Parallel modular multiplication on multi-core processors

Pascal Giorgi, Laurent Imbert, Thomas Izard

Parallel modular multiplication on multi-core processors

Pascal Giorgi
LIRMM, CNRS, UM2
Montpellier, France
pascal.giorgi@lirmm.fr

Laurent Imbert
LIRMM, CNRS, UM2
Montpellier, France
laurent.imbert@lirmm.fr

Thomas Izard
SILKAN
Montpellier, France
thomas.izard@silkan.com

Abstract

Current processors typically embed many cores running at high speed. The main goal of this paper is to assess the efficiency of software parallelism for low level arithmetic operations by providing a thorough comparison of several parallel modular multiplications. Famous methods such as Barrett, Montgomery as well as more recent algorithms are compared together with a novel k-ary multipartite multiplication which allows to split the computations into independent processes. Our experiments show that this new algorithm is well suited to software parallelism.

Keywords

Modular multiplication, bipartite, tripartite, k-ary multipartite algorithms, parallel arithmetic, multi-core

1. Introduction

Fast multiplication is critical for many applications [1], in particular when many of them need to be computed in sequence as for an exponentiation; an operation that is essential for public-key cryptography (RSA, DH, Elgamal, etc.) with operands ranging from 1000 to 15000+ bits depending on the security level[1]. In these cases, multiplications are performed in finite rings $\mathbb{Z}/P\mathbb{Z}$; an operation referred to as modular multiplication. At the same time, current processors include several arithmetic units, allowing to perform several computations in parallel. Since exponentiation cannot be efficiently parallelized, it seems natural to investigate parallel multiplication algorithms and their implementations on multi-core processors.

Modular multiplication algorithms fall into two main classes: so-called interleaved methods perform the computation as a combined multiply-and-modulo operation, providing both high regularity and memory efficiency; they are generally preferred for hardware implementation. However, such interleaved methods prevent the use of fast, sub-quadratic or quasi-linear algorithms such as Karatsuba [2], Toom-Cook [3], [4] and the FFT-based methods [5]. Hence, when dealing with large operands and when memory is not a major concern, modular multiplication is implemented as two separate operations. We refer the reader to [6, chapter 2] for a recent, excellent survey.

1. See the ECRYPT II recommendations at http://www.keylength.com/en/3/

The most famous modular reduction algorithms are due to Barrett [7] and Montgomery [8]. Interleaved variants of Montgomery multiplication [9] do exist, but these are not considered here. These two algorithms share many similarities. In particular, they essentially consist of two integer multiplications that are intrinsically sequential: the result of the first one is needed for the second one. Although it is possible to parallelize each integer multiplication, this implies thread synchronizations whose cost should not be underestimated.

To the best of our knowledge, the first modular multiplication/reduction algorithm where parallelism is possible at the modular level was proposed in 2008 by Kaihara and Takagi [10]. The so-called bipartite modular multiplication uses two independent modular reduction algorithms, which reduce the input from both the least and most significant digits at the same time. In [11], Sakiyama *et al.* proposed a natural extension of the bipartite algorithm that can be implemented with three processes at the cost of some extra synchronizations. We recall these algorithms and their complexity in Section 2.

The bipartite algorithm is designed for run on two independent units. When more cores are available, it is possible to parallelize the integer multiplications within the bipartite algorithm, but as we shall see, the number of thread synchronizations required by this approach induces an important overhead in practice. In this paper, we propose a novel k-ary multipartite algorithm which allows to split the computations on more than two independent processes, without increasing this parallelism overhead.

In Section 2, we recall the famous algorithms of Barrett and Montgomery and we introduce partial variants which proves useful for the sequel, in particular the bipartite algorithm is very easily described. We present our novel algorithm and its complexity in Section 3 and compare several parallelization strategies in Section 4. We present some timings in Section 5 which show that our algorithm may be of interest for multi-core processors.

2. Background

We consider multiple precision integers in radix β. Given $0 < P < \beta^n$ and $C \geq 0$, a modular reduction algorithm is a method for computing $R = C \bmod P$. When C is the result of a product AB we may talk about modular multiplication.

Very often, we want R to be fully reduced, *i.e.* less than P, but in some cases partial reduction may be sufficient. (We make the concept of partial reduction more precise below.) In the next paragraphs, we recall the algorithms of Montgomery [8], Barrett [7], as well as the more recent bipartite [10] and tripartite [11] algorithms. We express their complexity in terms of the number of integer multiplications. Following [6] we use $M(m,n)$ to denote the time to perform an integer multiplication with operands of size m and n respectively, or simply $M(n)$ when both operands have the same size.

2.1. Montgomery modular reduction

Given $0 < P < \beta^n$ and $0 \leq C < P^2$, Montgomery's algorithm computes the smallest integer Q such that $C + QP$ is a multiple of β^n. Hence $(C + QP)/\beta^n$ is exact and congruent to $C\beta^{-n} \pmod P$. If $\gcd(P, \beta) = 1$, this Q-value is computed as $Q = \mu C \bmod \beta^n$, where $\mu = -P^{-1} \bmod \beta^n$ is precomputed. A detailed description is given in [6]. When both the remainder and quotient are needed, Montgomery's method is also known as Hensel's division. Note that μC and QP can be evaluated using a low short product and a high short product respectively. The complexity of Montgomery reduction is $2M(n)$. By extension, the complexity of Montgomery multiplication is $3M(n)$.

