo every chosen prime requires $c_1 k^2 n$ multiplications in $\mathbb{F}_{p_i}$. Each coefficient can be reduced modulo p_i using $c_1 k$ multiplications in $\mathbb{F}_{p_i}$. We have n coefficients and k small primes. It means that the total cost of this step is equal to $c_1 k \cdot n \cdot k = c_1 k^2 n$.
2. We perform the FFT multiplication for all $i \in \{1, \ldots, k\}$:
   (a) Fourier transform of two power series with n coefficients requiring $2n \log(n)$ multiplications in $\mathbb{F}_{p_i}$,
   (b) scalar multiplication of two vectors with $2n$ coefficients, which requires $2n$ multiplications in $\mathbb{F}_{p_i}$,
   (c) inverse Fourier transform of the vector to the power series with $2n$ coefficients requiring $n \log(n)$ multiplications $\mathbb{F}_{p_i}$.
3. Application of the Chinese Remainder Theorem to get back final coefficients requires $c_2 k^2 n$ multiplications in $\mathbb{F}_{p_i}$. Each solution of the system $x \equiv a_i \bmod p_i$ can be reconstructed using $c_2 k^2$ multiplications in $\mathbb{F}_{p_i}$. Since we have to reconstruct n coefficients, the total cost is equal to $c_2 k^2 \cdot n = c_2 k^2 n$.

Thus the multiplication algorithm described in Theorem 1.3 consists of

$$c_1 k^2 n + kn(2 + 3\log(n)) + c_2 k^2 n$$

multiplications in $\mathbb{F}_{p_i}$. □

Finally, let us see how the new algorithm compares with the method using the Fast Fourier Transform for multiplying both: polynomials and coefficients. If we assume that numbers p_i are comprised within a single register of the processor, then the complexity of the algorithm which multiplies the polynomial and its coefficients using FFT is

$$O((n \log n)(k \log k)).$$

The complexity of our algorithm is equal to

$$O(kn \log n + k^2 n).$$

If we assume that $k = O(n)$, it is clear that the algorithm based totally on FFT is much faster. Its complexity is equal to $O(n^2 \log^2 n)$, whereas our algorithm works in time $O(n^3)$. But what happens when the polynomial coefficients are reduced? Let us assume that $k = O(\log n)$. Under this assumption, the complexity of the algorithm based totally on FFT is $O(n \log^2 n \log\log n)$, whereas the asymptotic complexity of our method is $O(n \log^2 n)$. Although the difference is not significant, we definitely managed to achieve our goal which was to develop an effective algorithm for multiplying polynomials with coefficients of an order much lower than the degree.

Table 1 Multiplication of two polynomials of degree $n/2-1$ with coefficients less than 2^{256}

Polynomial degree	FFT $\mathbb{F}_{p_{544}}$ (1 core)	FFT-CRT $\bigotimes_{i=1}^{18}\mathbb{F}_{p_i}$ (1 core)	FFT-CRT $\bigotimes_{i=1}^{18}\mathbb{F}_{p_i}$ (4 cores)		
$n/2-1$	$T_1(s)$	$T_2(s)$	$T_3(s)$	T_1/T_2	T_2/T_3
511	0.0423	0.0183	0.0052	2.3	3.5
1023	0.0930	0.0383	0.0111	2.4	3.4
2047	0.2020	0.0803	0.0259	2.5	3.1
4095	0.4360	0.1705	0.0481	2.6	3.5
8191	0.9370	0.3575	0.1012	2.6	3.5
16383	2.0100	0.7444	0.2161	2.7	3.4
32767	4.2700	1.5491	0.4283	2.8	3.6
65535	9.0700	3.2168	0.9339	2.8	3.4
131071	19.1700	6.6716	1.8919	2.9	3.5

Corollary 1.5 *If $k = O(\log n)$, the complexity of the proposed algorithm is lower than the complexity of the multiplication algorithm based on FFT only, and equals to*

$$O(n \log^2 n),$$

whereas the complexity of the FFT-based algorithm is

$$O(n \log^2 n \log\log n).$$

However, in practice we managed to achieve much more than this. The numerical experiments showed that the new algorithm brings obvious benefits already in the case of polynomial coefficients consisting of several hundred bits. It means that its application is effective already for small values of k and n.

4 Results of Practical Implementation for 32-bit Processors

The implementation of the fast algorithm for multiplying polynomials has been prepared for 32-bit architecture with the use of OpenMP interface. The obtained time results turned out exceptionally good. They confirmed that in practice, the combination of the Fast Fourier Transform with the Chinese Remainder Theorem considerably accelerates computations. Tables 1 and 2 present the performance times of the algorithm for multiplying polynomials of the same degree with coefficients ranging from $[0, 2^{256})$ and $[0, 2^{512})$.

