Multipartite Modular Multiplication

Pascal Giorgi, Laurent Imbert, Thomas Izard

Multipartite Modular Multiplication

Pascal Giorgi, Laurent Imbert, Thomas Izard

The authors are with the Laboratoire d'Informatique, de Robotique et
de Microélectronique de Montpellier (LIRMM), Université Montpellier 2, France.

September 1, 2011

Abstract

Current processors typically embeds many cores running at high speed. We may then ask ourselves whether software parallelism is practical for low level arithmetic operations. In this paper we generalize the bipartite modular multiplication method of Kaihara and Takagi. We present a multipartite multiplication. We carefully analyze its asymptotic complexity and measure its practical efficiency and scalability for nowadays multi-core architectures. We present several experimental results which illustrate the efficiency of our method and which bring a positive answer to the above question for a wide range of operand's sizes.

1 Introduction

Multiplication is certainly one of the most studied basic arithmetic operations [1] and our ability to compute a product as fast as possible, especially when the size of the operands grows, is critical for very many applications [2]. In this paper, we are interested in modular multiplication, i.e., the computation of $AB \bmod P$, where A, B are two integers less than P. Several algorithms have been proposed to perform this operation efficiently. See [3, chapter 2] for a recent, excellent survey. So-called interleaved methods perform the multiplication and the modular reduction in a digit-by-digit fashion. Interleaved algorithms are both highly regular and memory efficient. For these reasons, they are generally the preferred choice for hardware implementation. On the other hand, performing the multiplication digit-by-digit makes it impossible to use fast, sub-quadratic multiplication algorithms such as Karatsuba [4] Toom-Cook [5, 6] and the FFT-based methods [7]. When dealing with very large operands, and when memory is not a major issue, modular multiplication is generally performed using a first integer multiplication followed by a modular reduction.

Modular reduction algorithms generally fall into two distinct families depending on whether the reduction is performed from the most significant bits as in Barrett algorithm [8] or from the least significant ones as in Montgomery method [9]. In [10], Kaihara and Takagi proposed a bipartite modular multiplication algorithm which mixes the two options, where the upper and lower parts are reduced independently using different strategies. This algorithm is particularly well suited to parallel implementation and significant speedups can be obtained.

In this paper we further generalize the bipartite algorithm of Kaihara and Takagi. In Section 2, we first define two algorithms which perform a partial Montgomery reduction and a partial Barrett reduction. We prove their correctness and give their complexity. These algorithms are then used as building blocks for our multipartite multiplication presented and carefully analyzed in Section 3. In Section 4, we present some optimization strategies in the context of parallel implementation. Finally, we present timings and comparisons in Section 5.

2 Background

In the sequel, we consider multiple precision integers in radix β. Given $0 < P < \beta^n$ and $C > P$, a modular reduction algorithm is a method for computing $R = C \bmod P$, i.e. the remainder of the euclidean division of C by P, such that

$$R = C - \left\lfloor \frac{C}{P} \right\rfloor P. \tag{1}$$

Note that in practice, C is often the result of an integer multiplication between operands of size n (in radix β). In Sections 2.3 and 3 we consider algorithms for modular multiplication which compute $AB \bmod P$.

In general, performing the exact division to evaluate the quotient $\lfloor C/P \rfloor$ in (1) is very expensive and should be avoided. Several approaches have been proposed to bypass this operation by only computing an approximation of the quotient. As a result, the remainder might not be fully reduced (i.e. less than P) and a few number of final reduction steps (subtractions) may be needed. However, in many cases it is sufficient to work with partial results that are not fully reduced. The most widely used modular reduction algorithms are due to Barrett [8] and Montgomery [9]. Given an input integer $C > P$, both algorithms consist of adding or subtracting a suitable multiple of P from C to get a value which belongs to the same class of congruence as C modulo P and has fewer significant digits. By computing $R = C - QP$, where Q is an approximation of the quotient $\lfloor C/P \rfloor$, Barrett's algorithm reduces the input C from the most significant digits. Clearly, the result is congruent to C modulo P. The number of significant digits of $R < C$ depends on the quality of the approximation of the quotient. Symmetrically, Montgomery's algorithm performs this reduction from the least significant digits by adding to C a suitable multiple of P such that the m least significant digits of the result are all zeros. Therefore, a division by β^m (which reduces to simple shifts to the right) gives a value that is not exactly congruent to C modulo P, but rather to $C\beta^{-m} \bmod P$. (This extra factor β^{-m} introduced by Montgomery's reduction algorithm is easy to deal with in the context of a modular exponentiation, and to remove when necessary.) The bipartite modular multiplication algorithm proposed by Kaihara and Takagi in [10] simply aims at reducing the input C from both sides at the same time, in parallel. Its implementation requires a reduction algorithm from the most significant digits, together with a reduction algorithm such as Montgomery which operates from the least significant digits. In the next paragraphs, we present the algorithms of Montgomery, Barrett and the bipartite algorithm and we express their complexity in terms of the number of integer multiplications. Following [3] we use $M(m, n)$ to denote the time to perform an integer multiplication with operands of size m and n respectively and we simply note $M(n)$ when both operands have the same size.

2.1 Montgomery Reduction Algorithm

Montgomery's idea [9] was to trade a costly division for only two integer multiplications plus some right shifts. Given $0 < P < \beta^n$ and $0 \leq C < P^2$, the algorithm computes the smallest integer Q such that $C + QP$ is a multiple of β^n. Hence $(C + QP)/\beta^n$ is an integer less than $2P$ and congruent to $C\beta^{-n}$ modulo P. If $(P, \beta) = 1$, the value Q is easily obtained as $Q = \mu C \bmod \beta^n$, where $\mu = -P^{-1} \bmod \beta^n$ is a precomputed value[1]. A detailed description is given in [3]. We note that the products to compute μC and QP can be performed using a low short product and a high short product respectively. However, if one want to benefit from sub-quadratic multiplication algorithms, full products must be computed. Therefore the cost of Montgomery modular reduction is $2M(n)$. This is equivalent to $6M(n/2)$ with Karatsuba multiplication.

In the following, we will need a generalization of Montgomery reduction algorithm which only performs a partial reduction by computing a remainder $R \equiv C\beta^{-t} \pmod P$ for some $t \leq n$ such that $0 \leq R < \beta^{m-t}$. This partial reduction algorithm is described in Algorithm 1. (The original Montgomery modular reduction algorithm can be found in [3].)

Theorem 1. *Algorithm **PMR** is correct.*

[1] Because of this rather expensive precomputation, Montgomery algorithm is interesting when several reduction modulo P need to be computed.

Algorithm 1: PMR (Partial Montgomery Reduction)

Input: $0 < P < \beta^n$ with $(P, \beta) = 1$, $0 \leq C < P^2$, $C < \beta^m$, $t \leq n$ and $\mu = -P^{-1} \bmod \beta^t$ (precomputed)
Output: $R \equiv C\beta^{-t} \pmod{P}$ with $0 \leq R < \beta^{m-t}$

1 $Q \leftarrow \mu C \bmod \beta^t$
2 $R \leftarrow (C + QP)/\beta^t$
3 **if** $R \geq \beta^{m-t}$ **then** $R \leftarrow R - P$
4 **return** R

Proof. We first prove that $R \equiv C\beta^{-t} \pmod{P}$. Since $Q = \mu C = -P^{-1}C \bmod \beta^t$ we have $C + QP \equiv 0 \pmod{\beta^t}$. Therefore, the division by β^t in step 2 is exact and the result R is congruent to $C\beta^{-t}$ modulo P. Now, since $0 \leq C < \beta^m$ and $0 \leq Q < \beta^t$, we have that $0 \leq C + QP < \beta^m + \beta^t P$. Dividing by β^t in step 2 gives $0 \leq R < \beta^{m-t} + P$. Therefore, at most one subtraction is required in step 3 to get $0 \leq R < \beta^{m-t}$. □

Complexity of PMR

Step 1 of **PMR** corresponds to the multiplication of the t least significant digits of C with the t least significant digits of the precomputed constant μ. This step costs either $M(m, t)$ when $m < t$ or $M(t)$ otherwise. Step 2 involves a multiplication between P and Q, of size n and t respectively of cost $M(n, t)$. As a result:

$$\mathrm{PMR}(t, n, m) = \begin{cases} M(m, t) + M(n, t) & \text{if } m < t \\ M(t) + M(n, t) & \text{otherwise} \end{cases} \tag{2}$$

2.2 Barrett Reduction Algorithm

Barrett algorithm [8] is an interesting alternative to Montgomery reduction since it shares the same fundamental property: it performs the modular reduction without division. The idea is to precompute an approximation of the inverse of P in order to obtain an approximation of the quotient with just one multiplication. More precisely, Barrett's algorithm computes an approximation Q of the quotient $\lfloor C/P \rfloor$ as

$$Q = \left\lfloor \left\lfloor \frac{C}{\beta^n} \right\rfloor \nu/\beta^n \right\rfloor, \tag{3}$$

where $\nu = \lfloor \beta^{2n}/P \rfloor$ is precomputed. The remainder is then obtained by computing $C - QP$, possibly followed by at most three subtractions if one wants the remainder to be fully reduced. The detailed description and proof of correctness can be found in [3]. (See also [11] for a slightly different version.) Assuming full products, the complexity of Barrett reduction is $2M(n)$.

As for Montgomery algorithm, we will need a generalization of the Barrett reduction algorithm which only computes a partially reduced remainder, i.e. an integer $R \equiv C \pmod{P}$ such that $0 \leq R < \beta^{m-t}$ with $t \leq m - n$. The idea behind this generalization is to zero-out only the first t leading digits of C. To achieve this, one only needs to compute the first t leading digits of Q as:

$$Q = \left\lfloor \frac{\left\lfloor \frac{C}{\beta^{m-t}} \right\rfloor \left\lfloor \frac{\beta^{n+t}}{P} \right\rfloor}{\beta^t} \right\rfloor \beta^{m-n-t} \tag{4}$$

Note that the constant ν is then set to $\nu = \lfloor \beta^{n+t}/P \rfloor$. Following the original Barrett reduction we compute $R = C - QP$, possibly followed by a few subtractions in order to guarantee the desired bound on the remainder. The detailed description of this partial modular reduction is given in Algorithm 2.