In order to simplify the presentation of the next algorithms, let us now introduce a partial Montgomery reduction. Roughly speaking, the goal of this partial reduction is to zero out the t least significant digits of C. For $C < \beta^m, C < P^2$, Algorithm 1 computes $R \equiv C\beta^{-t} \pmod P$ for some $0 \leq t \leq n$ such that $0 \leq R < \max(\beta^{m-t}, \beta^n)$. In other words, it takes a value C of size m and computes $R \equiv C\beta^{-t} \pmod P$; a value that is shorter than C by t digits. If $t > m - n$, the return value R is the proper remainder, *i.e.* less than P.

Algorithm 1: PMR (Partial Montgomery Reduction)

Input: integers P, C, t, μ such that $\beta^{n-1} < P < \beta^n$, $\gcd(P, \beta) = 1$, $0 \leq C < P^2$, $C < \beta^m$, $0 \leq t \leq n$ and $\mu = -P^{-1} \bmod \beta^t$ (precomputed)

Output: $R \equiv C\beta^{-t} \pmod P$ with $0 \leq R < \max(\beta^{m-t}, \beta^n)$

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

The proof of correctness is easily deduced from that of the original Montgomery's reduction algorithm. (See [6, Chapter 2]). Let us focus on its complexity: step 1 corresponds to the multiplication of the t least significant digits of C with the t least significant digits of μ. This step costs either $M(m,t)$ when $m < t$ or $M(t)$ otherwise. Step 2 involves a multiplication of P and Q, of size n and t respectively.

As a result, the complexity is:

$$\mathrm{PMR}(t,n,m) = \begin{cases} M(t,m) + M(n,t) & \text{if } m < t \\ M(t) + M(n,t) & \text{otherwise} \end{cases} \quad (1)$$

2.2. Barrett modular reduction

Barrett's algorithm [7] computes an approximation of the quotient $\lfloor C/P \rfloor$ as $Q = \lfloor \lfloor C/\beta^n \rfloor \nu/\beta^n \rfloor$, 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 [6]. (See also [12] for a slightly different version.) Assuming full products, the complexity of Barrett's reduction is $2M(n)$ (or $3M(n)$ for Barrett's multiplication).

As for Montgomery's algorithm, we introduce a partial Barrett reduction algorithm (see Algorithm 2) which takes $C < P^2, C < \beta^m$ as input and computes $R \equiv C \pmod P$ such that $0 \leq R < \beta^{m-t}$ with $t \leq m - n$. The idea is to zero out the t most significant digits of C. This is achieved by computing the t leading digits of Q as:

$$Q = \left\lfloor \frac{\left\lfloor \frac{C}{\beta^{m-t}} \right\rfloor \left\lfloor \frac{\beta^{n+t}}{P} \right\rfloor}{\beta^t} \right\rfloor \beta^{m-n-t} \quad (2)$$

Note that the factor $\nu = \lfloor \beta^{n+t}/P \rfloor$ in the numerator replaces Barrett's original precomputed constant.

Algorithm 2: PBR (Partial Barrett Reduction)

Input: integers P, C, t, ν such that $\beta^{n-1} < P < \beta^n$, $0 \leq C < P^2$, $C < \beta^m$, $0 \leq t \leq m - n$ and $\nu = \lfloor \beta^{n+t}/P \rfloor$ (precomputed)

Output: $R \equiv C \pmod P$ with $0 \leq R < \beta^{m-t}$

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

Lemma 1: 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 $R \equiv C \pmod P$. Therefore, we only need to prove $0 \leq R < \beta^{m-t}$. Since $\nu < \beta^{n+t}/P$, writing C as $C = C_1\beta^{m-t} + C_0$ with $0 \leq C_0 < \beta^{m-t}$ gives

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

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

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

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

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

Thus

$$R = C - QP < \beta^{m-t} + 2\beta^{m-n-t}P \tag{3}$$

Finally, since $P < \beta^n$ we have $\beta^{m-n-t}P < \beta^{m-t}$. Hence, if $R \geq \beta^{m-t}$ then $R - \beta^{m-n-t}P \geq 0$. According to (3) at most two subtractions are required to ensure $0 \leq R < \beta^{m-t}$. □

Regarding complexity: since $\beta^{n-1} < P < \beta^n$ we have $\beta^t < \beta^{n+t}/P < \beta^{t+1}$ and thus $\beta^t \leq \nu < \beta^{t+1}$. Hence, step 1 is the multiplication of C_1 of size t and ν of size $t+1$ of asymptotic cost $M(t)$. However, as we shall encounter this situation in the k-ary multipartite multiplication algorithm presented in Section 3, we note that if $C = C_1'\beta^{m-s} + C_0$, with $0 \leq C_0 < \beta^{m-t}$ (*i.e.* if C_1 has only $s < t$ significant digits in the left-most positions) then the cost of step 1 can be reduced to $M(s,t)$. Step 2 is a multiplication of 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:

$$\text{PBR}(t,n,s) = \begin{cases} M(t,s) + M(n,t) & \text{if } s < t \\ M(t) + M(n,t) & \text{otherwise} \end{cases} \tag{4}$$

2.3. Bipartite modular multiplication

The bipartite algorithm proposed by Kaihara and Takagi [10] computes $AB \bmod P$ for $0 \leq A, B < P < \beta^n$ and $\gcd(P,\beta) = 1$ using two independent, parallel processes. One operand, say B, is split into two parts, say $B = B_1\beta^{n/2} + B_0$, such that $AB \bmod P = (AB_1\beta^{n/2} \bmod P + AB_0 \bmod P) \bmod P$. However, in that form the sizes of the operands to be reduced are very unbalanced: $2n$ for $AB_1\beta^{n/2}$ versus $3n/2$ for AB_0. So the computation of the former would take much longer to complete than the latter. Instead, Kaihara and Takagi use a Montgomery-like representation to compute the result of $AB\beta^{-n/2} \bmod P$ as $\left(AB_1 \bmod P + AB_0\beta^{-n/2} \bmod P\right) \bmod P$.