Our 32-bit implementation is fully compatible with ANSI C99 and was not optimised against any special architecture. The numerical experiments were done on Intel Core 2 processor (2.4 GHz) and confirmed that the simultaneous application

Table 2 Multiplication of two polynomials of degree $n/2-1$ with coefficients less than 2^{512}

Polynomial degree	FFT $\mathbb{F}_{p_{1088}}$ (1 core)	FFT-CRT $\bigotimes_{i=1}^{36}\mathbb{F}_{p_i}$ (1 core)	FFT-CRT $\bigotimes_{i=1}^{36}\mathbb{F}_{p_i}$ (4 cores)		
$n/2-1$	$T_1(s)$	$T_2(s)$	$T_3(s)$	T_1/T_2	T_2/T_3
511	0.1598	0.0511	0.0136	3.1	3.7
1023	0.3500	0.1055	0.0280	3.3	3.8
2047	0.7600	0.2203	0.0608	3.4	3.6
4095	1.6420	0.4562	0.1210	3.6	3.8
8191	3.5310	0.9430	0.2527	3.7	3.7
16383	7.5500	1.9412	0.5254	3.9	3.7
32767	16.0900	3.9944	1.0960	4.0	3.6
65535	34.1300	8.2184	2.1926	4.1	3.7
131071	72.2100	16.9245	4.5895	4.3	3.7

of CRT and FFT is very efficient. To the end of this section we will assume that: $p_{544} = 2^{544} - 2^{32} + 1$, $p_{1088} = 2^{1088} - 2^{416} + 2^{256} + 1$ and $2^{31} < p_i < 2^{32}$. We compare our FFT-CRT based implementation with multiplication algorithm based on FFT over fields $\mathbb{F}_{p_{544}}$ and $\mathbb{F}_{p_{1088}}$.

We use OpenMP standard to implement parallel version of the proposed algorithm. In Tables 1 and 2 fraction T_2/T_3 gives us information about how many of our 4 cores are on average used by a single multiplication. We can see that the algorithm based on the FFT and CRT uses between 80 % and 90 % computational power. It is a very good result for arithmetic algorithm.

5 Fast 64-bit Implementation and its Application to Integer Multiplication

To demonstrate the speed of our method we decided to compare it with GMP (The GNU Multiple Precision Arithmetic Library) implementation of integer multiplication. Every algorithm dedicated to multiply integer polynomials can be easily adapted to multiply integers. Suppose we have two integers $a, b \in \mathbb{Z}$ such that

$$a = a_0 + a_1 R + a_2 R^2 + \cdots + a_{n-1} R^{n-1},$$
$$b = b_0 + b_1 R + b_2 R^2 + \cdots + b_{n-1} R^{n-1}.$$

These integers can be converted to integer polynomials

Table 3 Speed comparison of our algorithm and GMP FFT integer multiplication

Factor bits	GMP-FFT	FFT-CRT			
		512-bit digits		1024-bit digits	
	1 core (s)	1 core (s)	4 cores (s)	1 core (s)	4 cores (s)
2^{21}	0.040	0.069	0.023	0.073	0.022
2^{22}	0.082	0.145	0.050	0.153	0.047
2^{23}	0.176	0.302	0.106	0.318	0.097
2^{24}	0.395	0.612	0.224	0.663	0.203
2^{25}	0.858	1.268	0.419	1.365	0.403
2^{26}	1.837	2.628	0.971	2.820	0.873

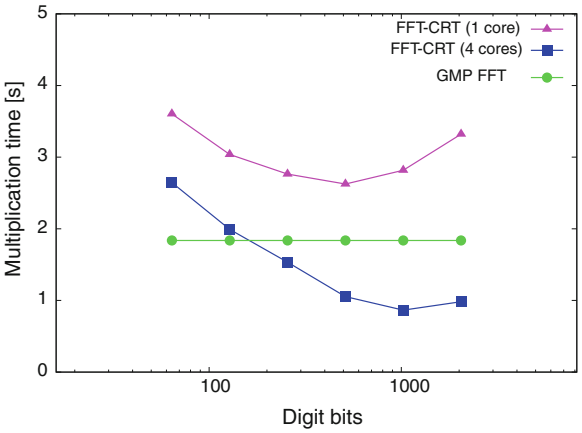

Fig. 1 Mutiplication time of two integers with 2^{26} bits each

$$A(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1},$$
$$B(X) = b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}$$

and multiplied to get polynomial $C(X) = \sum_{i=0}^{2n-2} c_i X^i$. If $nR < R^2$, then we can compute $c = ab$ as follows

$$c = \sum_{i=0}^{2n-2} (c_i \bmod R) R^i + R \sum_{i=0}^{2n-2} \left(\left\lfloor \frac{c_i}{R} \right\rfloor \bmod R \right) R^i +$$
$$R^2 \sum_{i=0}^{2n-2} \left(\left\lfloor \frac{c_i}{R^2} \right\rfloor \bmod R \right) R^i.$$

One can see that if we have coefficients c_i and R is power of 2 then c can be computed using two multiple precision additions.

To properly compare the proposed algorithm with 64-bit implementation of integer multiplication in GMP we decided to optimise our implementation against x84-64 architecture (including inline assembler functions). Numerical tests show that we can achieve the best performance for a single thread for $R = 2^{512}$. Unfortunately it is about 1.45 times slower than in GMP. We have a better situation in the case of parallel computing. If we can use 4 parallel threads, then R should be equal to 2^{1024} and our implementation is about 2 times faster than GMP which can not be run in parallel.