Algorithm 2: PBR (Partial Barrett Reduction)

Input: $\beta^{n-1} < P < \beta^n$, $0 \le C < P^2$, $C < \beta^m$, $t \le m-n$ and $\nu = \lfloor \beta^{n+t}/P \rfloor$ (precomputed)
Output: $R \equiv C \pmod P$ with $0 \le R < \beta^{m-t}$

1 $Q \leftarrow \lfloor C_1 \nu / \beta^t \rfloor \beta^{m-n-t}$ where $C = C_1 \beta^{m-t} + C_0$ with $0 \le C_0 < \beta^{m-t}$
2 $R \leftarrow C - QP$
3 **while** $R \ge \beta^{m-t}$ **do** $R \leftarrow R - \beta^{m-n-t} P$
4 **return** R

Theorem 2. *Algorithm **PBR** is correct and step 3 is performed at most twice.*

Proof. Since all operations are just adding multiples of P to C, it is clear that $R \equiv C \pmod P$. Therefore, we only need to prove that $0 \le R < \beta^{m-t}$. Since $\nu < \beta^{n+t}/P$, writing C as $C = C_1 \beta^{m-t} + C_0$ with $0 \le C_0 < \beta^{m-t}$ gives

$$Q \le \frac{C_1 \nu}{\beta^t} \beta^{m-n-t} \le \frac{C_1 \beta^{m-t}}{P} \le \frac{C}{P}$$

which implies $R = C - QP \ge 0$.

Now, given the definition of ν and Q, we have $\nu > \beta^{n+t}/P - 1$ and $Q > (C_1 \nu/\beta^t - 1)\beta^{m-n-t}$. Thus we also have $\nu P > \beta^{n+t} - P$ and $\beta^t Q > C_1 \nu \beta^{m-n-t} - \beta^{m-n}$, yielding

$$\beta^t QP > C_1 \nu \beta^{m-n-t} P - \beta^{m-n} P > \beta^t (C - C_0) - P(\beta^{m-n} + C_1 \beta^{m-n-t})$$

Since we have $C_0 < \beta^{m-t}$ and $C_1 < \beta^t$ we get

$$\beta^t QP > \beta^t C - \beta^t \beta^{m-t} - \beta^t (2\beta^{m-n-t} P)$$

which gives the following upper bound on R:

$$R = C - QP < \beta^{m-t} + 2\beta^{m-n-t} P \tag{5}$$

Finally, since $P < \beta^n$ we have $\beta^{m-n-t} P < \beta^{m-t}$. Hence, if $R \ge \beta^{m-t}$ then $R - \beta^{m-n-t} P \ge 0$. According to (5) we see that at most two subtractions are required in step 3 to guarantee that $0 \le R < \beta^{m-t}$. □

Complexity of PBR

Since $\beta^{n-1} < P < \beta^n$ we have $\beta^t < \beta^{n+t}/P < \beta^{t+1}$ and thus $\beta^t \le \nu < \beta^{t+1}$. Hence, step 1 is a multiplication between C_1 of size t and ν of size $t+1$ of asymptotic cost $M(t)$. However, if C_1 has only $s < t$ significant digits in the left-most position, i.e, $C_1 = C_1' \beta^{m-s} + C_0$ with $0 \le C_0 < \beta^{m-t}$, the cost of step 1 can be reduced to $M(s,t)$. (We shall encounter this situation in Section 3). Step 2 is a multiplication between P and Q, more exactly with the $t+1$ leading digits of Q (the other ones being all zeros). If one assumes that multiplying by powers of β is free, the cost of step 2 is $M(n,t)$. Hence the complexity:

$$\mathrm{PBR}(t,n,s) \begin{cases} M(s,t) + M(n,t) & \text{if } s < t \\ M(t) + M(n,t) & \text{otherwise} \end{cases} \tag{6}$$

2.3 Bipartite Modular Multiplication

The bipartite modular multiplication proposed in [10] by Kaihara and Takagi aims at computing $AB \bmod P$ for $0 \leq A, B < P < \beta^n$. Their idea consists of splitting one operand in two parts and to perform the resulting computations in parallel using two independent processes. Although any splitting does work, we illustrate the bipartite algorithm in the case where one operand, say B, of size n, is split in two parts of equal size $n/2$, i.e. $B = B_1\beta^{n/2} + B_0$. We have $AB \bmod P = (AB_1\beta^{n/2} \bmod P + AB_0 \bmod P) \bmod P$. In that form the sizes of the operands to be reduced are very unbalanced: $5n/2$ for $AB_1\beta^{n/2}$ versus $3n/2$ for AB_0, and then the computation of $AB_1\beta^{n/2} \bmod P$ would take much longer to complete than $AB_0 \bmod P$. Instead, Kaihara and Takagi use a Montgomery-like representation to compute

$$AB\beta^{-n/2} \bmod P = \left(AB_1 \bmod P + AB_0\beta^{-n/2} \bmod P\right) \bmod P \tag{7}$$

Now, it is easy to see that (7) can be computed with one call to **PBR** for $AB_1 \bmod P$ and one call to **PMR** for $AB_0\beta^{-n/2} \bmod P$, and that these two computations are independent. (Note that in the original bipartite algorithm, $AB_1 \bmod P$ is computed using a classical interleaved reduction algorithm.) Both **PBR** and **PMR** reduce operands from size $3n/2$ to size n. The complexity of the bipartite modular multiplication is easily deduced. The two partial products AB_0 and AB_1 cost $M(n, n/2)$ each. Adding the costs of **PMR** and **PBR** with $t = n/2$ from (2) and (6) respectively yields a total cost of $2M\left(n/2\right) + 4M\left(n, n/2\right)$.

3 Multipartite Modular Multiplication

In this section we introduce a generalization of the bipartite modular multiplication, which allows to divide the computation into an arbitrary number of independent operations, with operands of smaller sizes, therefore minimizing the overall complexity. Let $0 \leq A, B < P < \beta^n$. All the algorithms described in the next sections compute $AB\beta^{-n/2} \bmod P$.

3.1 Introduction

Let us start with a basic extension of the bipartite algorithm. Instead of splitting only one operand in two parts (of equal size), a straightforward extension is to split both operands in two parts each. The so-called quadripartite multiplication algorithm then consists of computing

$$\begin{aligned} AB\beta^{-n/2} \bmod P &= (A_1\beta^{n/2} + A_0)(B_1\beta^{n/2} + B_0)\beta^{-n/2} \bmod P \\ &= (A_1B_1\beta^{n/2} + A_1B_0 + A_0B_1 + A_0B_0\beta^{-n/2}) \bmod P \end{aligned}$$

This quadripartite multiplication requires four independent products of complexity $M(n/2)$ each, plus two modular reductions:

- from (2), the low product $A_0B_0\beta^{-n/2} \bmod P$ is computed using **PMR** with $t = n/2$, at a cost of $M(n/2) + M(n, n/2)$.
- From (6), the high product $A_1B_1\beta^{n/2}$ of size $3n/2$ is reduced modulo P using **PBR** with $t = n/2$, at a cost of $M(n/2) + M(n, n/2)$.
- The middle products A_0B_1 and A_1B_0, which costs $M(n/2)$ each, need not be reduced modulo P. Indeed, since $A < P = P_1\beta^{n/2} + P_0$, we have $A_1\beta^{n/2} \leq P_1\beta^{n/2} < P$, since P is odd. With $B_0 < \beta^{n/2}$, we get $A_1B_0 < A_1\beta^{n/2} < P$. Clearly, the same reasoning yields $A_0B_1 < P$.

We remark that computing the two middle products takes $2M(n/2)$, which is less than the time required to compute the high or the low products with reduction. The quadripartite multiplication can then be computed on a parallel architecture in $2M(n/2) + M(n, n/2)$ multiplications using three arithmetic units. The four partial products, each of size n, are added together to get the final result less than $4P$. (Note that

the addition of the two middle products and the potential subtraction if their sum is greater than P can be completed before the end of the other two processes. Hence, in the same time, one gets a result that is less than $3P$.)

3.2 Generalization

Let us now consider a full generalization of the above algorithm, where the operands are divided into k parts each. We have $A = \sum_{i=0}^{k-1} A_i \beta^{ni/k}$ and $B = \sum_{i=0}^{k-1} B_i \beta^{ni/k}$. Hence the modular product shifted to the right by $n/2$ digits rewrites:

$$AB\beta^{-n/2} \bmod P = \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-1} A_i B_j \beta^{d_{i,j}} \bmod P \right) \bmod P, \tag{8}$$

with $d_{i,j} = n(i+j)/k - n/2$. (Note that $3n/2 - 2n/k \leq d_{i,j} \leq -n/2$).

As in the quadripartite version, some partial products $A_i B_j \beta^{d_{i,j}}$ will have to be reduced modulo P using either **PMR** or **PBR** depending on their weight, whereas some others will not require any reduction. More exactly, if $d_{i,j} < 0$, the products $A_i B_j$ are reduced with **PMR** yielding $A_i B_j \beta^{d_i,j} \bmod P$. Similarly, if $d_{i,j} > n - 2n/k$, the terms $A_i B_j \beta^{d_{i,j}} \bmod P$ are computed with **PBR**. And for $0 \leq d_{i,j} \leq n - 2n/k$, we have $A_i B_j \beta^{d_{i,j}} < \beta^n$ and no further reduction is necessary after the multiplication $A_i B_j$. In Figure 1, we illustrate the case where both operands are divided in $k = 5$ parts. The final result $AB\beta^{-n/2}$ is then obtained by adding modulo P all these reduced partial products together, which can also be done in parallel.

3.3 Complexity analysis

In order to analyze the complexity of the multipartite multiplication algorithm, we need to know how many calls to **PMR** (resp. **PBR**) are required as a function of k, as well as the exact number of significant digits of the operands to be reduced.

