Fast Massively Parallel Modular Arithmetic

B. Dixon*[†] A. K. Lenstra[‡]

July 10, 1996

Abstract

High-performance implementations on massively parallel architectures of many number theoretic algorithms and cryptographic schemes require efficient methods to perform modular arithmetic. This paper shows how this can be achieved on a SIMD array of processors. The goal of our algorithms is to trade parallelism for time, which we are able to do in an optimal fashion. For the purposes of performing modular arithmetic, we can effectively reconfigure a 16K processor massively parallel SIMD computer into a machine having 16K/k processors that are k times faster. This is particularly useful for our high-performance application, where a high degree of slow parallelism is much less efficient than a lower degree of faster parallelism.

1 Introduction

Fast modular multiplication is crucial for an efficient implementation of many number theoretic algorithms or cryptographic schemes. In particular, the elliptic curve integer factoring algorithm [3] spends most of its time multiplying numbers mod n, where n is the number that one attempts to factor [1]. In our first implementation of the elliptic curve algorithm on the 16K processor MasPar[1] computer we ran 16K completely independent factoring attempts, on the same n. From a high level, the algorithm was simple since the attempts could

*Department of Computer Science, Princeton University, Princeton, NJ 08544.

[†]Research partially supported by a National Science Foundation Graduate Fellowship and by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center, Grant No. NSF-STC88-09648.

[‡]Bellcore, Room 2Q-334, 445 South Street, Morristown, NJ 07960, U. S. A.

[1]It is the policy of Bellcore to avoid any statements of comparative analysis or evaluation of products or vendors. Any mention of products or vendors in this presentation or accompanying printed materials is done

easily work in a data parallel manner. The process was organized in a such way that the 16K independent factoring attempts did not overlap with each other, so that we achieved a 16K-fold speed-up compared to a single attempt running on one processor.

This is as good as one can hope to achieve, except that the elliptic curve method has the peculiarity that its optimal degree of parallelism depends on and grows with the size of the factor p one tries to find. Similarly, the optimal effort to spend per trial grows with the size of p. An optimal application of the elliptic curve method with 16K attempts in parallel would aim for much larger p than is currently believed to be in the realistic range, and the run-time per attempt would be exceedingly long, especially using the slow MasPar processors. So, although our first implementation might turn out to be very useful on later generations of massively parallel machines that have much faster processors, a better solution for the current technology would be to have fewer but faster attempts. This requires distributing modular arithmetic among multiple processors, which motivates the research presented here.

In this paper we describe how modular multiplication can be carried out efficiently and in parallel on large 'Single Instruction Multiple Data' (SIMD) arrays of processors. Using the algorithm presented here, a modular multiplication using k processors requires time about $1/k$ times the amount of time to perform a modular multiplication using one processor, which is the optimal speed-up one can achieve using k processors. This should be compared to the straightforward SIMD implementation, which would require $1.5/k$ times the time for the one processor version, due to an inherent unparallelizable inefficiency in the algorithm. We have been able to overcome this efficiency, thus gaining a speed-up of a factor 1.5.

In addition to applications to cryptanalysis, our method can be used for a 'central facility' that has to sign or verify public key certificates, or decrypt RSA messages [6], for many users simultaneously. There too, as in the elliptic curve implementation, a low degree of fast parallelism is more desirable than a high degree of slow parallelism. Such applications are not uncommon in the telecommunication industry. In the following sections, we describe the multiplication algorithms. Section 2 describes the sequential version, and section 3 begins with the description of SIMD addition and subtraction algorithms and finishes with the SIMD multiplication algorithm.

---

where necessary for the sake of scientific accuracy and precision, or to provide an example of a technology for illustrative purposes, and should not be construed as either a positive or negative commentary on that product or vendor. Neither the inclusion of a product or a vendor in this presentation or accompanying printed materials, nor the omission of a product or a vendor, should be interpreted as indicating a position or opinion of that product or vendor on the part of the presenter or Bellcore.

2 Modular arithmetic

It is well known that if the modulus remains unchanged throughout the computation, as is always the case in our applications, a considerable speed-up can be gained by using the so-called *Montgomery representation* [5]. This speed-up comes from the fact that divisions are replaced by shifts to return a result in the required range. The remainder of the paper discusses numbers in Montgomery form.