Using the previously defined **PBR** for $AB_1 \bmod P$ and **PMR** for $AB_0\beta^{-n/2} \bmod P$, the complexity of the bipartite algorithm 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 (1) and (4) respectively yields a total cost of $2M(n/2) + 4M(n, n/2)$.

2.4. Tripartite modular multiplication

The tripartite modular multiplication proposed by Sakiyama *et. al.* in [11] is a natural extension of the bipartite algorithm, where both operands are divided into two parts. Karatsuba's scheme is used to reduce the number of integer multiplications. The tripartite algorithm evaluates $AB\beta^{-n/2} \bmod P$ as

$$(A_1\beta^{n/2} + A_0)(B_1\beta^{n/2} + B_0)\beta^{-n/2} \bmod P$$

which is equivalent to

$$(X_2\beta^{n/2} + (X_1 - X_2 - X_0) + X_0\beta^{-n/2}) \bmod P$$

where $X_2 = A_1B_1$, $X_1 = (A_1 + A_0)(B_1 + B_0)$ and $X_0 = A_0B_0$. The reductions of both $X_2\beta^{n/2}$ and $X_0\beta^{-n/2}$ can be expressed using **PBR** and **PMR** respectively. The term $X_1 - X_2 - X_0$, of size at most n, is reduced by a small number of subtractions. Thus, the complexity of the tripartite multiplication is $3M(n/2)$ for X_0, X_1, X_2 plus $2M(n/2) + 2M(n, n/2)$ for the two reductions, leading to an overall complexity of $5M(n/2) + 2M(n, n/2)$.

3. A novel k-ary multipartite modular multiplication

In this section we introduce a generalization of the bipartite and tripartite algorithms, which allows to break the computations into an arbitrary number of independent processes, without increasing the parallelization overhead.

Let $0 \leq A, B < P < \beta^n$ and $\gcd(P,\beta) = 1$. Splitting A, B into k parts each as $A = \sum_{i=0}^{k-1} A_i\beta^{ni/k}$ and $B = \sum_{i=0}^{k-1} B_i\beta^{ni/k}$, the modular product shifted to the right by $n/2$ digits rewrites

$$AB\beta^{-n/2} \bmod P = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} A_iB_j\beta^{d_{i,j}} \bmod P, \tag{5}$$

where $d_{i,j} = n(i+j)/k - n/2$ and $-n/2 \leq d_{i,j} \leq 3n/2 - 2n/k$. Without loss of generality, we assume through the rest of the paper that $k|n$; this can always be achieved by padding operands with leading zeros.

As in the bipartite and tripartite algorithms, if $d_{i,j} < 0$, the products A_iB_j are reduced with **PMR** yielding $A_iB_j\beta^{d_{i,j}} \bmod P$. Similarly, if $d_{i,j} > n - 2n/k$, the terms $A_iB_j\beta^{d_{i,j}} \bmod P$ are reduced with **PBR**. For $0 \leq d_{i,j} \leq n - 2n/k$, we have $A_iB_j\beta^{d_{i,j}} < \beta^n$ and no reduction is necessary. Unlike the tripartite algorithm, all the terms are fully independent. Hence the k-ary multipartite algorithm can be implemented without synchronization. In Figure 1, we illustrate the case where both operands are divided into $k = 3$ parts. The final result $AB\beta^{-n/2}$ is then obtained by summing up modulo P all the partially reduced terms $A_iB_j\beta^{d_{i,j}}$.

3.1. Complexity analysis

In the form described above, the complexity of the k-ary multipartite algorithm depends on the number of calls to **PMR** and **PBR** as well as the exact number of significant digits of their respective inputs.

Lemma 2: Let $k > 0$ denote the number of blocks of each operand. Then, the number of calls to **PMR** (resp. **PBR**) is

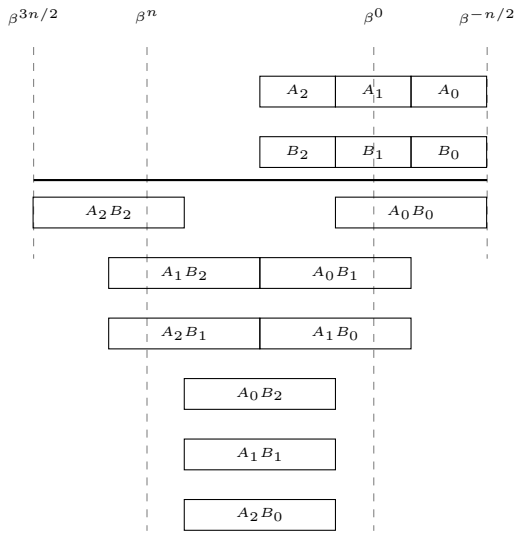

Figure 1: Illustrating the 3-ary multipartite multiplication

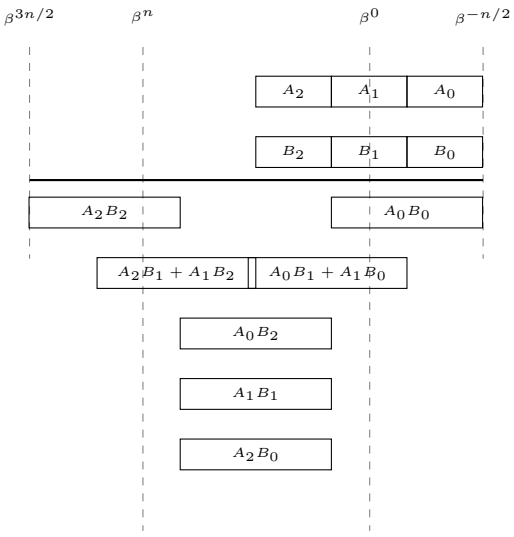

Figure 2: Reducing the number of call to **PMR** and **PBR** in the 3-ary multipartite algorithm by accumulating partial products of identical weight

equal to $k(k+2)/8$ if k is even and $(k+1)(k+3)/8$ if k is odd. Moreover, 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.