Lemma 1. *If the operands of the multipartite modular multiplication are both decomposed in $k > 0$ blocks, the number of calls to **PMR** is equal to $k(k+2)/8$ if k is even and $(k+1)(k+3)/8$ if k is odd.*

Proof. As explained above, **PMR** is used when $d_{i,j} < 0$, which is equivalent to $0 \leq i+j < k/2$. Since $i+j$ is an integer, this is equivalent to $0 \leq i+j \leq \lceil k/2 \rceil - 1$. Now, for every value $i+j$, there are exactly $i+j+1$ partial products $A_i B_j$. Therefore, the number of calls to **PMR** is equal to $1+2+\cdots+\lceil k/2 \rceil = \lceil k/2 \rceil (\lceil k/2 \rceil + 1)/2$. Replacing $\lceil k/2 \rceil$ by $k/2$ when k is even and $(k+1)/2$ when k is odd concludes the proof. □

Lemma 2. *If the operands of the multipartite modular multiplication are both decomposed in $k > 0$ blocks, the number of calls to **PBR** is equal to $k(k+2)/8$ if k is even and $(k+1)(k+3)/8$ if k is odd.*

Proof. As explained above, **PBR** is used whenever $d_{i,j} > n - 2n/k$, which is equivalent to $i + j > 3k/2 - 2$. Since $i + j$ is an integer, this is equivalent to $\lfloor 3k/2 \rfloor - 1 \leq i + j \leq 2k - 2$. Note that there are exactly $2k-2-(i+j)+1$ partial products for every value $i+j$. Therefore the number of partial products $A_i B_j$ that need to be reduced with **PBR** is also equal to $1+2+\cdots+\lceil k/2 \rceil = \lceil k/2 \rceil (\lceil k/2 \rceil + 1)/2$. In fact, the algorithm is perfectly symmetrical: if $A_i B_j$ is reduced with **PMR**, then for all i', j' such that $i' + j' = 2k - i - j$, the partial product $A_{i'} B_{j'}$ is reduced with **PBR**. □

Corollary 1. *The number of partial products that do not need to be reduced is $(3k^2 - 2k)/4$ if k is even and $(3k^2 - 4k - 3)/4$ is k is odd.*

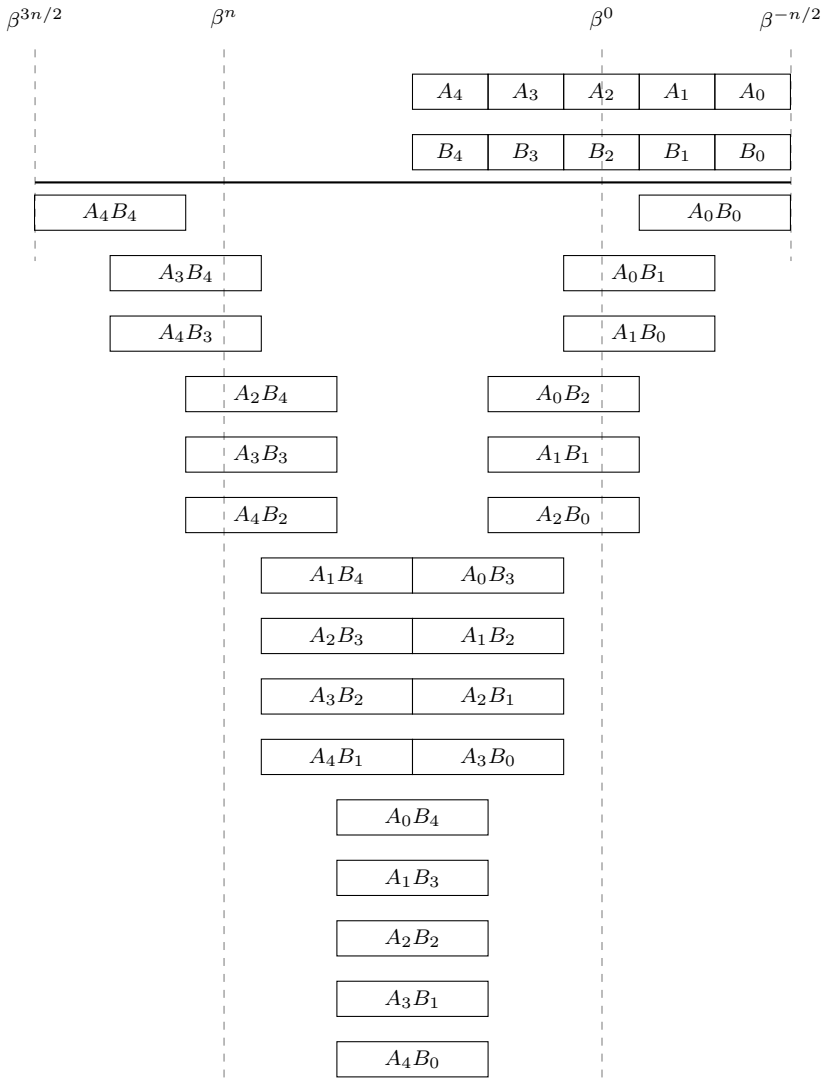

Figure 1: Multipartite multiplication with $k = 5$: the partial product A_0B_0, A_0B_1, A_1B_0, A_0B_2, A_1B_1 and A_2B_0 are all reduced modulo P using **PMR** with t equals respectively to : $n/2$, $3n/10$, $3n/10$, $n/10$, $n/10$, $n/10$ (note that since A_0B_0 has only $m = 2n/k < n/2$ digits, the cost of this **PMR** is reduced as stated in (2)). Symmetrically, A_4B_4, A_3B_4, A_4B_3, A_2B_4, A_3B_3 and A_4B_2 are reduced with **PBR** with t equals respectively to : $n/2$, $3n/10$, $3n/10$, $n/10$, $n/10$, $n/10$ (as explained in (6), the cost for A_4B_4 is reduced since A_4B_4 has only $s = 2n/k < n/2$ significant digits. All the other middle products need not be reduced.

Proof. Simply note that the number of such partial products is equal to k^2 minus the number of partial products that have to be reduced using either **PMR** or **PBR**. □

Theorem 3. *The total cost of the multipartite modular multiplication is less than*

$$k^2 M\left(\frac{n}{k}\right) + 2\sum_{i+j=0}^{\lceil \frac{k}{2} \rceil - 1} \big(M(t_{i,j}) + M(n, t_{i,j})\big), \quad \textit{where } t_{i,j} = \frac{n}{2} - \frac{n(i+j)}{k} \tag{9}$$

Proof. First, notice that the term $k^2 M(n/k)$ corresponds to the k^2 products $A_i B_j$ with operands of size n/k. All the other multiplications come from the calls to **PMR** and **PBR**, with operands' sizes depending on $i+j$. Let us first analyze the complexity resulting from the calls to **PMR**. As seen in the proof of Lemma 1, for $0 \leq i+j \leq \lceil k/2 \rceil - 1$, our algorithm performs a partial Montgomery reduction. More specifically, it partially reduces modulo P the value $A_i B_j \beta^{d_{i,j}}$, of size $m_{i,j} = n(i+j+2)/k$, in order to obtain $A_i B_j \beta^{-t_{i,j}}$ as explained in Algorithm 1. Hence, we have $t_{i,j} = -d_{i,j} = n/2 - n(i+j)/k$. Therefore, following (2), each call to **PMR** costs $\mathrm{PMR}(t_{i,j}, n, m_{i,j})$, which is at most $M(t_{i,j}) + M(n, t_{i,j})$.

Symmetrically, for $\lfloor 3k/2 \rfloor - 1 \leq i+j \leq 2k-2$, our algorithm performs a partial Barrett reduction (see proof of Lemma 2). More specifically, it partially reduces modulo P the operand $A_i B_j \beta^{d_{i,j}}$, of size $n(i+j+2)/k$ digits among which only the $s_{i,j} = 2n/k$ are non-zero, in order to get a value less than $\beta^{3n/2}$. Hence, following (6), each call to **PBR** costs $\mathrm{PBR}(n(i+j+2)/k - 3n/2, n, 2n/k)$. The total cost resulting from all the calls to **PBR** is then less than

$$\begin{aligned}
&= \sum_{i+j=\lfloor \frac{3k}{2} \rfloor - 1}^{2k-2} M\left(\frac{n(i+j+2)}{k} - \frac{3n}{2}\right) + M\left(n, \frac{n(i+j+2)}{k} - \frac{3n}{2}\right) \\
&= \sum_{2k-2-\lfloor \frac{3k}{2} \rfloor + 1}^{0} M\left(\frac{n(2k-i-j)}{k} - \frac{3n}{2}\right) + M\left(n, \frac{n(2k-i-j)}{k} - \frac{3n}{2}\right) \\
&= \sum_{i+j=0}^{\lceil \frac{k}{2} \rceil - 1} M\left(\frac{n}{2} - \frac{n(i+j)}{k}\right) + M\left(n, \frac{n}{2} - \frac{n(i+j)}{k}\right) \\
&= \sum_{i+j=0}^{\lceil \frac{k}{2} \rceil - 1} M\left(t_{i,j}\right) + M(n, t_{i,j})
\end{aligned}$$

which concludes the proof. (The exact complexity taking into account the cases $m < t$ in **PMR** and $s < t$ in **PBR** is given in Section A.) □

3.4 Reduction of the number of PMR and PBR

Intuitively, the multipartite multiplication exhibits k^2 partial products among which only a certain number need to be reduced modulo P by using either **PMR** or **PBR**. However, one may remark that some of these computations are redundant. Indeed, for each $A_i B_j$ that need to be reduced modulo P, a Q-value is computed to zero-out a part of the partial product (the least significant bit for **PMR** and the most significant bits for **PBR**). Note that some of these $A_i B_j$ have the same weight and the corresponding Q-value are computed to zero-out exactly the same part. Therefore one may prefer to add all these partial products to perform only one reduction modulo P. This situation is illustrated in Figure 2 where the number of **PMR** and **PBR** is reduced from 6 to 4 when the operands are split into $k = 3$ chunks each.

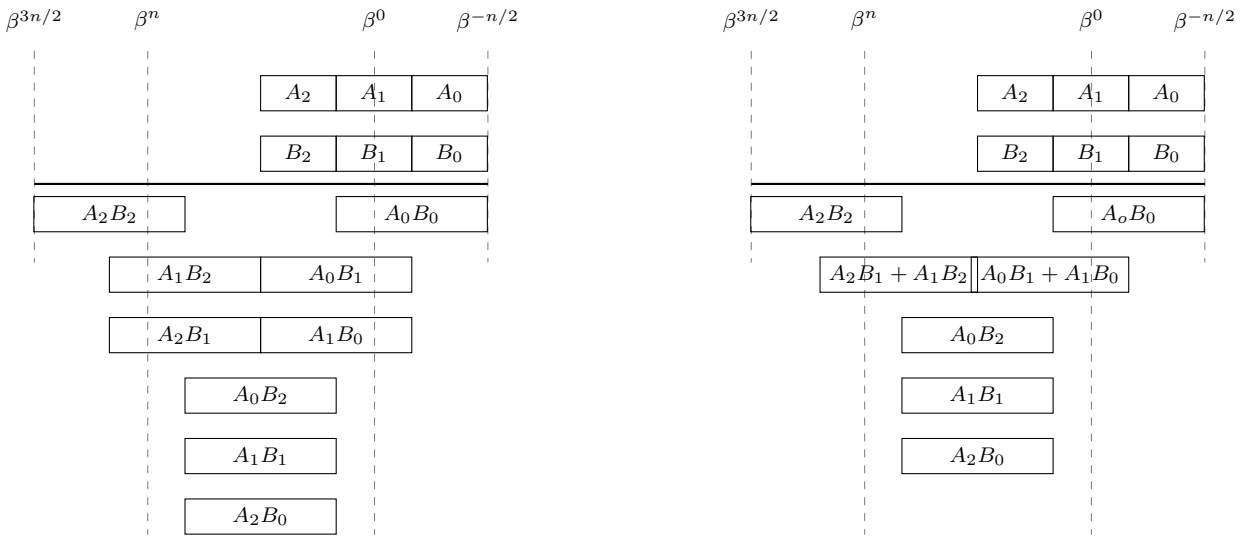

Figure 2: Reducing the number of call to **PMR** and **PBR** in the multipartite algorithm (with $k = 3$) by adding some of the partial products of identical weight

The idea illustrated in Figure 2 is to sum up every partial product A_iB_j with the same weight that have to be reduced with either **PMR** or **PBR**. Using this trick, it is easy to see that the number of call to **PMR** and **PBR** is reduced. The following lemma gives the precise number of **PMR** and **PBR** that have to be performed.