Throughout this paper we fix n as an odd positive integer that will serve as the modulus. For an integer x we define the residue of x modulo n, denoted $x \bmod n$, as the smallest non-negative integer that is congruent to x modulo n. We will measure the run-time of the algorithms to be presented in number of *elementary multiplications*, where one elementary multiplication computes the $2b$-bit product of two b-bit integers, for some positive integer b that depends on the word size of the machine/processor we are using. We will assume that $b \geq 2$. The case $b = 1$ is trivial, and can be found in [5]. We are concerned with arithmetic on numbers which are much too large to be stored in one computer word. Thus we represent each number $x \bmod n$ as a sequence $x_{r-1}x_{r-2}\dots x_2x_1x_0$ of r non–negative b bit words, which means that $x \equiv \sum_{i=0}^{r-1} x_i 2^{bi} \bmod n$ and $0 \leq x_i < 2^b$ for $0 \leq i < r$, where r is the smallest integer with $2^{br} > n$. The Montgomery representation $\tilde{x}$ of an integer x is the integer $(x \cdot R) \bmod n$, where $R = 2^{br}$. Notice that because n is fixed r and R are also fixed.

Clearly, if $s = (x + y) \bmod n$, then $\tilde{s}$ is either $\tilde{x} + \tilde{y}$ or $\tilde{x} + \tilde{y} - n$, so that addition (or subtraction) of numbers in Montgomery representation is not different from ordinary modular addition (or subtraction). Multiplication, however, becomes simpler than ordinary multiplication modulo n. If $z = (x \cdot y) \bmod n$, then $\tilde{x} \cdot \tilde{y} \equiv x \cdot y \cdot R^2 \bmod n$, and therefore $\tilde{z} = (x \cdot y \cdot R) \bmod n = (\tilde{x} \cdot \tilde{y} \cdot R^{-1}) \bmod n$. It follows that the Montgomery product of two numbers in Montgomery representation can be computed using one ordinary multiplication of $\tilde{x}$ and $\tilde{y}$ followed by a division by R modulo n. To be able to easily divide by R, the Montgomery multiplication algorithm adds a multiple of n to $\tilde{x} \cdot \tilde{y}$ which is computed so that the low order br bits of the result are zero. This allows division by R as a simple shift. The result of the shift has only br bits and thus can easily be reduced to the required residue $\bmod\, n$. The details of the computation follow.

2.1 Montgomery multiplication. Let d be such that $d \cdot n \equiv -1 \bmod 2^b$ and $0 < d < 2^b$; this d is well-defined because n is odd. First, compute the ordinary product $u = \tilde{x} \cdot \tilde{y}$. Let $u = u_{2r-2}u_{2r-3}\dots u_2u_1u_0$ be the radix 2^b representation of u. Next, we iterate the following from $i = 0$ to $r - 1$: compute $t = \Big((u_i \cdot d) \bmod 2^b\Big) \cdot n$ and note that $t_0 \equiv -u_i \bmod 2^b$. Thus, we add $2^{bi} \cdot t$ to u to "zero" u_i. We increment i and continue the loop. Notice that after iteration i, the b bit words u_i, u_{i-1}, …, u_0 are all zero and the new u is congruent to the old u modulo n. Thus when the loop terminates, we replace u by u/R by shifting u over br bits to the right. We now have that either $\tilde{z} = u$ or $\tilde{z} = u - n$; this follows from the fact that the original u is $\leq (n-1)^2$, that at most $\sum_{i=0}^{r-1} 2^{bi}(2^b - 1)n = (R-1)n$ was added to u in the course of the computation, and that $((n-1)^2 + (R-1)n)/R < 2n$. This implies that the final result follows after a few subtractions and comparisons.

This multiplication algorithm can be carried out in $2r^2 + r$ elementary multiplications, plus some additions: r^2 multiplications for the computation of $\tilde{x} \cdot \tilde{y}$, plus 1 multiplication for $(u_i \cdot d) \bmod 2^b$ and r multiplications for $((u_i \cdot d) \bmod 2^b) \cdot n$ for each of the r iterations. Here we assume that d is computed once and for all at the initialization of the modular arithmetic for n using, for instance, the extended Euclidean algorithm. Notice that no divisions are needed because division with remainder by 2^b can be carried out by shifting and masking operations.