6 Summary

We present an analysis of a new algorithm for multiplying integer polynomials and integers. It has been designed so as to exploit fully the computing power offered by modern multicore processors. Thanks to using the Chinese Remainder Theorem, it is possible to easily allocate tasks between the available threads. Moreover, under the adopted approach there is no need to synchronize the computations and to ensure communication between individual threads, which is an additional asset. For that reason the algorithm can be easily implemented with the use of a parallel programming standard OpenMP. The ratio T_2/T_3 in Tables 1 and 2 shows how many processors out of the four ones available were used on average during a single multiplication. The measurements show that the algorithm uses from 80 to 90 % of the available computing power. In the case of an arithmetic algorithm, this should be considered a very good result. Therefore, we may conclude that the goal which consisted in designing a parallel algorithm for multiplying polynomials has been achieved.

As far as the theoretical results of the chapter are concerned, the analysis conducted in Sect. 3 and Corollary 1.5 being its essence, are of key importance. If we assume that the degree of the polynomial is n and the accuracy of its coefficients is k, then the asymptotic complexity of the proposed algorithm is

$$O(kn \log n + k^2 n).$$

Owing to the two essential components of the asymptotic function, it is impossible to determine explicitly whether the new solution is better or worse than the method based on FFT only. It is due to the fact that if we use the Fast Fourier Transform to multiply both the polynomial and its coefficients, the complexity is equal to

$$O((n \log n)(k \log k)).$$

Therefore, one can see that if $k = O(n)$, the proposed algorithm performs worse than the method based on FFT only. However, if $k = O(\log n)$, the complexity of the new algorithm is lower. The computational complexity ratio is $O(\log n)$ to the advantage of the method presented in the chapter. This reasoning allows us to conclude that the algorithm based on CRT and FFT should be used when the number of coefficients of a polynomial exceeds greatly their accuracy. This is often the case

when computations use long polynomials or power series with a modular reduction of coefficients.

The results of numerical tests presented in Sect. 4 show that the proposed method has numerous practical applications. In this section the algorithm has been intentionally compared with the implementation using the classic algorithm for multiplying coefficients in large fields $\mathbb{F}_p$. It results from the fact that in the case of numbers p consisting of 500 or 1000 bits, multiplication based on the Fourier Transform is completely ineffective. The measurement results came as a great surprise, as it turned out (Tables 1 and 2) that the proposed algorithm is several times faster even when its application is not parallel.

In Sect. 5 we prove that our algorithm can be also used to multiply integers. Table 3 and Fig. 1 show that our four-thread parallel implementation is much faster than single-thread implementation of GMP.

References

1. Charles D, Lauter K (2005) Computing modular polynomials. J Comput Math 8:195–204
2. Chmielowiec A (2012) Fast, parallel algorithm for multiplying polynomials with integer coefficients. Lecture notes in engineering and computer science: Proceedings of the world congress on engineering WCE 2012, 4–6 July 2012 UK, London, pp 1136–1140
3. Cormen TH, Leiserson CE, Rivest RL, Stein C (2003) Introduction to algorithms. MIT Press, New York
4. Crandall R, Fagin B (1994) Discrete weighted transforms and large integer arithmetic. Maths Comput 62:305–324
5. Crandall R, Pomerance C (2001) Prime Numbers— a computational perspective. Springer, New York
6. Enge A (2009) Computing modular polynomials in quasi-linear time. Math Comp 78:1809–1824
7. Garcia L (2005) Can Schönhage multiplication speed up the RSA encryption or decryption?. University of Technology, Darmstadt
8. Gorlatch S (1998) Programming with divide-and-conquer skeletons: a case study of FFT. J Supercomput 12:85–97
9. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing. Addison Wesley, London
10. Harvey D (2009) A cache-friendly truncated FFT. Theor Comput Sci 410:2649–2658
11. Knuth DE (1998) Art of computer programming. Addison-Wesley Professiona, London
12. Müller V (1995) Ein Algorithmus zur Bestimmung der Punktzahlen elliptischer Kurven über endlichen Körpern der Charakteristik grösser drei. Ph.D. Thesis, Universität des Saarlandes
13. Nussbaumer HJ (1980) Fast polynomial transform algorithms for digital convolution. IEEE Trans Acoust Speech Signal Process 28(2):205–215
14. Schönchage A (1982) Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. Lecture notes in computer science, vol 144. Springer, Berlin, pp 3–15
15. Schönhage A, Strassen V (1971) Schnelle multiplikation grosser zahlen. Computing 7:281–292
16. Takahashi D, Kanada Y (2000) High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. J Supercomput 15:207–228
17. Van der Hoeven J (2004) The truncated Fourier transform and applications. ISSAC 2004 ACM, pp 290–296
18. Van der Hoeven J (2005) Notes on the truncated Fourier transform. Unpublished, avaliable on http://www.math.u-psud.fr/vdhoeven/