Lemma 3. *Assuming all partial products with the same weight involved in either **PMR** or **PBR** are added together, the number of call to **PMR** is $\lceil k/2 \rceil$ and the number of call to **PBR** is $\lceil k/2 \rceil$.*

Proof. The number of call to **PMR** corresponds exactly to the number of integers $i \geq 0$ such that $\beta^0 > \beta^{i(n/k)-n/2} \geq \beta^{-n/2}$ which is equivalent to $k/2 > i \geq 0$. It is of course immediate that the number of **PMR** is equal to $\lceil k/2 \rceil$. Symmetrically, the number of call to **PBR** correspond exactly to the number of integers $i \geq 0$ such that $\beta^{3n/2-2n/k} \geq \beta^{i(n/k)-n/2} > \beta^{n-2n/k}$ which is equivalent to $2k-2 \geq i > 3k/2-2$. It is again immediate that the number of **PBR** is equal to $\lceil k/2 \rceil$. □

The main advantage of reducing the number of **PMR** and **PBR** is that it allows to trade some of the multiplications for some additions. Following Lemma 3, one can reduce the complexity of the multipartite algorithm to a total cost of less than

$$k^2 M\left(\frac{n}{k}\right) + 2 \sum_{\ell=0}^{\lceil \frac{k}{2} \rceil - 1} \left(M(t_\ell) + M(n, t_\ell)\right), \quad \text{where } t_\ell = \frac{n}{2} - \frac{n\ell}{k} \tag{10}$$

The above cost is immediate from the proof of Theorem 3. It suffices to remark that the **PMR** (respectively **PBR**) for all partial product A_iB_j for a given $\ell = i + j$ are replaced by only one **PMR** (respectively **PBR**) using the operand $\sum_{i+j=\ell} A_iB_j$.

4 Analysis of Parallel Algorithms

Integer multiplication can be achieved in parallel in time $O(\log n)$ using $O(n \log n \log\log n)$ processors. This result comes from the parallelization of the Schönhage-Strassen Algorithm [7]. It is clear that this result also

holds for integer modular multiplication whether one uses Montgomery or Barrett modular multiplication algorithms. Nevertheless, this theoretical result assumes that a large number of processor is available which is not the reality of nowadays computers having between 2 and 12 processors per chip. Hence, a natural question arises: what complexity can we get with a fixed number of processors? In the following we discuss this question by embedding parallelism into the algorithms presented in the previous sections. We discuss two models. One will only consider a coarse grain parallelism, meaning that we only authorize parallelism at the modular arithmetic level. The second will authorize parallelism at every levels and especially at the integer arithmetic one. Readers shall keep in mind that our discussion consider algorithms with sub-quadratic complexity.

4.1 Parallelism at the modular arithmetic level

We consider parallelism only at the level of modular multiplication without using any parallelism for the underlying integer arithmetic. In this model, one of the benefits of the bipartite algorithm [10] has been to bring intrinsic parallelism within modular multiplication algorithm. Indeed, before [10], all sub-quadratic modular multiplication algorithms suffer from a sequential dependency of the integer multiplications. Barrett or Montgomery algorithms require three sequential calls to integer multiplication, and no parallelism can be achieved at the modular arithmetic level. The bipartite algorithm described in section 2.3 allows to split this sequentiality and exhibits a natural 2-ways parallelism since the calls to **PMR** and **PBR** can be performed in parallel. If we assume that two processors are available, the cost of the bipartite algorithm is:

$$M\left(\frac{n}{2}\right) + 2M\left(n, \frac{n}{2}\right) \tag{11}$$

Even if the bipartite algorithm introduces some parallelism, each call to **PMR** and **PBR** has a sequential dependency with regards to integer multiplications. As we will in Section 5, this requires thread synchronization which impacts on practical performances.

Our multipartite algorithm described in Section 3.2 generalizes the bipartite algorithm as it induces a natural k^2-way parallelism where k is the number of chunks of each operand. The following theorem gives the parallel complexity obtained with the multipartite algorithm.

Theorem 4. *Assuming k^2 processors are available with $k > 3$, the cost of the multipartite modular multiplication is*

$$M\left(\frac{n}{k}\right) + M\left(\frac{2n}{k}, \frac{n}{2}\right) + M\left(n, \frac{n}{2}\right) \tag{12}$$

Proof. Since k^2 processors are available, all the partial products and partial reductions (when necessary) are computed in parallel, the highest cost is reached for $i+j = 0$ (or $i+j = 2k-2$), i.e. when the cost of **PMR** (resp. **PBR**) is equal to $M(m, t_{i,j}) + M(n, t_{i,j})$, with $m = 2n/k$. □

Corollary 2. *Assuming k^2 processors are available with $k = 2$ or $k = 3$, the cost of the multipartite modular multiplication is*

$$M\left(\frac{n}{k}\right) + M\left(\frac{n}{2}\right) + M\left(n, \frac{n}{2}\right) \tag{13}$$

Proof. The proof is similar to the one of Theorem 4. It is sufficient to remark that the cost of **PMR** (resp. **PBR**) is equal to $M(t_{i,j}) + M(n, t_{i,j})$, since $t_{i,j} < m = 2n/k$. □

The following lemma give the complexity estimate of the multipartite algorithm when the number of processor is not a square.

Lemma 4. *Assuming $k1 \times k2$ processors are available. Let $n/k1 + n/k2 < n/2$, the cost of the multipartite modular multiplication is*

$$M\left(\frac{n}{k1}, \frac{n}{k2}\right) + M\left(\frac{n}{k1} + \frac{n}{k2}, \frac{n}{2}\right) + M\left(n, \frac{n}{2}\right). \tag{14}$$

When $n/k1 + n/k2 >= n/2$, the cost of the multipartite modular multiplication becomes

$$M\left(\frac{n}{k1}, \frac{n}{k2}\right) + M\left(\frac{n}{2}\right) + M\left(n, \frac{n}{2}\right). \tag{15}$$

Proof. The proof is similar to the one of Theorem 4. It is sufficient to remark that one has to split each operand in $k1$ and $k2$ chunks respectively. □

One can reduce the number of processors needed to achieve the same complexity by only computing a sub-quadratic number of A_iB_j mod P using the Karatsuba [4] or Toom-Cook [5] methods. This has recently been one of the motivations of [12] which uses Karatsuba multiplication to improve hardware implementation of the bipartite algorithm. Another way to decrease the number of processors with our multipartite algorithm is to use the version given in Section 3.4. In particular, it is easy to see that this approach only needs $3k^2/4 + k/2$ processors (instead of k^2) to achieve the complexity of Theorem 4.

4.2 Parallelism at the integer arithmetic level

We now consider a more realistic model where parallelism is introduced both at the higher (aka modular arithmetic) and the underlying integer arithmetic levels. In order to still achieve a reduction to integer multiplication, our parallelization is based on a quadratic approach. One can achieve a time complexity of $M\left(\frac{n}{k1}, \frac{n}{k2}\right)$ for integer multiplication using $k1 \times k2$ processors. Note that one can achieve the same complexity with less processors by using Karatsuba or Toom-cook but at a price of a more complex parallelization. One can also think of using Karatsuba or Toom-Cook to decrease the parallel complexity for a fixed number of processors. However, to the best of our knowledge, such a result does not exist and it seems to be a hard task. In the following, our complexity estimates assume the availability of $k1 \times k2$ processors. According to our model, the parallel complexity of Barrett and Montgomery modular multiplication is then:

$$3M\left(\frac{n}{k1}, \frac{n}{k2}\right).$$

Applying the same parallelization scheme to the bipartite modular multiplication gives a parallel complexity of

$$2M\left(\frac{n}{2k1}, \frac{n}{2k2}\right) + 4M\left(\frac{n}{k1}, \frac{n}{2k2}\right)$$

However, one can decide to use the intrinsic parallelism of the bipartite algorithm when the number of processors is even. Therefore, the parallel complexity becomes

$$M\left(\frac{n}{k1}, \frac{n}{2k2}\right) + 2M\left(\frac{n}{k1}, \frac{n}{k2}\right)$$

For the multipartite algorithm, Lemma 4 gives the complexity estimate. However, this complexity only takes into account parallelism at the modular arithmetic level. In particular, one can modify the method to decrease the parallel complexity by introducing parallel integer arithmetic. Indeed, one can remark that the parallel cost given in Lemma 4 has a dominant term of $M\left(n, \frac{n}{2}\right)$ which corresponds to the cost of computing QP within the reduction of A_0B_0 (resp. $A_{k-1}B_{k-1}$). Each other calls to **PMR** or **PMR** perform a computation of QP but with smaller Q-values. Therefore, it is preferable to sum-up every Q-value computed in each **PMR** and **PBR**, and then to compute only one product QP using a parallel integer multiplication. Assuming that $k1 \times k2$ processors are available, the term $M\left(n, \frac{n}{2}\right)$ in Lemma 4 becomes $M\left(\frac{n}{k1}, \frac{n}{k2}\right)$. The following lemma holds.

Lemma 5. *Assuming $k1 \times k2$ processors are available. Let $n/k1 + n/k2 < n/2$, the cost of the multipartite modular multiplication is*

$$2M\left(\frac{n}{k1}, \frac{n}{k2}\right) + M\left(\frac{n}{k1} + \frac{n}{k2}, \frac{n}{2}\right). \tag{16}$$

When $n/k1 + n/k2 >= n/2$, the cost of the multipartite modular multiplication becomes

$$2M\left(\frac{n}{k1}, \frac{n}{k2}\right) + M\left(\frac{n}{2}\right). \tag{17}$$

Proof. The proof is similar to the one of Theorem 4. It is sufficient to remark that the sum of the Q-values computed in each **PMR** and **PBR** is less than β^n. □

Even if this lemma takes into account parallelism at every arithmetic level, it gives a pessimistic estimate of the parallel complexity since one can balance parallel computation to minimize critical path. Indeed, each parallel task has to compute one product A_iB_j, and if needed its corresponding Q to reduce the product modulo P. The tasks which do not calculate a Q-value are thus cheaper and may be gather in a single task without increasing the parallel cost. Furthermore, splitting the operands in $k1$ and $k2$ chunks each may not be the better approach to optimize load balancing. In the next section, we discuss this issue and we provide an approach to optimize the balancing of the tasks.