2.2 Reducing the number of multiplications. If we precompute $d \cdot n$, then the product $u_i \cdot (d \cdot n)$ can be computed using only r multiplications since the low order block of $(d \cdot n)$ is known to be -1. This means that the total number of multiplications could be reduced to $2r^2$, if we would replace u by $u + 2^{bi} \cdot u_i \cdot (d \cdot n)$ instead of $u + 2^{bi} \cdot \big((u_i \cdot d) \bmod 2^b\big) \cdot n$, for $i = 0, 1, \ldots, r-1$ in succession. The disadvantage of this approach is, however, that $\tilde{z}$ cannot be derived easily from the final u, because the final u can be as large as $2^b \cdot n$.

2.3 Remark. There are various other ways to avoid divisions in modular multiplication. For instance, if a table containing $2^{bi} \bmod n$ for $i = r, r+1, \ldots, 2r$ has been precomputed, the product of two residues modulo n can be reduced modulo n using r^2 elementary multiplications. This is less than the $r^2 + r$ elementary multiplications needed in the reduction part of Montgomery multiplication, and has the (small) advantage that one can work with the 'ordinary' representation. In practice, however, it often turns out to be slower, due to the frequent memory fetches to retrieve the elements of the table; another disadvantage that might be serious for some implementations is the fact that the method requires storage for the table.

In the next section we discuss a version of Montgomery multiplication that uses $r + 1$ processors, and that takes time $2r + 1$, plus some additions.

3 SIMD Montgomery arithmetic

Let P_0, P_1, ..., P_r be a *processor array*, with r as above, that operates in 'SIMD mode,' i.e., the P_i carry out the same instruction at the same time, but each P_i has its own data, and not every P_i has to participate. The instruction stream for the processor array is provided by a *master processor* M, which might simultaneously provide several different processor arrays with (identical) instructions. We assume that P_i can send data to P_{i+1} (for all $i < r$ simultaneously) or to P_{i-1} (for all $i > 0$ simultaneously), and that both P_0 and P_r can broadcast data to all other P_i. Furthermore, we assume that each P_i has a special flag register that can be set either to *true* or *false*, such that it takes M a small constant amount of time (independent of r and independent of the number of processor arrays governed by M) to decide if at least one of the flags is set to true. Finally, we assume that each P_i can carry out elementary multiplications, for some appropriately chosen $b > 1$, as well as additions, subtractions, and shifting and masking operations on local $2b$-bit integer variables. These last assumptions are meant to allow division with remainder by 2^b without actually dividing.

This set-up is not unrealistic. For instance, a 128×128 processor MasPar computer [4], can be thought to consist of $128 \cdot [128/(r+1)]$ disjoint processor arrays that satisfy the above conditions, for any $r < 128$ and $b \leq 32$.

In this section we describe how Montgomery arithmetic can be performed on numbers represented by the processor array P_0, P_1, ..., P_r, where the processor array represents a non-negative integer $a < 2^{b(r+1)}$ if each P_i contains a non-negative b-bit integer a_i such that $a = \sum_{i=0}^{r} a_i 2^{bi}$. The a_i's are said to represent a. Let x and y be two integers, and let $\tilde{x}$ and $\tilde{y}$ be their Montgomery representations.

3.1 SIMD Montgomery addition

Let $\tilde{x}_i$ and $\tilde{y}_i$ be non-negative b-bit integers stored at P_i, for $i = 0, 1, \ldots, r$, such that the $\tilde{x}_i$'s represent $\tilde{x}$ and the $\tilde{y}_i$'s represent $\tilde{y}$. Notice that both $\tilde{x}_r = 0$ and $\tilde{y}_r = 0$. We describe how to compute non-negative b-bit integers $\tilde{s}_i$ on P_i, for $i = 0, 1, \ldots, r$, that represent $\tilde{s}$, where $s = x + y$. We assume that P_i has local variables c_i, e_i, n_i, t_i, and u_i, where the n_i's represent the modulus n, and that these variables can contain $(b+2)$-bit integers.

Rough description: The algorithm first computes $t = \tilde{x} + \tilde{y}$ (steps 1 through 7), next $u = t - n$ (steps 8 through 14), and finally sets $\tilde{s} = t$ if $u < 0$ or $\tilde{s} = u$ if $u \geq 0$ (Steps 15 and 16).