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 $0 \leq i+j \leq \lceil k/2 \rceil - 1$. Note also that for every value $i+j$, there are exactly $i+j+1$ partial products A_iB_j. Similarly, calls to **PBR** occur for $\lfloor 3k/2 \rfloor - 1 \leq i+j \leq 2k-2$ and there are exactly $2k-2-(i+j)+1$ partial products for every value $i+j$. Therefore, the number of calls to **PMR** (resp. **PBR**) 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. □

Analyzing those calls to **PMR** (resp. **PBR**), we note that some computations are redundant. Indeed, for each $A_iB_j\beta^{d_{i,j}}$ to be reduced modulo P, we compute a Q-value whose goal is to zero out some digits of that partial product. Thus, for a given weight $d_{i,j}$, several Q-values are computed to zero out digits of identical weight. Therefore, adding those partial products of identical weight together before computing a single Q-value is more efficient, at least theoretically. This is illustrated in Figure 2 for $k=3$. Note that this approach reduces the number of calls to **PMR** (resp. **PBR**) from $O(k^2)$ to $\lceil k/2 \rceil$. A description of the k-ary multipartite is given in Algorithm 3. Note that the partial products A_iB_j that need not be reduced can either be added together into $C_{d_{i,j}}$ as in Algorithm 3 (line 2) or at the end into R. This latter option will be considered in Section 4.4 to reduce the parallel complexity.

Theorem 1: The total arithmetic cost of the k-ary multipartite modular multiplication is at most

$$k^2 M\left(\frac{n}{k}\right) + 2\sum_{i+j=0}^{\lceil \frac{k}{2} \rceil - 1} \left(M\left(t_{i,j}, \frac{2n}{k}\right) + M\left(n, t_{i,j}\right)\right), \quad (6)$$

where $t_{i,j} = n/2 - n(i+j)/k$.

Algorithm 3: k**-ary multiplication**

Input: integers k, P, A, B, μ, ν such that $k \geq 2$, $0 < P < \beta^n$, $\gcd(P, \beta) = 1$, $0 \leq A, B < P$, $\mu = -1/P \bmod \beta^{n/2}$, $\nu = \left\lfloor \beta^{3n/2}/P \right\rfloor$ where $A = \sum_{i=0}^{k-1} A_i \beta^{in/k}$ and $B = \sum_{i=0}^{k-1} B_i \beta^{in/k}$

Output : $R \equiv AB\beta^{-n/2} \pmod P$ with $0 \leq R < P$

1: **foreach** $d_{i,j}$ **from** $-n/2$ **to** $3n/2 - 2n/k$ **by** n/k **do**
2: $\quad C_{d_{i,j}} \leftarrow \sum A_iB_j$ such that $d_{i,j} = n(i+j)/k - n/2$
3: $\quad$ **if** $d_{i,j} < 0$ **then**
4: $\quad\quad R_{d_{i,j}} \leftarrow$ **PMR**$(P, C_{d_{i,j}}, t_{i,j}, \mu \bmod \beta^{t_{i,j}})$, where $t_{i,j} = -d_{i,j}$
5: $\quad$ **else if** $d_{i,j} > n - 2n/k$ **then**
6: $\quad\quad R_{d_{i,j}} \leftarrow$ **PBR**$(P, C_{d_{i,j}}\beta^{d_{i,j}}, t_{i,j}, \left\lfloor \nu/\beta^{n/2-t_{i,j}} \right\rfloor)$, where $t_{i,j} = d_{i,j} + 2n/k - n$
7: $\quad$ **else**
8: $\quad\quad R_{d_{i,j}} \leftarrow C_{d_{i,j}}$
9: $R \leftarrow \sum R_{d_{i,j}}$
10: **while** $R \geq P$ **do** $R \leftarrow R - P$
11: **return** R

Proof: First, notice that there are k^2 products A_iB_j with operands of size n/k, whence the term $k^2M(n/k)$. The other multiplications come from the calls to **PMR** and **PBR**. As seen in the proof of Lemma 2, there are exactly $\lceil k/2 \rceil$ calls to **PMR** and $\lceil k/2 \rceil$ calls to **PBR**. Now, by adding together partial products of identical weight, the operands to be reduced may grow due to carry propagation. However, we note that these carries are not an issue in **PMR** since they do not change the number of digits to be reduced. For **PBR**, the problem is easily bypassed by zeroing out the $t_{i,j}$ most significant digits (as before), at the cost of a few extra subtraction at

the end. (Note that Barrett's algorithm already requires a few final subtractions anyway.) In practice, this extra number of subtractions is negligible. As illustrated in Figure 2, the number of digits to zero out with **PMR** is exactly $-d_{i,j}$. Therefore, we have $t_{i,j} = -d_{i,j} = n/2 - n(i+j)/k$. Following (1), each call to **PMR** costs PMR$(t_{i,j}, n, m_{i,j})$, *i.e.* at most $M(t_{i,j}, 2n/k) + M(n, t_{i,j})$. Symmetrically, for each $\lfloor 3k/2 \rfloor - 1 \leq i + j \leq 2k - 2$, our algorithm performs a partial Barrett reduction, whose cost is bounded by $M(t_{i,j}, 2n/k) + M(n, t_{i,j})$. Hence the thesis. □

Note that the complexity given in (6) does not take into account the cost induced by the parallelism, in particular that induced by thread synchronizations. In the next section, we discuss several parallel implementation strategies.

