Massively parallel elliptic curve factoring

B. Dixon*, A. K. Lenstra**

Abstract. We describe our massively parallel implementations of the elliptic curve factoring method. One of our implementations is based on a new systolic version of Montgomery multiplication.

1. Introduction

The study of theoretical and practical aspects of factoring algorithms is of continuing interest for the analysis of various well known public-key cryptosystems. In this paper we describe two massively parallel implementations of the elliptic curve factoring method [5].

Usually one distinguishes two types of factoring algorithms, the *general purpose algorithms* whose expected run time depends solely on the size of the number n being factored, and the *special purpose algorithms* whose expected run time also depends on properties of the (unknown) factors of n. Examples of special purpose algorithms are *trial division*, *Pollard's rho method*, *Pollard's $p-1$ method*, and the *elliptic curve method*. To find a factor p, trial division needs time approximately linear in p, Pollard's rho method is approximately linear in $\sqrt{p}$, and Pollard's $p-1$ method is approximately linear in the largest prime factor of $p-1$. Thus, in the worst case all these methods take exponential time. The elliptic curve method takes expected time $O((\log n)^2 L_p[\sqrt{2}])$ to find p, where

$$L_x[a] = \exp((a+o(1))\sqrt{\log x \log\log x}),$$

for $x \to \infty$. It follows that in the worst case $p \approx \sqrt{n}$ the method takes expected time $L_n[1]$, which is subexponential in n. Because their run time depends so strongly on the size of smallest factor of n, and only polynomially on the size of n itself, the performance of special purpose methods is usually measured by the size of the prime discovered.

From a security point of view, the above exponential time methods are not something to worry about, except that one sometimes takes the precaution to construct n such that Pollard's $p-1$ method will not be unexpectedly lucky. Such precautions cannot be taken against the elliptic curve method. Whereas in Pollard's $p-1$ method p will be discovered if $p-1$ has only small factors, the elliptic curve method will find p if the method is so lucky

* Department of Computer Science, Princeton University, Princeton, NJ 08544, U.S.A, email: bdd@princeton.edu.

** Bellcore, rm 2Q334, 445 South Street, Morristown, NJ 07960, U.S.A, email: lenstra@bellcore.com.

to hit upon a number *close to* p that has only small factors. Because the method consists of many independent trials, there is always the possibility that p will be discovered.

In this paper we consider two implementations of the elliptic curve method on a particular type of massively parallel computer, so-called *single instruction, multiple data* (SIMD) machines. As far as we know SIMD elliptic curve implementations have not been considered before, although massive parallelism is not new to this area. In [4], for instance, a distributed network implementation of the elliptic curve method is described, which may be viewed as a large scale *multiple instruction, multiple data* (MIMD) approach. We will see that SIMD machines, even though they are relatively cheap compared to MIMD machines, nevertheless can achieve an impressive elliptic curve performance*.

Using one of our implementations we have been able, for the first time for the elliptic curve method, to find a 40 digit factor**. The previous elliptic curve record was 38 digits, which occurred three times as far as we know. On the negative side, the elliptic curve method has also missed many smaller factors, in the 35 digit range, even after serious efforts. Using the run time estimates given above, one finds that finding a 60 digit factor can be expected to be more than 3000 times more difficult than finding a 40 digit factor. Given how much computing time has been invested to this date in the elliptic curve method, it seems safe to say that it is unlikely that with present day technology we will ever be able to discover factors of 60 or more digits using the elliptic curve method. Estimates of this type are, for instance, useful for the design of cryptosystems based on discrete logarithms modulo a composite modulus, like [8]. Finding 50 digit factors with the elliptic curve method is approximately 70 times more difficult than finding 40 digit factors; it is therefore not inconceivable that such factors might be found using more powerful machines.

The remainder of this paper is organized as follows. In Section 2 a short description of the SIMD machine that we use for our implementations is given. A very superficial

---

* 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 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.

** In the mean time Dave Rusin from the University of Northern Illinois has found a 42 digit factor, using Peter Montgomery's elliptic curve program.

description of the elliptic curve method, and our first SIMD implementation are presented in Section 3. Section 4 describes our block-wise SIMD multi-precision modular integer arithmetic. The second elliptic curve implementation, which is based on this block-wise arithmetic, is presented in Section 5, along with some results concerning random generators with very long period, that were constructed using this second elliptic curve implementation.

2. The hardware

This section contains a short overview of the 16K MasPar, the massively parallel computer that we have used for the implementations to be described in this paper. Our description is *very* incomplete, and only covers those aspects of the machine that are referred to in the following sections. For a complete description of the MasPar we refer to the manuals, like [7].

The 16K MasPar is a SIMD machine, consisting of, roughly, a *front end*, an *array control unit* (ACU), and a 128×128 array of *processing elements* (PE array). Masks, or conditional statements, can be used to select and change a subset of active processors in the PE array, the so-called *active set*. The fact that it is a SIMD machine means that instructions are carried out sequentially, and that instructions involving parallel data are executed simultaneously by all processors in the active set, while the other processors in the PE array are idle. The instructions involving singular (i.e., non-parallel) data are executed either on the front end or on the ACU; for the purposes of our descriptions the front end and the ACU play the same role.