The algorithm. Perform the steps below sequentially on all P_i simultaneously, unless indicated otherwise.

```
1    $t_i = \tilde{x}_i + \tilde{y}_i$;                      {The addition}
2    $e_i = t_i/2^b$; $c_i = 0$;                      {Compute carry bit $e_i$}
3    while some $e_i = 1$ do                    {Carry propagation loop using special flag}
4        for all $P_i$ with $i > 0$ do
5            $c_i = e_{i-1}$;                      {Send the carry to the next processor}
6        $t_i = t_i \bmod 2^b + c_i$;                {Add the carry}
7        $e_i = t_i/2^b$;                          {Compute the new carry}
8    $u_i = t_i - n_i$;                          {Compute $t - n$ in case $t$ might be $> n$}
9    $e_i = 0$;                                 {Zero the carry bit and compute carry}
     for all $P_i$ with $i < r$ do              {on all processors except the leftmost}
         if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$
10   while some $e_i = 1$ do                    {Carry propagation loop using special flag}
11       for all $P_i$ with $i > 0$ do
12           $c_i = e_{i-1}$;                      {Send the carry to the next processor}
13       $u_i = u_i - c_i$;                        {Subtract the carry}
14       $e_i = 0$;                               {Zero the carry bit and compute carry}
         for all $P_i$ with $i < r$ do          {on all processors except the leftmost}
             if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$
15   $c_i = u_r$;                               {Broadcast $u_r$ to all $P_i$}
16   if $c_i < 0$ then $\tilde{s}_i = t_i$ else $\tilde{s}_i = u_i$;  {Select $t$ or $t - n$}
```

In steps 5 and 12 we use our assumption that processors can send data to neighboring processors; in Step 15 the broadcast feature is used. It is easy to find examples where the carry propagation loops (steps 4 through 7, and steps 11 through 14), have to be repeated r times. A $\log r$ depth carry propagation tree would give a better worst case performance (if the processor array would allow implementation of such a structure). In our experience, however, our variant gives a much better average performance, because the carry propagation loops are hardly ever executed more than once. This is the case even if M governs several thousand processor arrays simultaneously. In that case the 'speed' is determined by the processor array that needs most iterations through steps 3 or 10; notice that the other processor arrays carry out the same instructions, but that the instructions have no effect on the values stored.

We subtract n from $t = \tilde{x} + \tilde{y}$ without first checking if the subtraction is actually needed; this is because it is as easy to compute $u = t - n$ as it is to test whether $t \geq n$, and given t and u it is easy to decide which of the two is the final answer. This is faster than doing a comparison to see if the subtraction has to be carried out, in particular if M governs more processor arrays. Clearly, this SIMD addition can also be used for ordinary (i.e., non-Montgomery) addition of residues modulo n.

3.2 SIMD Montgomery subtraction

Under the same assumptions as in (3.1), we describe how to compute non-negative b-bit integers $\tilde{h}_i$ on P_i, for $i = 0, 1, \ldots, r$, that represent $\tilde{h}$, where $h = x - y$.

Rough description. The algorithm first computes $u = \tilde{x} - \tilde{y}$ (steps 1 through 7) and terminates if $u \geq 0$, setting $\tilde{h} = u$ (Step 8), otherwise it computes $t = u + n$ (steps 9 through 15) and sets $\tilde{h} = t$ (Step 16).

The algorithm. Perform the steps below sequentially on all P_i simultaneously, unless indicated otherwise.

```
1     $u_i = \tilde{x}_i - \tilde{y}_i$; $c_i = 0$;              {The subtraction}
2     $e_i = 0$;                             {Zero the carry bit and compute carry}
      for all $P_i$ with $i < r$ do           {on all processors except the leftmost}
          if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$
3     while some $e_i = 1$ do                 {Carry propagation loop using special flag}
4         for all $P_i$ with $i > 0$ do
5             $c_i = e_{i-1}$;                 {Send the carry to the next processor}
6         $u_i = u_i - c_i$;                   {Add the carry}
7         $e_i = 0$;                         {Zero the carry bit and compute carry}
          for all $P_i$ with $i < r$ do       {on all processors except the leftmost}
              if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$
8     $e_i = u_r$;                           {Broadcast $u_r$ to all $P_i$ so that}
      if $e_i = 0$ then $\tilde{h}_i = u_i$ else      {the result can be decided}
9         $t_i = u_i + n_i$;                   {The addition}
10        $e_i = t_i/2^b$;                    {Compute carry bit $e_i$}
11        while some $e_i = 1$ do             {Carry propagation loop using special flag}
12            for all $P_i$ with $i > 0$ do
13                $c_i = e_{i-1}$;             {Send the carry to the next processor}
```

```
14            $t_i = t_i \bmod 2^b + c_i$;     {Add the carry}
15            $e_i = t_i/2^b$;                 {Compute the new carry}
16        $\tilde{h}_i = t_i$;                 {Store the result}
```

If M governs more processor arrays, it might be better to perform steps 9 through 15 irrespective of the value of u_r, and to choose between u_i and t_i at the very end. With only one processor array, however, the version presented above is faster. We refer for the discussion after Algorithm (3.1) for additional comments.