4. Parallelization

In this section, we analyze and compare several parallel implementations of the algorithms presented above. We give the parallel complexity together with the number of synchronizations between concurrent threads. The latter should not be underestimated as it largely impacts practical performances. In [13] the overhead due to synchronization is defined as follows: if Ts is the sequential time for a section of code, and Tp the time for the parallel version of this on p processors, then the overhead is given by $Op = Tp - Ts/p$. Using their benchmarking tools, we were able to estimate the number of cycles of one synchronization on our architecture (Intel Xeon X5650 Westmere running at 2.66 GHz). We obtained the following results: 927 cycles for 2 threads, 1303 cycles for 3 threads, 1559 cycles for 4 threads, 2221 for 6 threads and 2571 for 8 threads.

In order to parallelize the algorithms without introducing too much parallelism overhead, we heavily rely on a quadratic scheme. The idea is to split the operands in θ_1 (resp. θ_2) chunks and to perform $\theta_1\theta_2$ multiplications in parallel. Thus, one can easily derive a parallel algorithm to multiply two n-digits integers with a parallel complexity of $M(n/\theta_1, n/\theta_2)$ on $T = \theta_1\theta_2$ cores and 2 thread synchronizations occuring before and after the summation of the partial products. Our parallel complexities are solely based on that quadratic scheme. Note that additions are not performed in parallel as it would require too many synchronizations. Although sub-quadratic schemes such as Karatsuba or Toom-Cook perform fewer operations, they also introduce many synchronizations which make them unsuitable in our context. They would however be of interest for bigger operands.

In the next sections, we consider a model where the algorithms receive synchronized data and output synchronized data such that it is possible to call them multiple times in sequence without extra synchronization.

In order to provide bounds and legible comparisons, we consider the following assumption on integer multiplication, which almost reflects the practical behaviour of integer multiplication except for quasi linear methods:

$$M(n_1 + n_2, n) = M(n_1, n) + M(n_2, n) \tag{7}$$

4.1. Barrett/Montgomery:

Although intrinsically sequential, both algorithms can be distributed on several arithmetic units by parallelizing the integer multiplications. Using the quadratic scheme, both Barrett and Montgomery's algorithms have a parallel complexity of $3M(n/\theta_1, n/\theta_2)$ on $T = \theta_1\theta_2$ cores. Each integer multiplication requires 1 synchronization after the parallel evaluation of the partial products, plus 1 synchronization at the end (after the summation of those partial products), allowing to move on to the next multiplication with synchronized data. Hence, both Barrett and Montgomery algorithms require 6 synchronizations.

Note that these algorithms are symmetric and have the same complexity. We verified experimentally that the performances of both methods are indeed very close. For the sake of clarity, we only give results for Montgomery's multiplication.

4.2. Bipartite:

The Bipartite multiplication was designed to run in parallel on two cores, with a parallel complexity of $M(n/2) + 2M(n, n/2)$. However, if more than two arithmetic units are available, the integer multiplications in **PMR** and **PBR** can also be parallelized, using the strategies discussed above. On $T = \theta_1\theta_2$ cores, splitting the operands in θ_1 and $\theta_2/2$ chunks, the cost becomes $M(n/2\theta_1, n/\theta_2) + 2M(n/\theta_1, n/\theta_2)$, or equivalently $2.5M(n/\theta_1, n/\theta_2)$ using (7) (assuming θ_2 is even). As for Montgomery, the bipartite algorithm requires 6 synchronizations.

4.3. Tripartite:

As seen in Section 2.4, the complexity of the tripartite multiplication is $3M(n/2)$ for X_0, X_1, X_2 plus $2M(n/2) + 2M(n, n/2)$ for the two reductions. X_0, X_1, X_2 can be computed in parallel on three cores in $M(n/2)$. After a synchronization, the two reductions can also be computed in parallel in $M(n/2) + M(n, n/2)$; each reduction involving an extra synchronisation. As for the bipartite algorithm, the integer multiplications can be parallelized if more units are available at the cost of some extra synchronizations. On $T = \theta_1\theta_2$ cores, splitting the operands in θ_1 and $\theta_2/3$ chunks, the cost becomes $2M(n/2\theta_1, 3n/2\theta_2) + M(n/\theta_1, 3n/2\theta_2)$, or equivalently $3M(n/\theta_1, n/\theta_2)$ using (7) (assuming θ_2 is divisible by 3). The tripartite algorithm requires a total of 6 synchronizations. In [11] a variant with 5 tasks using two levels of Karatsuba is also presented. Although potentially of interest for hardware implementations, the number of synchronizations makes it unsuitable in software for the targeted operand sizes.

4.4. k-ary multipartite:

The k-ary multipartite multiplication offers more flexibility. A naive implementation would require k^2 threads, one for each

$A_iB_j\beta^{d_{i,j}}$, with a parallel complexity equivalent to the cost of the most expensive of them. That is, the computations of $A_0B_0\beta^{-n/2} \bmod P$ using **PMR** or that of $A_{k-1}B_{k-1} \bmod P$ using **PBR** of exact same cost. Although feasible for small values of k, this strategy requires many cores and is very unbalanced.

As pointed out in Section 3.1, it is more efficient to combine the partial products of identical weight together and to compute only one Q-value per weight as in Algorithm 3. Indeed, we reach the same parallel complexity while reducing the number of threads. This is achieved by observing that the instructions in the foreach block of Algorithms 4 are independent and can therefore be computed in parallel. To further reduce the complexity, we postpone and reduce all the products by P appearing in each **PBR** and **PMR** of Algorithm 3 into a unique multiplication QP done, in parallel, at the end. In order to do so, each thread computes $C_{d_{i,j}}$ (as in Algorithm 3) together with the corresponding Q-value. All these Q-values are then added together after a thread synchronization. This is illustrated in Algorithm 4.