4.3 A parallel approach for multipartite modular multiplication

As seen in Lemma 5, one can fully exploit the intrinsic parallelism of the multipartite algorithm, but this approach leads to very unbalanced computations. The splitting of the operands has a great impact on task unbalancing (the smaller the chunks the more unbalanced will the tasks be). It is therefore crucial to choose a good splitting and a good balancing strategy to minimize the parallel cost of the multipartite method.

The next theorem shows that the unbalancing is really important and one can drastically reduce the number of processors needed by the multipartite method and then improve its complexity.

Theorem 5. *Let us consider that $M(n_1 + n_2, n) = M(n_1, n) + M(n_2, n)$. Assume $k1 \times k2$ processors are available and let $k = \frac{1}{2}k1 \times k2$ be the number of chunks of each operand with $k > 3$. The cost of the multipartite modular multiplication is*

$$M\left(\frac{2n}{k1k2}\right) + M\left(\frac{4n}{k1k2}, \frac{n}{2}\right) + M\left(\frac{n}{k1}, \frac{n}{k2}\right). \tag{18}$$

When $k = 2$ or $k = 3$ the cost becomes

$$M\left(\frac{2n}{k1k2}\right) + M\left(\frac{n}{2}\right) + M\left(\frac{n}{k1}, \frac{n}{k2}\right). \tag{19}$$

Proof. The cost given in Lemma 5 corresponds to the cost of the computation of A_0B_0 and its corresponding Q-value, followed by a parallel computation of QP where Q corresponds to the summation of all the Q-values computed by each **PMR** and **PBR**. The proof of Theorem 5 relies on the fact that when $k > 3$, during the computation of A_0B_0 and its corresponding Q-value, one can also compute:

1. $k + 1$ partial products A_iB_j,
2. $\sum_{\ell=i+j} A_iB_j$ and its corresponding Q-values for $0 < \ell \leq \lceil k/2 \rceil$.

The computation of A_0B_0 and its corresponding Q-value has a cost of $M\left(\frac{n}{k}\right) + M\left(\frac{2n}{k}, \frac{n}{2}\right)$ which is equivalent to $(k+1)M\left(\frac{n}{k}\right)$ according to the assumption on $M(n)$. Therefore point 1) is correct.

For a given ℓ such that $0 < \ell \leq \lceil k/2 \rceil$, the cost of the calculation of $\sum_{\ell=i+j} A_iB_j$ and its corresponding Q-value is less than

$$\ell M\left(\frac{n}{k}\right) + M\left(\frac{n}{2} - \ell\frac{n}{k}, \frac{2n}{k}\right)$$

which is equivalent to $(k-\ell)M\left(\frac{n}{k}\right)$. Since this value is bounded by $kM\left(\frac{n}{k}\right)$, point 2) is also correct.

The complexity of Theorem 5 is thus achieved by assigning one processor per **PMR** and **PBR**, which correspond to $2\lceil k/2 \rceil$ processors using Lemma 3. We recall that in these **PMR** and **PBR** the computations of QP are delayed at the end to only one large QP computation performed in parallel.

Therefore, it remains $k^2 - 1/2(\lceil k/2 \rceil)(\lceil k/2 \rceil + 1)$ partial products A_iB_j to compute during the same time (see Corollary 1). It is easy to see that $k-2$ processors are sufficient to handle these computations in the given time since $\forall k > 1$, we have $(k+1)(k-2) > k^2 - 2\lceil k/2 \rceil > k^2 - 1/2(\lceil k/2 \rceil)(\lceil k/2 \rceil + 1)$. Thus, one can handle the calculations of all k^2 products A_iB_j and every **PMR** and **PBR** (without QP computations) in a cost of $M\left(\frac{n}{k}\right) + M\left(\frac{2n}{k}, \frac{n}{2}\right)$ using $2\lceil k/2 \rceil + k - 2 \leq 2k$ processors. Replacing k with $\frac{1}{2}k1 \times k2$ and adding the cost of a computation of the large QP on $k1 \times k2$ processors concludes the proof for $k > 3$.

When $k = 2$ the number of A_iB_j is exactly 4. The computation of A_0B_0 and its corresponding Q-value costs exactly $2M(n/2)$. The cost is identical with A_1B_1 and its corresponding Q. These two calculations can be done on 2 processors. The remaining A_0B_1 and A_1B_0 can be computed during the same time on another processor. Therefore, three processors are sufficient which conclude the proof for $k = 2$.

Finally, when $k = 3$ the number of A_iB_j is exactly 9. The computation of A_0B_0 and its corresponding Q-value costs exactly $M(n/3) + M(n/2)$. The cost is identical with A_1B_1 and its corresponding Q-value. The computation of $A_1B_0 + A_0B_1$ and its corresponding Q-value costs exactly $2M(n/3) + M(n/6)$, which is equivalent to $M(n/3) + M(n/2)$ under our assumption on $M(n)$. The cost is identical with $A_1B_2 + A_2B_1$ and its corresponding Q-value. The remaining A_0B_2, A_1B_1, A_2B_0 can be handle in exactly $2M(n/3)$ on 2 processors. Using six processors the cost is then $M(n/3) + M(n/2)$ plus the calculation of the large QP on six processors which costs $M(n/2, n/3)$. □

Theorem 5 gives a generic strategy to optimize the load balancing of the parallel tasks involved in the multipartite modular multiplication. However, this strategy is not optimal since the tasks computing the Q-values in each **PMR** and **PBR** are still unbalanced and one can try to fill-up these gaps by some of the partial product A_iB_j to further reduce the number of processors. Moreover, Theorem 5 uses an assumption on $M(n)$ which does not reflect the reality of the integer multiplication implementation which may use quasi-linear algorithm and trade-offs to switch between different sub-quadratic methods. It is therefore quite complicated to theoretically improve the complexity of the multipartite method since too many parameters have to be considered, especially the method switching for integer multiplication implementations.

4.4 Summary of the parallel complexity of modular multiplication

The following table summarizes the parallel complexity of modular multiplication with different algorithms on a given number of processor reflecting nowadays architectures. One can see that the bipartite method is always offering the better theoretical complexity. Our multipartite algorithm has a greater complexity estimate but offers more balanced integer multiplication dependency which may have an impact in practice. Furthermore, as we will see in Section 5.3, in some special cases, we can tweak the multipartite implementation to get more balanced tasks, then achieving better complexity and practical performance.

Time complexity is a good estimate to compare parallel algorithms. However task dependencies have a great impact on practical performances. In particular, our multipartite algorithm does not have any dependencies to compute every parts of the quotient Q while Montgomery, Barrett and the bipartite algorithm have to wait until the full product AB is computed before this task can be started. In practice, this requires synchronization barriers which penalize performances as will be seen in the next section.

Table 1: Parallel complexity of modular multiplication algorithms according to the number of processors

#processors	Barret/Montgomery	Bipartite	Multipartite (Theorem 5)
2	$3M(n, n/2)$	$2M(n, n/2) + M(n/2)$	-
4	$3M(n/2)$	$2M(n/2) + M(n/2, n/4)$	$3M(n/2)$
6	$3M(n/2, n/3)$	$2M(n/2, n/3) + M(n/2, n/6)$	$M(n/2, n/3) + M(n/3) + M(n/2)$
8	$3M(n/2, n/4)$	$2M(n/2, n/4) + M(n/4)$	$3M(n/4) + M(n/2)$
12	$3M(n/3, n/4)$	$2M(n/3, n/4) + M(n/4, n/6)$	$M(n/3, n/4) + M(n/6) + M(n/2, n/3)$

5 Parallel Implementation

In order to evaluate the performance of our parallel modular multiplication algorithm, we developed a C++ library for performing integer arithmetic operations on shared memory, MIMD architecture. Our library provides functions for multiple precision integer and modular arithmetic operations, including in particular the bipartite and multipartite algorithms. We based our implementation on GMP, the GNU Multiple Precision arithmetic library [13] and the OpenMP API, a framework to design parallel codes on shared-memory architectures. In the following, we present our implementation and experimental results. All the tests are made using random data. We performed our benchmark on one node of the HPC@LR centre[2], composed of two Intel Xeon processors X5650 Westmere, having six cores each running at 2.66GHz with a L3-Cache of 12MB. Our implementation is based on GNU MP v. 5.0.2 and OpenMP and the Intel C++ compiler version 10.

5.1 Software Implementation Using OpenMP

Multi-threading is generally best adapted to computational intensive applications since dealing with software parallelism induces a cost that is only amortized when the overall amount of computation is important. This extra cost comes from thread launching (`clone()`, `fork()`), thread synchronization (`wait()`) and memories operations (placement, transfers and cache misses).

The first general rule of thumb to reduce this parallelism overhead is to run no more threads than the number of cores available of the architecture. Indeed, the latency arising from the parallel operations increases proportionately with the number of running threads. Thus, launching more thread than the number of processors increases the cost without any benefit in term of computational efficiency, except when multi-threading support is available on the targeted architecture. The latency caused by the launching of the threads can be easily amortized by computing full-parallel operations, i.e. the threads are started and stopped only once. Finally, the cost of memory operations can be reduced by paying particular attention to data alignment and placement, and by minimizing the number of allocations and frees. The architecture at the HPC@LR centre that we have been using for our benchmarks embeds two processors per node. Each processor has a 12MB L3-cache that was large enough to store all the data (operands and results of our arithmetic operations), but the communication between the two processors had to be minimized. We managed to reduce this overhead by grouping threads together as much as possible on one processor (by carefully setting the `GOMP_CPU_AFINITY` environment variable).

Our implementation is generic in both the number of threads and the operands' sizes. This model can be used for a wide class of applications requiring multiple precision computations in finite structures, in particular for asymmetric cryptosystems. Given this genericity, we define the number and size of memory blocks required for each part of the computation and we allocate them only once. This also allows us to precompute several constants which only depends of the size of operands and the number of threads.