3.3 SIMD Montgomery multiplication

The straightforward SIMD implementation of Montgomery multiplication as it appears in (2.1) would involve $3r$ multiplications per processor using r processors. In each of r iterations, the product $x_i \cdot y_j$ is first computed (for $i = 0, 1, \ldots, r-1$ in parallel, i.e., r multiplications in the time of a single one), and then $u_0 \cdot d \bmod 2^b$ must be computed (a single multiplication). The final product in each iteration is computing $(u_0 \cdot d \bmod 2^b) \cdot n_i$ (again, r multiplications in the time of a single one) using the result of the previous product. Since the second result is needed at all processors in the final step, the second product can either be computed on a single processor and the result broadcast to all other processors or it can be computed at all processors. Since we are dealing with SIMD algorithms, there is no advantage to computing the product on a single processor; all other processors would simply be idle during this time. In any case, this straightforward method clearly leads to a major inefficiency caused by the inherent sequential nature of the process, which allows us to parallelize only $2r$ of the $2r+1$ multiplications per iteration; *one* multiplication remains unparallelizable and its execution costs a full multiplication per iteration.

Our goal is to reduce this inefficiency in the SIMD version and to get rid of this single unparallelizable multiplication per iteration. The observation in (2.2) helps lead to an efficient algorithm. By precomputing the product $d \cdot n$ and storing one word of the result at each processor in the array, the Montgomery multiplication will only take 2 multiplications per iteration. The drawback to this approach, as mentioned in (2.2), is that the result can be rather large and cannot be easily reduced to a residue mod n. The fix to this problem is to use the precomputed value only for the first $r-1$ iterations of the algorithm and use the longer 3 multiplication form for the last iteration. Using this approach the result can be shown to be less than $3n$ (cf. our analysis below), so that at most two subtractions suffice to do the final reduction modulo n.

More precisely, in the notation of (2.1), the algorithm works by first replacing $u = \tilde{x} \cdot \tilde{y}$ by $u + 2^{bi} \cdot u_i \cdot (d \cdot n)$ for $i = 0, 1, \ldots, r-2$ in succession, after which u is replaced by $u + 2^{b(r-1)} \cdot ((u_{r-1} \cdot d) \bmod 2^b) \cdot n$, where the computation of $\tilde{x} \cdot \tilde{y}$ is merged with the divisions by 2^b modulo n. So, during iteration i for $i < r-1$, we add a multiple of n to u which is less than $2^{bi}(2^b - 1)^2 n$, and during the last iteration we add a multiple of n which is less than $2^{b(r-1)}(2^b - 1)n$ because the last iteration is identical to ordinary Montgomery multiplication presented in (2.1). This implies that at most

$$\Big(\sum_{i=0}^{r-2} 2^{bi}(2^b-1)^2 n\Big) + 2^{b(r-1)}(2^b-1)n$$

is added to u in the course of the computation. Since the original u is bounded by $(n-1)^2$, the final u is bounded by

$$\Big((n-1)^2 + (2^b-1)(2^{b(r-1)}-1)n + Rn - 2^{b(r-1)}n\Big)/R < 3n.$$

The final result is therefore u, $u-n$, or $u-2n$.

Given $\tilde{x}_i$ and $\tilde{y}_i$ representing $\tilde{x}$ and $\tilde{y}$, we describe how to compute non-negative b-bit integers $\tilde{z}_i$ on P_i, for $i = 0, 1, \ldots, r$, that represent $\tilde{z}$, where $z = (x \cdot y) \bmod n$. We assume that P_i has local variables c_i, e_i, m_i, n_i, t_i, u_i, and w_i, where the n_i's represent the modulus n and the m_i's represent $(d \cdot n)/2^b$, where d as in (2.1) is assumed to exist on all P_i's. The variable w_i can contain $2b$-bit integers, for the others $b+2$ bits suffice.