Algorithm 4: k**-ary multiplication**

Input: integers k, P, A, B, μ, ν such that $k \geq 2$, $0 < P < \beta^n$, $\gcd(P, \beta) = 1$, $0 \leq A, B < P$, $\mu = -1/P \bmod \beta^{n/2}$, $\nu = \left\lfloor \beta^{3n/2}/P \right\rfloor$ where $A = \sum_{i=0}^{k-1} A_i\beta^{in/k}$ and $B = \sum_{i=0}^{k-1} B_i\beta^{in/k}$

Output : $R \equiv AB\beta^{-n/2} \pmod P$ with $0 \leq R < P$

1: **foreach** $d_{i,j}$ **from** $-n/2$ **to** $3n/2 - 2n/k$ **by** n/k **do**
2: $\quad C_{d_{i,j}} \leftarrow \sum A_iB_j$ such that $d_{i,j} = n(i+j)/k - n/2$
3: $\quad$ **if** $d_{i,j} < 0$ **then**
$\quad\quad$ /* Compute $Q_{d_{i,j}}$ as in **PMR** of Algo 3 */
4: $\quad\quad Q_{d_{i,j}} \leftarrow -(C_{d_{i,j}}\mu \bmod \beta^{t_{i,j}})\beta^{n/2+d_{i,j}}$, where $t_{i,j} = -d_{i,j}$
5: $\quad$ **else if** $d_{i,j} > n - 2n/k$ **then**
$\quad\quad$ /* Compute $Q_{d_{i,j}}$ as in **PBR** of Algo 3 */
6: $\quad\quad Q_{d_{i,j}} \leftarrow \left\lfloor \left\lfloor C_{d_{i,j}}\beta^{d_{i,j}-n} \right\rfloor \left\lfloor \nu\beta^{t_{i,j}-n/2} \right\rfloor /\beta^{t_{i,j}} \right\rfloor \beta^{n/2}$, where $t_{i,j} = d_{i,j} + 2n/k - n$
7: $\quad$ **else**
8: $\quad\quad Q_{d_{i,j}} \leftarrow 0$
9: $Q \leftarrow \sum Q_{d_{i,j}}$
10: $R \leftarrow \left(\sum C_{d_{i,j}}\beta^{n/2+d_{i,j}} - QP\right)\beta^{-n/2}$
11: **while** $R \geq P$ **do** $R \leftarrow R - P$
12: **return** R

Before stating our main theorem which gives an upper bound on the parallel complexity of Algorithm 4, we present the scheduling principle leading to this bound on two small examples. Starting with $k = 2$, it is easy to see that computing A_0B_0 plus its corresponding Q-value costs $2M(n/2)$; the same applies for A_1B_1. This is equivalent to the cost of computing the two products A_0B_1 and A_1B_0. Hence, the 2-ary multipartite can be implemented on three threads in $2M(n/2)$, plus the cost of computing QP in parallel after a thread synchronization. For $k = 3$ (see Figure 2), A_0B_0 and its corresponding Q-value costs $c_0 = M(n/3) + M(n/2) = 13M(n/6)$ according to (7). The cost is identical for A_2B_2 and its corresponding Q-value. Computing $A_0B_1 + A_1B_0$ and the corresponding Q-value costs $c_1 = 2M(n/3) + M(n/6) = 9M(n/6)$. Symmetrically, evaluating $A_2B_1 + A_1B_2$ and its Q-value costs c_1 as well. Since $c1 < c_0$, those computations can be performed in parallel on four cores in $c_0 = 13M(n/6)$. The three remaining products A_0B_2, A_1B_1, A_2B_0 can be computed on one thread in $3M(n/3) = 12M(n/6) < c_0$. (Although it does not change the number of threads for $k = 3$, those remaining partial products should be considered independently as it allows a finer task scheduling for larger values of k.) Hence, the 3-ary multipartite multiplication can be implemented on five threads in $13M(n/6)$, plus the cost of computing QP in parallel after a thread synchronization. Using the same idea, Theorem 2 below gives an upper bound on the parallel complexity of the k-ary multipartite algorithm given in Algorithm 4. This theorem assumes that $k|n, \theta_1|n$ and $\theta_2|n$, which can always be achieved with padding.

Theorem 2: Let us assume that $M(n_1 + n_2, n) = M(n_1, n) + M(n_2, n)$. If $T = \theta_1\theta_2$ cores are available, then using a quadratic scheme with $k > 3$ and $T \geq 3\lceil k/2 \rceil$, the parallel complexity of the k-ary multipartite multiplication is at most

$$M\left(\frac{n}{k}\right) + M\left(\frac{n}{2}, \frac{2n}{k}\right) + M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right) \tag{8}$$

Proof: The term $M\left(n/\theta_1, n/\theta_2\right)$ comes from the final product QP, computed in parallel on $\theta_1\theta_2$ cores. So, let us focus on the other terms. We start by proving that any of the following computations can be computed in at most $M(n/k) + M(n/2, 2n/k)$:

(i) A_0B_0 and its corresponding Q-value,
(ii) $\sum_{\ell=i+j} A_iB_j$ and its corresponding Q-value plus ℓ non-reduced A_iB_j for $1 \leq \ell \leq \lceil k/2 \rceil - 1$,
(iii) $k+1$ partial products A_iB_j.