[2]More info at `www.hpc-lr.univ-montp2.fr`

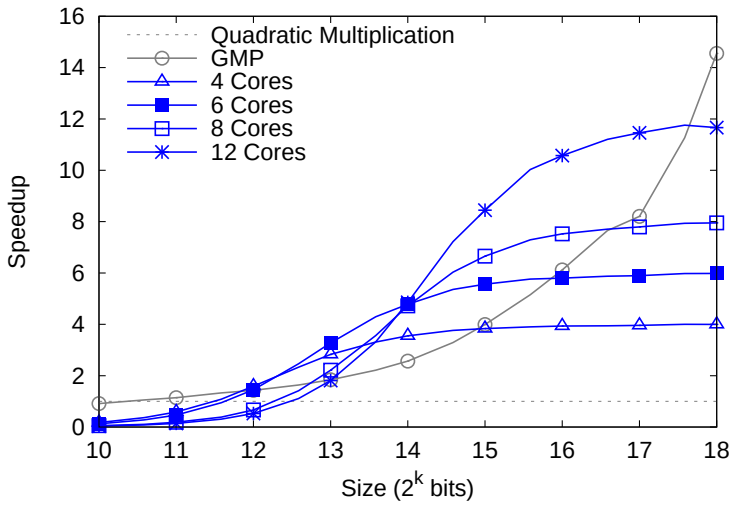

Figure 3: Performance comparison of parallel integer multiplication implementations based on GMP quadratic sequential multiplication

5.2 Parallel Integer Multiplication

As seen before, all modular multiplication algorithms heavily depends on integer multiplication, i.e multiplication in $\mathbf{Z}$. We implemented a parallel integer multiplication based on a quadratic scheme where the splitting of the operands depends on the number of processors: if $k1 \times k2$ processors are available, the operands are cut in $k1$ and $k2$ parts respectively. The number of processors can be reduced by using a sub-quadratic parallelization scheme such as Karatsuba and Toom-Cook, but this has not been implemented yet. Each thread computes a product A_iB_j, which is itself a sequential multiple precision multiplication. These computations are followed by the additions of all those partial products ; these additions are performed sequentially by one thread after a synchronization barrier. Note that one may use a parallel reduction tree to perform those additions, but in our case, the operands and number of chunks are too small to benefit from this optimization: the performances are degenerated by the synchronization barriers required by the addition tree.

Figure 3 shows the practical speedups obtained for the parallel integer multiplication when a quadratic sequential multiplication (`mpn_mul_basecase`) is used for each of the $k1 \times k2$ partial products A_iB_j. As expected, when the size of the operands increases, the overhead introduced by the parallelism (thread launching and synchronization) is totally amortized and the practical speedups match the theoretical complexity (n threads lead to a speedup of n). However, forcing quadratic sequential integer multiplication for the A_iB_j is clearly not giving the best performances, and sub-quadratic algorithms should be used for large operands. The GMP library offers five integer multiplications (basecase, Karatsuba, Toom-3, Toom-4, FFT) which are called whenever the inputs' sizes exceed some pre-defined thresholds based on the architecture. We illustrate this fact on Figure 3 by adding a curve which correspond to GMP multiplication (`mpn_mul`). In order to get the best sequential integer multiplication, our parallel implementation is based on GMP `mpn_mul` function instead of a quadratic multiplication. The speedups are shown in Figure 4.

On Figure 4, we remark that for small operand, the parallelism overhead is still present, and that the use of optimized sequential algorithms reduces the overall speedups (n threads does not lead to a speedup of n anymore, but rather to $n/2$). However, we also remark that parallelism is still worth it for a wide range of operand's sizes. We also note that the speedups of our parallel implementations decreases for inputs of size greater than 2^{17} bits, which corresponds to the `MUL_FFT_THRESHOLD` set by GMP (after tuning) for our architecture.

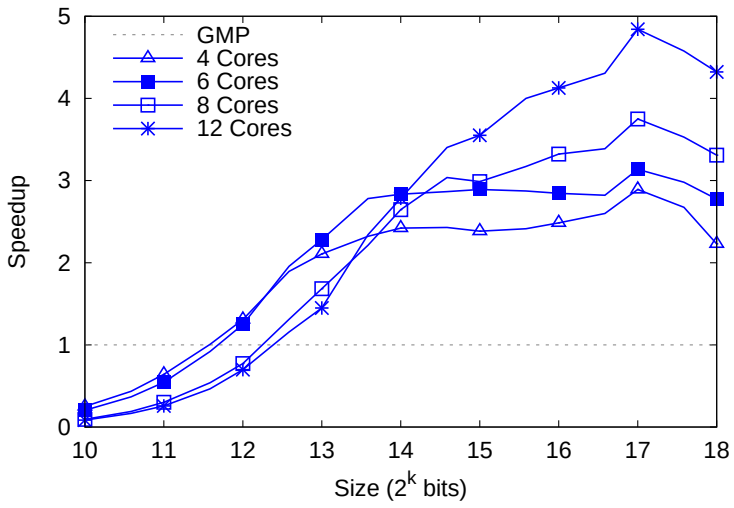

Figure 4: Performance comparison of parallel integer multiplication implementations based on GMP optimized (sub-quadratic, quasi-linear) multiplication

5.3 Parallel Modular Multiplication

In order to provide parallel implementations of Barrett, Montgomery and the bipartite algorithms, we use the parallel integer multiplication code described in the previous section. Note that for the bipartite method, we parallelize at the modular arithmetic level since the number of processors is always even. For the multipartite algorithm we optimized task scheduling based on the assumption that a quadratic sequential integer multiplication is used for the A_iB_j. This assumption is reasonable for a wide range of inputs, for example for operands of cryptographic size ranging from say 2^{10} to 2^{14} bits.

We noticed that Theorem 5 overestimates the number of processors for a given k. Indeed, some partial products A_iB_j can be scheduled on the same processors as some of the **PBR** and **PMR**, without increasing the parallel cost. In this case one can choose a larger value for k to reduce the complexity without needing any extra processors. The following example illustrates this optimization. We then summarizes the complexities and the performances from our benchmarks.

Example

If 4 processors are available, Theorem 5 tells us that one can achieve a complexity of $3M(n/2)$ for the multipartite algorithm with $k = 2$. In fact, choosing $k = 4$ reduces the complexity. For $k = 4$ the different tasks are given in Table 2. The optimal scheduling on 4 processors (based on our assumption on $M(n)$) is summarized in Table 3. This optimization leads to a parallel cost of $3M(n/4)+2M(n/2)$ instead of $3M(n/2)$ for $k = 2$.

In Table 3, we also remark that tasks T_1 and T_2 need to be completed before tasks T_{14} and T_{15} are started (the same holds for tasks T_{16} and T_{17} with tasks T_3 and T_4). On this example, the complexity of the multipartite $(2M(n/2) + 3M(n/4))$ is close, although greater than that of the bipartite algorithm (equivalent to $2M(n/2)+2M(n/4)$). However, the multipartite algorithm only requires one synchronization barrier instead of two for the bipartite multiplication.

Similarly, with 6 processors, choosing $k = 4$ allows to reduce the parallel cost. The optimal scheduling is given in Table 4, where tasks $T_1, \ldots, T_{13}$ are identical to those of Table 2 and tasks T_{14} to T_{19} correspond to $(Q_0+2^{n/4}Q_1)P_0$, $(Q_0+2^{n/4}Q_1)P_1$, $(Q_0+2^{n/4}Q_1)P_2$, $(Q_2+2^{n/4}Q_3)P_0$, $(Q_2+2^{n/4}Q_3)P_1$, and $(Q_2+2^{n/4}Q_3)P_2$ respectively, where $P = P_0+2^{n/3}P_1+2^{2n/3}P_2$. In this case, the parallel cost of the multipartite multiplication reduces to $M(n/4) + M(n/2) + M(n/2, n/3)$ which is better than the estimate given by Theorem 5.

Table 2: Description of the tasks of the multipartite multiplication for $k = 4$, where $\mu = -P^{-1} \bmod 2^{n/2}, \mu_0 = \mu \bmod 2^{n/4}, \nu = 2^{3n/2}/P, and \nu_1 = \nu/2^{n/4}; P = P_0 + 2^{n/2}P_1$

Tasks	Operations	Cost
T_0	$Q_0 = (A_0B_0)\mu \bmod 2^{n/2}$	$M(n/4) + M(n/2)$
T_1	$Q_1 = (A_0B_1 + A_1B_0)\mu_0 \bmod 2^{n/4}$	$2M(n/4) + M(n/4)$
$T_2, \ldots, T_{11}$	A_iB_j for $i + j = 2, \ldots, 4$	$M(n/4)$
T_{12}	$Q_2 = \frac{(A_3B_2 + A_2B_3)}{2^{n/4}}\nu_1$	$2M(n/4) + M(n/4)$
T_{13}	$Q_3 = (A_3B_3)\nu$	$M(n/4) + M(n/2)$
T_{14}	$(Q_0 + 2^{n/4}Q_1)P_0$	$M(n/2)$
T_{15}	$(Q_0 + 2^{n/4}Q_1)P_1$	$M(n/2)$
T_{16}	$(Q_2 + 2^{n/4}Q_3)P_0$	$M(n/2)$
T_{17}	$(Q_2 + 2^{n/4}Q_3)P_1$	$M(n/2)$

Table 3: Optimal scheduling of the 18 tasks involved in the multipartite multiplication when using 4 processors with $k = 4$

Processor	Tasks	Cost
P0	T_0, T_2, T_3, T_{14}	$3M(n/4) + 2M(n/2)$
P1	$T_1, T_4, T_5, T_6, T_{15}$	$6M(n/4) + M(n/2)$
P2	$T_{12}, T_7, T_8, T_9, T_{16}$	$6M(n/4) + M(n/2)$
P3	$T_{13}, T_{10}, T_{11}, T_{17}$	$3M(n/4) + 2M(n/2)$

Table 4: Optimal scheduling of the 20 tasks involved in the multipartite multiplication when using 6 processors with $k = 4$

Processor	Tasks	Cost
P0	T_0, T_{14}	$M(n/4) + M(n/2) + M(n/2, n/3)$
P1	T_1, T_2, T_{15}	$4M(n/4) + M(n/2, n/3)$
P2	$T_3, T_4, T_5, T_6, T_{16}$	$4M(n/4) + M(n/2, n/3)$
P3	$T_7, T_8, T_9, T_{10}, T_{17}$	$4M(n/4) + M(n/2, n/3)$
P4	T_{12}, T_{11}, T_{18}	$4M(n/4) + M(n/2, n/3)$
P5	T_{13}, T_{19}	$M(n/4) + M(n/2) + M(n/2, n/3)$

Table 5: Best costs for the multipartite multiplication on 4, 6, 8, and 12 cores with the corresponding k-values

#processors	k	Parallel Cost
4	4	$3M(n/4) + 2M(n/2)$
6	4	$M(n/4) + M(n/2) + M(n/2, n/3)$
8	8	$13M(n/8)$
12	6	$M(n/6) + M(n/2, n/3) + M(n/3, n/4)$

Note that the cost given in Table 4 may be further reduced. For example, with 6 processors, choosing $k = 6$ together with an optimized tasks scheduling leads to a parallel cost of $3M(n/6) + 2M(n/2, n/3)$. However, this estimate is only better than that for $k = 4$ if one assumes an underlying quadratic multiplication. More generally, the minimal complexity for the multipartite multiplication depends both on the choice of k and the underlying integer multiplication algorithms. According to these two parameters, it is possible to find an optimal scheduling for any given number of processors, although this optimization strategy seems difficult to automatize.