Rough description. The algorithm consists of r iterations. During the jth iteration, with $0 \leq j < r$, it computes $w = \tilde{y}_j \cdot \tilde{x}$ (steps 3 and 4) and replaces the accumulator u by $u + w$ (steps 5 through 9), where u is initially set to 0 (step 1). Not all carries are propagated in this addition, but u_0 *is* normalized (steps 8 and 9). Next u's last b bits are made zero by adding an appropriate multiple of n to u. For this purpose, u_0 is first broadcast to all processors and stored in t_i there (Step 10), after which u is shifted to the right (in Step 11, where $\tilde{y}$ is also shifted to the right, to facilitate the broadcast of $\tilde{y}_j$ in Step 3), and a multiple of n is added to the shifted u. In the first $r-1$ iterations this is done in steps 14 through 20 by adding $w = t \cdot ((d \cdot n)/2^b)$ to the already shifted u (steps 15 through 17, where the carry of the part of u that was shifted out is taken care of by c_0 on P_0); after this addition, carries in u are propagated once (steps 18 through 20). In the last iteration (steps 21 through 30) a different w, namely $((t \cdot d) \bmod 2^b) \cdot (n/2^b)$, which takes one more elementary multiplication to compute, is added (steps 22 through 26), and in this last iteration, all carries in u are propagated (steps 27 through 30). In steps 31 through 37 the correct result (u, $u - n$, or $u - 2n$) is chosen.

The algorithm. Perform the steps below sequentially on all P_i simultaneously, unless indicated otherwise.

```
1   $u_i = 0$; $c_i = 0$;                        {Zero the accumulator and carry bit}
2   for $j = 0$ to $r - 1$ do                    {One iteration for each block of $y_i$}
        {Multiplication step: Compute $\tilde{x} \cdot \tilde{y}_j$ and add the result to $u$}
3       $t_i = \tilde{y}_0$                      {Broadcast the next block of $y_i$}
4       $w_i = t_i \cdot \tilde{x}_i$; $e_i = w_i/2^b$;    {Do the multiplication and compute the carry}
5       for $P_i$ with $i > 0$ set $c_i = e_{i-1}$;   {Propagate carries once for $w$}
6       $u_i = u_i + (w_i \bmod 2^b) + c_i$;     {Add new result to accumulator}
7       $e_i = u_i/2^b$;                         {Compute the new carry}
8       for $P_i$ with $i > 0$ set $c_i = e_{i-1}$;   {Propagate carries once for $u$}
9       $u_i = (u_i \bmod 2^b) + c_i$;           {Add the carry and normalize $u$}
        {Mod step: Zero $u_0$ by adding the correct multiple of $n$}
10      $t_i = u_0$;                             {Broadcast the low-order bits of $u$}
11      for all $P_i$ with $i < r$ do
            $u_i = u_{i+1}$;                     {Shift out the "zeroed" bits of $u$}
            $\tilde{y}_i = \tilde{y}_{i+1}$;     {Move the next block of $y$ to $P_0$}
12      $u_r = 0$;                               {Shift a zero into the high bits of $u$}
13      if $j < r - 1$ then                      {If this is not the last iteration}
14          if $t_0 > 0$ then                    {No correction necessary if $u_0 = 0$}
15              $w_i = t_i \cdot m_i$; $e_i = w_i/2^b$;   {$w = t_0 \cdot \left((d \cdot n)/2^b\right)$ and compute carry}
16              $c_i = e_{i-1}$ for $i > 0$, and $c_0 = 1$ for $i = 0$ only;
                    {The above multiplication would have produced a carry into $u_0$}
17              $u_i = u_i + (w_i \bmod 2^b) + c_i$; {Add the multiple of $n$ to $u$}
18          $e_i = u_i/2^b$;                     {Propagate carries once}
19          $c_i = e_{i-1}$ for $i > 0$, and $c_0 = 0$ for $i = 0$ only;
20          $u_i = (u_i \bmod 2^b) + c_i$;
21      else if $t_0 > 0$ then                   {Last iteration, check for $u_0 = 0$}
22          $e_i = n_{i+1}$ for $i < r$, and $e_r = 0$ for $i = 0$ only;
23          $w_i = \left((t_i \cdot d) \bmod 2^b\right) \cdot e_i$;   {$w = \left((t_0 \cdot d) \bmod 2^b\right) \cdot (n/2^b)$ }
24          $e_i = w_i/2^b$;                     {Propagate carries once}
25          $c_i = e_{i-1}$ for $i > 0$;
```

```
26          $u_i = u_i + (w_i \bmod 2^b) + c_i$;    {Add the proper multiple of $n$ and the carry}
27  $e_i = u_i/2^b$;                               {Compute the carry and}
28  while some $e_i = 1$ do                        {propagate completely using special flag}
29      $c_i = e_{i-1}$ for $i > 0$;
30      $u_i = (u_i \bmod 2^b) + c_i$;  $e_i = u_i/2^b$;
    {Choose the proper value: $u$, $u - n$, or $u - 2n$}
31  $t_i = u_i - n_i$;                             {Subtract $n$ and propagate carries}
32  $e_i = 0$; if $t_i < 0$ and $i < r$ then $e_i = 1$, $t_i = t_i + 2^b$;
33  while some $e_i = 1$ do
34      $c_i = e_{i-1}$ for $i > 0$;
35      $t_i = t_i - c_i$;
36      $e_i = 0$; if $t_i < 0$ and $i < r$ then $e_i = 1$, $t_i = t_i + 2^b$;
37  if $t_r < 0$ then $\tilde{z}_i = u_i$                {If $t < 0$ then $u < n$ is the proper result}
    else $u_i = t_i$ and repeat the subtraction in lines 31–37;
```