According to (1), the computation of A_0B_0 and its corresponding Q-value costs $M\left(\frac{n}{k}\right) + M\left(\frac{2n}{k}, \frac{n}{2}\right)$, proving (i). Using (1) again, computing $\sum_{\ell=i+j} A_iB_j$ for $1 \leq \ell \leq \lceil k/2 \rceil - 1$, and the corresponding Q-values costs

$$(\ell+1)M\left(\frac{n}{k}\right) + M\left(\frac{n}{2} - \ell\frac{n}{k}, \frac{2n}{k}\right) \tag{9}$$

Under our assumption on $M(n)$, we have $M\left(\frac{n}{2} - \ell\frac{n}{k}, \frac{2n}{k}\right) = (k-2l)M\left(\frac{n}{k}\right)$. Adding the cost of ℓ products A_iB_j gives a total of $(k+1)M\left(\frac{n}{k}\right)$, whence (ii). Finally, using (7), it is easy to see that $M\left(\frac{2n}{k}, \frac{n}{2}\right) = kM\left(\frac{n}{k}\right)$, so (iii) is proved too.

To complete the proof, we need to show that there exists a scheduling with $3\lceil k/2 \rceil$ cores satisfying (8). According to point (ii), all $C_{d_{i,j}}$ and $Q_{d_{i,j}}$ such that $d_{i,j} < 0$ or $d_{i,j} > n - 2n/k$ can be computed in $M\left(\frac{2n}{k}, \frac{n}{2}\right) + M\left(\frac{n}{k}\right)$ on $2\lceil k/2 \rceil$ cores (this of course includes A_0B_0, $A_{k-1}B_{k-1}$ and their respective Q-values). This is achieved by assigning one core per $C_{d_{i,j}}, Q_{d_{i,j}}$. Furthermore, the same scheduling allows to perform ℓ extra A_iB_j per core without increasing

the cost. (Note the ℓ-values depends on $i+j$, hence are not the same for all the cores.) The remaining terms correspond to the products A_iB_j such that $0 \leq d_{i,j} \leq n - 2n/k$. Their number is exactly $N_k = k^2 - 2\sum_{l=0}^{\lceil k/2 \rceil - 1} 2l + 1 = k^2 - 2\lceil k/2 \rceil^2$. According to (iii), one can handle $k+1$ partial products A_iB_j on one core in the given time. Replacing k by $\lceil k/2 \rceil + \lfloor k/2 \rfloor$ in the previous equation, gives $N_k \leq k \lfloor k/2 \rfloor < (k+1) \lfloor k/2 \rfloor$. Hence, $\lfloor k/2 \rfloor$ cores are needed for the N_k remaining products, which concludes the proof. □

Corollary 1: Let $T = \theta_1\theta_2$. Let $k = 2T/3$. Then, assuming $M(n_1+n_2, n) = M(n_1, n)+M(n_2, n)$, the parallel complexity of the k-ary multipartite algorithm is bounded by

$$(2.5 + \frac{9}{4\theta_1\theta_2})M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right) \tag{10}$$

and the number of thread synchronizations is exactly 3.

Proof: Using (8) with $k = 2\theta_1\theta_2/3$, the proof is immediate. □

As illustrated in Section 5, we observed that synchronizations are very expensive and should be limited. In particular for small operands, when the relative cost between a synchronization and a product is very large. In those cases, one may prefer not to gather all Q-values together before computing QP. Each thread then has to compute its own product QP which costs $M\left(\frac{n}{2} - \ell\frac{n}{k}, n\right)$. This, of course drastically increases the parallel complexity but saves 1 synchronization, possibly leading to significant speed-ups in practice (see Section 5). For completeness, the parallel complexity of this approach is $(1.5 + \frac{9}{4\theta_1\theta_2} + \frac{\theta_1\theta_2}{2})M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$. In the next section, we refer to Algorithm 4 as k-ary multipartite version 1, and to Algorithm 3 as version 2. In Table 1 we summarize the parallel complexities and the number of synchronizations of the presented parallel modular multiplication algorithms.

Algorithm	Parallel complexity on $\theta_1\theta_2$ cores	# synch.
Montgomery/Barret	$3M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	6
Bipartite	$2.5M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	$(*)$ 6
k-ary multipartite v1	$(2.5 + \frac{9}{4\theta_1\theta_2})M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	3
k-ary multipartite v2	$(1.5 + \frac{9}{4\theta_1\theta_2} + \frac{\theta_1\theta_2}{2})M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	2

Table 1: Parallel complexity of various modular multiplication algorithms. $(*)$ when $\theta_1\theta_2 = 2$, the bipartite algorithm requires only 2 synchronizations

5. Comparisons

As already mentioned, software parallelism induces a cost that is only amortized when computation time is important. This extra cost comes from thread launching (`clone()`, `fork()`), thread synchronization (`wait()`) and memories operations (placement, transfers and cache misses). This fact motivates our study of a parallel modular multiplication algorithm which minimize this impact.

In order to validate our theoretical study, we developed a C++ library implementing the algorithms presented in the previous sections on a shared memory MIMD architecture.

In the following, we discuss some benchmarks of this library on an architecture embedding two Intel Xeon processors X5650 Westmere, with six cores each, running at 2.66 GHz. Our implementation is based on the GNU multiple precision (GMP) library version 5.0.2 [14] and the OpenMP API framework. Compilation was done with the Intel C++ compiler version 10. Our implementation is generic in the operands' sizes. The sequential integer operations computed by each parallel task are the ones from GMP (e.g. product is done with `mpn_mul`). For each algorithm and number of cores, we optimized the task scheduling in order to minimize the parallel complexity and to reduce the number of synchronizations. When possible, parallel tasks were gathered on the same processor to avoid data movement through the system bus. Our timings are summarized in Table 2.