Benchmarks

We ran a lot of experiments (see Section B) in order to measure the performance of version of the multipartite algorithm we implemented. In Table 5, we give the parallel costs of versions of our multipartite multiplication that happen to be the fastest in practice.

We give all the timings from our benchmarks for various k-values and various number of cores in appendix B. Note that we use the term "cores" instead of "processors" as it reflects the current architectures where a single processor embeds multiple cores. Figure 5 illustrates the performances of all the existing parallel methods for integer modular multiplication against the sequential GMP modular multiplication. One can see that for small size integers (i.e. less than 2048 bits) the cost of the parallelism (thread launching and synchronization) dominates the total cost since the most efficient method is the sequential GMP modular multiplication. For larger integers (i.e. between 2048 and 16384 bits) the most efficient method is always the multipartite multiplication. This can be explained by the fact that, for such sizes, the cost of the parallelism still represent an important part of the total cost, and the few number of synchronizations in the multipartite multiplication pays in our favour. When integers are getting larger, the bipartite method tends to be the best alternative. This comes from the fact that its theoretical complexity is better than that of all the other methods. Note however that in the case of 6 cores, the performance of our multipartite multiplication is very close to that the bipartite method for large integers, while it is not the case for 4, 8 and 12 cores. Remember that our optimization of the multipartite method is based on the assumption that the underlying integer multiplication has quadratic complexity. Therefore, for very large integers, this assumption may not give the best results and the efficiency of our multipartite multiplication may be further improved by defining an optimization strategy based on integer multiplication with sub-quadratic or quasi-linear time. The above observation for 6 cores ($k = 4$) comes from the fact that optimization strategy remains valid when FFT multiplication is used. Finally, our benchmarks show that neither Barrett or Montgomery is a good alternative for software parallelism.

5.4 Parallel Modular Exponentiation

A typical case where fast modular multiplication is critical is for modular exponentiation, an operation of utmost importance for cryptographic algorithms such as RSA [14] or the Diffie-Hellmann-Merkle key exchange [15]. In order to evaluate the performance of our multipartite multiplication in this context, we implemented the classical binary left-to-right exponentiation (also known as square-and-multiply) which computes $g^e \bmod P$ with $g, e < P < 2^n$. It is well known that this algorithm requires on average $1.5 \log_2(e) = 3n/2$ modular multiplications. In fact, it requires n squares plus $n/2$ multiplications but for simplification

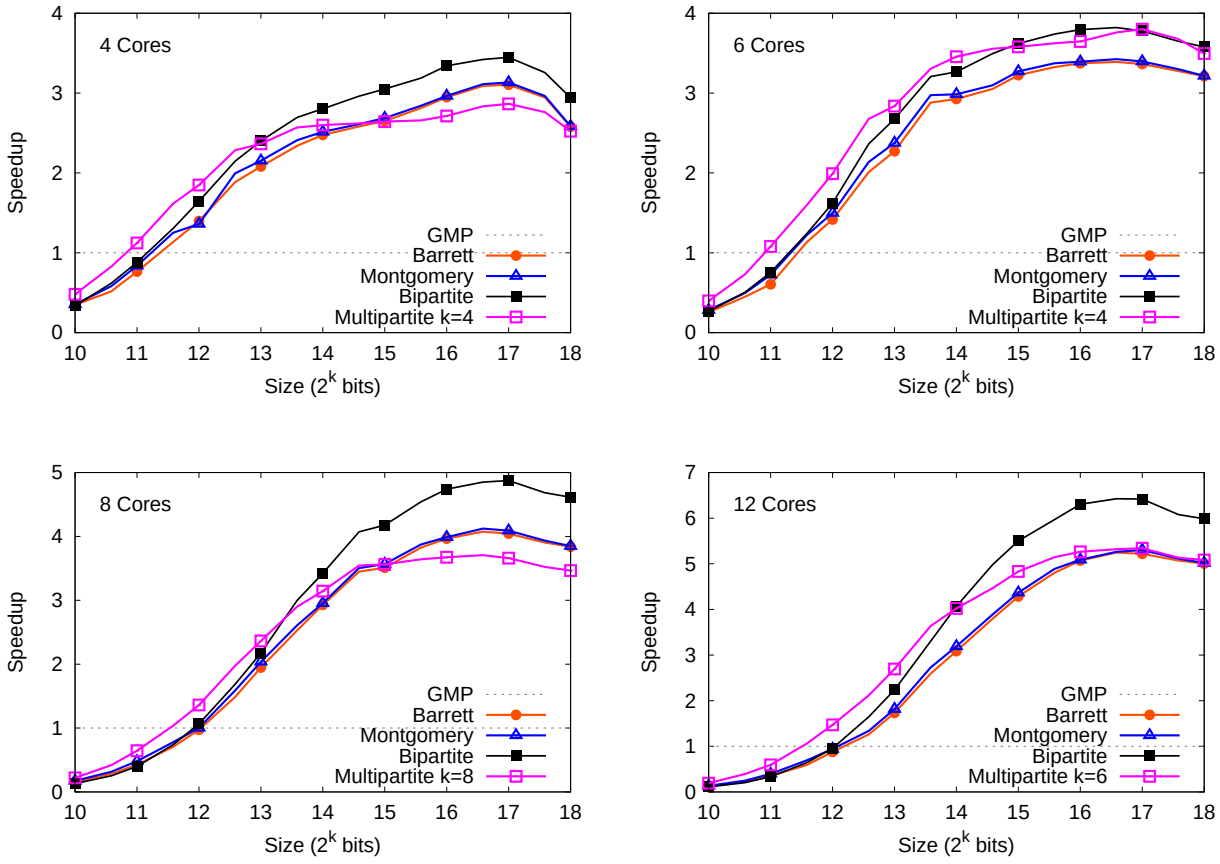

Figure 5: Performance comparison of Montgomery, Barrett, the bipartite and the multipartite modular multiplications

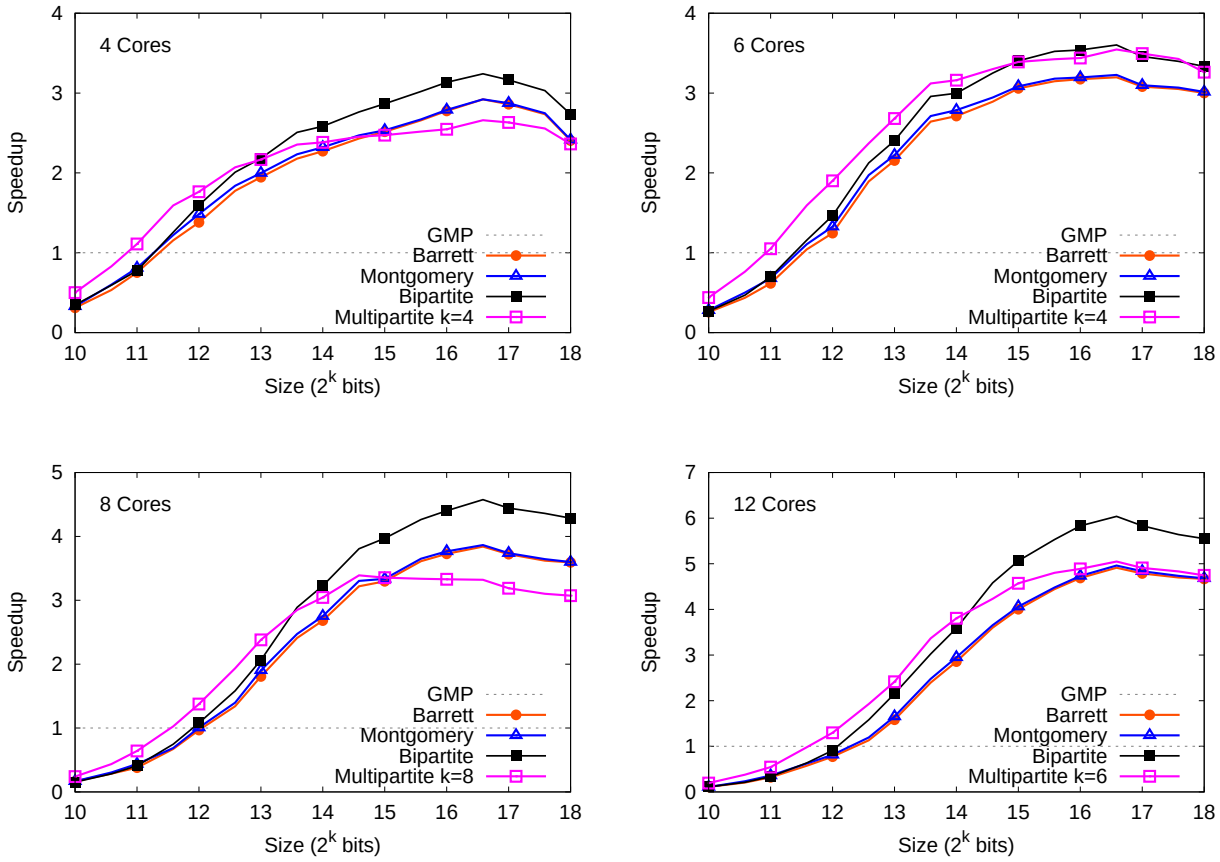

Figure 6: Performance comparison of a binary left-to-right modular exponentiation based on our parallel implementations of Montgomery, Barrett, the bipartite and multipartite modular multiplications

purpose our proof-of-concept implementation does not differentiate multiplications from squarings. Note that a multipartite squaring can be easily derived from the multipartite multiplication.

The chosen exponentiation algorithm is generic, meaning that we only had to change the modular multiplication in order to compare the exponentiations based on the four parallel algorithms presented in previous sections. Our exponentiation only requires one launching of tasks, followed by fully parallel computations.