There are two elementary multiplications per P_i in the first $r - 1$ iterations (in steps 4 and 15), and three in the last iteration (one in Step 4 and two in Step 23). This implies that the algorithm takes time $2r + 1$, plus some additions. In the computation of $t_i \cdot d$ in Step 23, the value of t_i is identical on all P_i; computing this product on only one processor, and broadcasting the result to the others, does not save any time in our SIMD architecture. Notice that the full $2b$-bit product $t_i \cdot d$ is not needed in Step 23, but that its least significant b bits suffice. This might make this single extra multiplication slightly faster.

4 Conclusion

This paper shows how to avoid a direct but inefficient parallel implementation of a well known algorithm through a reformulation of the algorithm and its invariants. In our particular case, an efficient solution can be attained by relaxing the invariant for the first part of the algorithm and returning to a tighter invariant for a final stage in order to produce a correct result. Although we have only discussed the application of this general technique to modular multiplication, we suspect that similar ideas may be used to enhance the SIMD parallelizability of other algorithms.

The modular multiplication algorithm itself has applications to cryptanalyis and parallel RSA systems as mentioned earlier, and we have applied the Montgomery arithmetic from

(3.1), (3.2), and (3.3) in our massively parallel implementation of the elliptic curve factoring algorithm on a 16K processor MasPar computer. The efficiency of the implementation on this machine is helped by the fact that all relevant values can be kept in registers, which is not true for the single processor variants of these operations. The result is that we achieve a direct speed up; with k processors, our algorithm runs k times faster than the single processor algorithm.

In the elliptic curve method the odd composite number to be factored is used as the modulus throughout the algorithm, which makes our SIMD algorithms applicable. We used $b = 30$, so that for n in the range $[2^{300}, 2^{330})$, for instance, we got $r = 11$ and therefore $128 \cdot [128/(11+1)] = 1280$ processor arrays. This implies that we can run 1280 elliptic curve trials in parallel for such n. Our elliptic curve implementation has broken factoring records by being the first to find a 40 digit factor using elliptic curves, as reported in [1]. This implementation is probably the fastest known method for factoring numbers that have factors of less than 40 digits. For a number that is the product of two roughly equal size primes, then the parallel implementation of the quadratic sieve given in [2] is efficient for numbers in the 80–120 digit range.

References

[1] B. Dixon, A. K. Lenstra, Massively parallel elliptic curve factoring, Advances in Cryptology, Eurocrypt '92,*Lecture Notes in Comput. Sci.* **658** (1993), 183–193.

[2] B. Dixon, A. K. Lenstra, Factoring integers using SIMD sieves, Advances in Cryptology, Eurocrypt '93,*Lecture Notes in Comput. Sci. To appear.*

[3] H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Ann. of Math.* **126** (1987), 649–673.

[4] *MasPar MP-1 principles of operation*, MasPar Computer Corporation, Sunnyvale, CA, 1989.

[5] P. L. Montgomery, Modular multiplication without trial division, *Math. Comp* **44** (1985), 519–521.

[6] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signitures and public–key cryptosystems, *Communications of the ACM*, **21**,2 (Feb. 1978).