In order to provide a universal reference, we give the time to perform one modular multiplication on one thread, *i.e.* a sequential implementation. The row entitled "Best seq." in Table 2 always corresponds to the fastest option between our sequential implementation of Montgomery and GMP (`mpn_mul`, `mpn_mod`). For parallel integer multiplications, we implemented both the quadratic scheme and Karatsuba whenever possible, and observed that the later was only advantageous when the number of cores is a multiple of 3. In the other cases, the quadratic scheme was always faster. As in the sequential case, our implementations always call the best parallel integer multiplication strategy available.

The bound given in the corollary of Theorem 2 assumes that $k = 2T/3$, where T is the number of cores. For 3 and 6 cores, and inputs of size less than 12000 bits, the best timings were indeed obtained with the k-ary multipartite algorithm with $k = 2$ and $k = 4$ respectively. When the number of cores did not allow an optimal mapping, we observed that choosing $k > 2T/3$ allows a finer scheduling resulting in faster implementations. On 4 and 8 cores, our timings correspond to the 4-ary and 8-ary multipartite algorithms respectively, where extra A_iB_j have been scheduled by hand.

For the larger operands, we also observed that the bipartite algorithm was the fastest. This is not surprising since the parallel complexity given in Table 1 is always better than the other algorithms and since the parallelism overhead (a constant factor) becomes negligible. The fact that the k-ary multipartite catches up with the bipartite for increasing T is not surprising either, this reflects the complexity given in Table 1.

For the smaller operands, however, the cost induced by the parallelism is important. This explains why the k-ary multipartite version 2 which only needs 1 synchronization is faster than version 1 which requires 2.

Finally, one should observe that it may be advantageous not to use all the cores available. For example, if one considers 2048-bit operands and if 8 cores are available, one can observe that the 2-ary multipartite version 2 on 3 cores is faster.

		sizes (in bits)								
	Algorithm	1024	1536	2048	3072	4096	6144	8192	12288	16384
1 Thread	Best seq.	1.32	2.53	4.13	8.18	13.06	26.03	41.10	79.76	125.18
3 Threads	Montgomery	3.63	4.15	4.95	6.71	8.84	13.42	20.11	33.96	50.50
	2-ary multi. v1	2.59	3.23	3.83	5.49	7.43	12.24	18.26	32.17	48.40
	2-ary multi. v2	1.47	2.04	2.86	4.84	7.45	12.91	19.95	37.59	58.48
4 Threads	Montgomery	3.73	4.34	4.94	6.54	9.60	13.05	19.09	33.09	49.77
	Bipartite	3.93	4.08	4.69	6.27	7.94	12.13	17.12	29.60	44.65
	4-ary multi. v1	2.75	3.05	3.68	5.06	7.07	11.40	17.38	31.04	48.16
	4-ary multi. v2	1.68	2.14	2.90	4.74	7.20	13.43	20.60	37.88	60.99
6 Threads	Montgomery	4.66	5.10	5.71	6.70	8.72	12.18	17.30	26.82	41.94
	Bipartite	4.82	5.20	5.39	6.45	7.88	10.99	15.16	22.92	34.58
	4-ary multi. v1	3.32	3.47	3.83	5.13	6.56	9.72	14.48	24.13	36.22
	4-ary multi. v2	1.95	2.42	3.03	4.96	6.91	12.00	17.82	32.08	49.84
8 Threads	Montgomery	7.62	7.99	8.59	10.51	13.01	16.39	20.18	30.59	42.36
	Bipartite	10.12	9.98	10.33	11.05	12.25	15.45	18.90	26.61	36.58
	8-ary multi. v1	5.85	6.03	6.44	7.57	8.87	12.18	16.06	25.43	37.80
	8-ary multi. v2	3.98	4.29	4.92	6.59	8.84	13.81	19.88	33.92	51.10

Table 2: Timings (in μs) for several parallel modular multiplication algorithms on 1, 3, 4, 6 and 8 cores, for operands ranging from 1024 to 16384 bits. For a given number of cores and a given size, the gray cells represent the fastest algorithm

6. Conclusions

The experiments presented in this article show that parallelization of modular multiplication is relevant at a software level. Increasing the number of core improves the performance but a full usage of the cores may not be the better strategy for certain sizes. For operands smaller than 2^{13} bits, it seems preferable to use synchronization-friendly algorithms rather than the one with the best complexity. Our k-ary multipartite algorithm, which is a synchronization-friendly generalization of the bipartite and tripartite algorithms, is therefore a good alternative.

For larger sizes, the bipartite method clearly offers the best alternative since the arithmetic complexity dominates the parallelism overhead. However, as soon as the number of core increases, the gain of the bipartite over our k-partite algorithm is postponed to larger integers.

The best results for the k-ary multipartite algorithm were obtained thanks to the scheduling proposed in Theorem 2 completed with hand optimizations when the number of cores did not allow a direct mapping. Generalizing these optimizations would allow to provide an automatic scheduler for efficient parallel modular multiplication on any number of cores.

It seems also interesting to consider the modular reduction problem rather than modular multiplication. An idea would be to use an heterogeneous splitting of the operand to be reduced. Modular squaring should be also of interest since it is a major operation for exponentiation.

Acknowledgments

We would like to thank the anonymous referees for their careful reading and their very useful comments in the preparation of the final version of the manuscript. This work has been supported by the ANR under the grants HPAC (ANR-11-BS02-013), CATREL (ANR-12-BS02-001) and PAVOIS (ANR-12-BS02-002).

References

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

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

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

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

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

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

[7] 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.

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

[9] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

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

[11] 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.

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

[13] "The EPCC OpenMP microbenchmarks v3.0," available at http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html.

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