Theoretically, the time to compute one exponentiation is simply the time to compute $3n/2$ modular multiplications. Figure 6 presents the performances of the different parallel exponentiations against a sequential version of the binary algorithm based on GMP sequential modular multiplication (`mpz_mul` followed by `mpz_tdiv_r`). Our timings, shown in Figure 6, confirm our assumption since the shape of the curves obtained for our naive modular exponentiation are similar to those shown in Figure 5 for modular multiplication only. In particular, the performance hierarchy is preserved and the fastest exponentiations are obtained with the fastest modular multiplications. The speedups are only inferior than those observed for our parallel multiplication benchmarks because `mpz_mul` is optimized for squarings whereas our multipartite implementation is not. For integers of sizes ranging from 2^{11} to 2^{13} bits (and up to 2^{15} bits on six cores), the multipartite multiplication leads to the fastest modular exponentiation.

6 Conclusion

We proposed a multipartite modular multiplication, which generalizes the bipartite algorithm, together with several software implementations dedicated to nowadays multi-core processors. We ran extensive benchmarks to measure its efficiency for a wide range of inputs, and we provided fair comparisons with other parallel implementations of Barrett, Montgomery and the bipartite algorithms. The best results were obtained thanks to optimal, hand-made task scheduling. Generalizing this optimizations to larger parameters (# cores, k-values) will be necessary for futures architectures with more than 12 cores, but this seems to be a difficult problem. Several problems remain to be studied. Our parallelization is based on a quadratic scheme (we perform k^2 partial products). A natural question is to analyze the performances of parallel implementations based on sub-quadratic methods such as Karatsuba and Toom-Cook. Going one step further, we shall also consider a parallel FFT algorithm in order to get a parallel cost close to $M(n)/k$ on k cores. Considering modular reduction only instead of modular multiplication should also be considered as it would give us way more freedom regarding operand splitting and parallelization levels. Finally, an optimized multipartite squaring would be necessary for fast parallel modular exponentiation.

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[2] D. J. Bernstein, *Algorithmic Number Theory.* MSRI Publications, 2008, vol. 44, ch. Fast multiplication and its applications, pp. 325–384.

[3] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic.* Cambridge University Press, 2010.

[4] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics—Doklady*, vol. 7, no. 7, pp. 595–596, Jan. 1963.

[5] A. L. Toom, "The complexity of a scheme of functional element realizing the multiplication of integers," *Soviet Mathematics*, vol. 3, pp. 714–716, 1963.

[6] D. Zuras, "More on squaring and multiplying larges integers," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 899–908, Aug. 1994.

[7] A. Schönage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[8] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology, CRYPTO'86*, ser. Lecture Notes in Computer Science, vol. 263. Springer, 1986, pp. 311–326.

[9] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[10] M. E. Kaihara and N. Takagi, "Bipartire modular multiplication method," *IEEE Transactions on Computers*, vol. 57, no. 2, pp. 157–164, 2008.

[11] A. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* CRC Press, 1997.

[12] K. Sakiyama, M. Knezevic, J. Fan, B. Preneel, and I. Verbauwhede, "Tripartite modular multiplication," *Integration, the VLSI Journal*, 2011, in Press, Corrected Proof. DOI: 10.1016/j.vlsi.2011.03.008.

[13] T. Granlund, "GMP, the GNU multiple precision arithmetic library," http://www.swox.com/gmp/.

[14] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[15] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.

A Exact complexity of the multipartite multiplication algorithm

$$
\begin{aligned}
& k^2 M\left(\frac{n}{k}\right) \\
& + 2 \sum_{i+j=\lceil \frac{k}{2} \rceil - 2}^{\lceil \frac{k}{2} \rceil - 1} \left[M\left(\frac{n}{2} - \frac{n(i+j)}{k}\right) + M\left(n, \frac{n}{2} - \frac{n(i+j)}{k}\right) \right] \\
& + 2 \sum_{i+j=0}^{\lceil \frac{k}{2} \rceil - 3} \left[M\left(\frac{2n}{k}, \frac{n}{2} - \frac{n(i+j)}{k}\right) + M\left(n, \frac{n}{2} - \frac{n(i+j)}{k}\right) \right]
\end{aligned}
\tag{20}
$$

B Summary of times for parallel modular multiplication algorithms

All the times below are given in μs and correspond to effective time of the parallel computation. We use the notation Multi($k = ...$) for the multipartite method using a splitting of the operands in k parts each.

Using 4 cores

n	GMP seq.	Barrett	Montgomery	Bipartite	Multi($k = 2$)	Multi($k = 4$)
1024	1.32	3.81	3.73	3.93	3.98	2.75
1536	2.53	4.88	4.34	4.08	4.24	3.05
2048	4.13	5.41	4.94	4.69	5.30	3.68
3072	8.18	7.20	6.54	6.27	6.72	5.06
4096	13.06	9.35	9.60	7.94	8.45	7.07
6144	26.03	13.81	13.05	12.13	12.71	11.40
8192	41.10	19.75	19.09	17.12	18.48	17.38
12288	79.76	34.10	33.09	29.60	32.33	31.04
16384	125.18	50.56	49.77	44.65	48.48	48.16
24576	233.07	90.27	89.26	78.71	86.97	88.96
32768	359.53	135.67	133.76	117.85	130.18	136.06
49152	655.69	233.09	230.74	205.72	227.51	246.68
65536	1003.86	340.52	338.67	300.43	331.88	370.01
98304	1817.74	588.45	584.13	531.34	574.89	641.33
131072	2716.14	875.00	866.76	787.79	858.99	948.19
196608	4581.89	1555.62	1546.41	1407.11	1526.28	1659.20
262144	6618.79	2566.33	2560.42	2249.76	2577.75	2623.53
393216	10799.80	4232.37	4219.57	3862.36	4206.71	4467.79

Using 6 cores

n	GMP seq.	Barrett	Montgomery	Bipartite	Multi($k=3$)	Multi($k=4$)	Multi($k=6$)
1024	1.32	5.19	4.66	4.97	3.30	3.32	3.18
1536	2.53	5.62	5.10	4.99	3.47	3.47	3.56
2048	4.13	6.83	5.71	5.47	4.07	3.83	3.88
3072	8.18	7.23	6.70	6.60	5.41	5.13	5.08
4096	13.06	9.22	8.72	8.06	6.73	6.56	6.63
6144	26.03	12.95	12.18	11.01	10.30	9.72	9.97
8192	41.10	18.10	17.30	15.36	15.51	14.48	14.72
12288	79.76	27.68	26.82	24.87	25.67	24.13	23.97
16384	125.18	42.78	41.94	38.34	39.20	36.22	36.76
24576	233.07	76.36	75.23	66.70	71.18	65.54	65.68
32768	359.53	111.52	109.90	99.33	106.32	100.44	99.69
49152	655.69	197.15	194.37	175.37	187.10	180.94	181.77
65536	1003.86	297.63	295.86	264.57	279.40	275.28	276.70
98304	1817.74	536.16	530.73	475.84	489.26	483.43	484.92
131072	2716.14	807.29	799.77	718.91	724.63	714.73	717.79
196608	4581.89	1398.94	1389.03	1255.69	1278.64	1246.51	1245.19
262144	6618.79	2061.01	2058.30	1850.16	2035.18	1893.40	1899.32
393216	10799.80	3484.32	3469.22	3223.20	3458.84	3216.16	3225.69

Using 8 cores

n	GMP seq.	Barrett	Montgomery	Bipartite	Multi($k=3$)	Multi($k=4$)	Multi($k=6$)	Multi($k=8$)
1024	1.32	8.50	7.62	10.12	5.59	5.70	6.02	5.85
1536	2.53	8.89	7.99	9.98	5.62	5.91	5.91	6.03
2048	4.13	9.73	8.59	10.33	6.13	6.73	6.46	6.44
3072	8.18	11.56	10.51	11.05	8.58	8.51	7.66	7.57
4096	13.06	13.44	13.01	12.25	10.09	9.71	9.02	8.87
6144	26.03	17.50	16.39	15.45	13.74	14.01	12.40	12.18
8192	41.10	21.11	20.18	18.90	18.52	17.65	16.05	16.06
12288	79.76	31.55	30.59	26.61	28.60	27.00	25.61	25.43
16384	125.18	42.75	42.36	36.58	41.19	38.35	38.79	37.80
24576	233.07	67.58	66.51	57.24	70.55	65.10	65.15	64.53
32768	359.53	102.44	100.75	86.06	104.83	96.05	100.80	100.58
49152	655.69	171.38	169.25	144.35	183.83	164.64	182.86	185.81
65536	1003.86	253.21	251.70	211.83	268.12	242.37	274.87	283.59
98304	1817.74	446.09	440.95	374.75	460.60	422.79	494.61	513.51
131072	2716.14	671.78	664.03	557.22	689.52	627.47	736.69	782.83
196608	4581.89	1173.37	1164.80	978.31	1208.48	1102.52	1276.74	1374.90
262144	6618.79	1723.76	1719.97	1434.54	1934.46	1783.66	1866.49	2013.57
393216	10799.80	3076.97	3064.03	2566.30	3342.68	2999.20	3335.80	3512.09

Using 12 cores

n	GMP seq.	Barrett	Montgomery	Bipartite	Multi($k = 4$)	Multi($k = 6$)	Multi($k = 8$)
1024	1.32	10.64	9.78	12.05	6.67	6.89	6.87
1536	2.53	11.23	10.22	12.35	7.12	6.44	6.96
2048	4.13	11.65	10.41	12.35	7.61	6.96	7.45
3072	8.18	13.86	11.84	12.77	8.54	7.72	7.86
4096	13.06	14.87	13.86	13.65	9.80	8.89	9.33
6144	26.03	20.65	19.46	15.83	13.03	12.30	12.30
8192	41.10	23.74	22.63	18.37	16.73	15.25	14.82
12288	79.76	30.74	29.22	24.14	24.61	21.93	22.05
16384	125.18	40.58	39.24	30.78	32.64	31.12	31.87
24576	233.07	61.39	59.90	46.78	53.67	52.19	51.64
32768	359.53	83.90	82.33	65.35	77.90	74.39	76.54
49152	655.69	136.45	134.15	109.84	134.65	127.44	140.09
65536	1003.86	197.99	197.27	159.23	202.74	190.74	209.76
98304	1817.74	346.44	345.43	282.92	352.99	341.65	373.94
131072	2716.14	520.58	512.33	423.20	518.82	508.47	566.53
196608	4581.89	903.31	895.97	753.73	894.20	892.68	992.61
262144	6618.79	1322.64	1319.41	1105.60	1327.70	1302.28	1458.21
393216	10799.80	2250.66	2240.18	1910.39	2313.70	2274.18	2487.44