According to our rough measurements, each PE can carry out approximately $2 \cdot 10^5$ additions on 32 bit integers per second, and can be regarded as a 0.2 MIPS processor. Furthermore, each PE has 64KBytes of memory, which implies that the entire PE array has 1GByte of memory. Each processor can communicate with its north, northeast, east, southeast, south, southwest, west, and northwest neighbor, with toroidal wraparound. Actually, a processor can send data to a processor at any distance in one of these eight directions, with the possibility that all processors that lie in between also get a copy of the transmitted data. There is also a global router that allows any processor to communicate with any other processor, but we never needed it.

Each job has a size between 8K and 64K, reflecting the amount of PE-memory it uses. Only those jobs which together occupy at most 64K are scheduled in a round robin fashion, giving each job 10 seconds before it is preempted, while the others must wait. This means

that jobs are never swapped out of PE-memory.

For our implementations we used the MasPar Parallel Application Language MPL, which is from our perspective, a simple extension of C.

3. Elliptic curve method

In this section we briefly describe our first SIMD implementation of the elliptic curve method. For a detailed description of the elliptic curve method, hints for its implementation and parameter choices, we refer to [3; 5; 10; 11]. For our purposes it suffices to know that the elliptic curve method consists of a number of independent trials. For each trial an elliptic curve E modulo n and a point x in a group G related to E are randomly selected. The group operation in G, which we will write multiplicatively, consists of several additions, subtractions, multiplications, and inversions of integers modulo n, and can be carried out in time $O((\log n)^2)$ per operation. Clearly, the group operation breaks down if some integer y for which $\gcd(n, y) \neq 1$ has to be inverted modulo n in the course of a group operation; in this case a factor of n has been found.

This is exploited as follows. Using the group operation, the point x is raised to a huge power k consisting of the product of all prime powers below a certain bound B_1. The trial is lucky if this computation cannot be completed because the group operation breaks down, since in this case a factor of n has been found. The trial fails if $x^k \in G$ has been computed successfully. If p is n's smallest prime divisor and $B_1 = L_p[\sqrt{1/2}]$, then each trial has probability $L_p[-\sqrt{1/2}]$ to factor n, as explained in [3; 5]. This implies that the number of independent trials needed to factor n can be expected to be $L_p[\sqrt{1/2}]$. One trial takes time $O((\log n)^2 B_1)$, from which the total expected run time $O((\log n)^2 L_p[\sqrt{2}])$ follows.

The computation of $x^k \in G$ is usually referred to as the *first phase* of the algorithm. In the *second phase* $x^{kq} \in G$ is computed for all primes q in $[B_1, B_2]$ for some bound B_2. This requires approximately an additional $\pi(B_2) - \pi(B_1)$ group operations, where $\pi(b)$ denotes the number of primes $\leq b$. It appears to be close to optimal to select B_1 and B_2 in such a way that the two phases take approximately the same amount of time, which makes B_2 an order of magnitude larger than B_1.

The optimal parameter choice for the elliptic curve method depends on the unknown factor p of n, and therefore cannot be known beforehand. A common approach is to do a few trials with fairly low B_1 (and corresponding B_2), upon failure a few trials with slightly larger B_1, and so on, until either the number is factored or the factoring attempt is

aborted. The expected amount of work to find a certain factor usually varies only slightly: the success probability for t trials with bound B_1 is not dramatically different from the success probability for $c \cdot t$ trials with bound B_1/c, but only for c in a limited range, say $1/2 < c < 2$. The optimal parameter choices depend on the implementation as well. For instance, for the implementation described in [4], the optimal choices are $t = 300$ and $B_1 = 65000$ to find 25 digit factors with a 60% success probability, $t = 950$, $B_1 = 275000$ for 30 digits, and $t = 2300$, $B_1 = 1100000$ for 35 digits.

The trials of the elliptic curve method are independent, but any number of them can be carried out simultaneously. Since the sequence of operations involved in the computation of $x^k \in G$ depends only on the value of k, and *not* on the actual data to which the operations are applied, t elliptic curve trials can be carried out in parallel on a t processor SIMD machine. An exception occurs if one of the trials factors n, but in that case the process can be terminated. Using this elementary approach we ran 16K trials in parallel on the 16K MasPar, with each of the 16K PE's working on its own elliptic curve, uniquely generated using a single random seed. For this purpose the multi precision integer arithmetic used in [4] was adapted to the MasPar, in such a way that each PE operates independently on its own extended precision integers, simultaneous with the other PE's (in the active set). Thus we can carry out 16K elementary operations ($+$, $-$, $*$, quotient and remainder) on extended precision integers in parallel, without interprocessor communication.

Because inversions modulo n are slow, particularly so on a SIMD machine where the cost is determined by the processor that needs the most iterations, we kept track of the numerators and denominators (modulo n) of the group elements, without performing the inversions. But since these inversions are supposed to lead to the factorization of n, they cannot be avoided entirely. At regular intervals we therefore computed the product modulo n of all denominators (using 14 (i.e., $\log_2(16K)$) multiplications modulo n on the PE array), and computed the greatest common divisor g of the resulting product and n on the (much faster) front end. If g turns out to be > 1, the PE's can be inspected to see how many of them found a factor, or to refine the factorization if g is not prime. The latter usually happened only in the presence of several very small factors (up to 10 digits); larger factors are usually found on only one PE.

Addition and subtraction modulo n are easily made efficient in SIMD mode, but multiplication modulo n is more problematic. This is caused by the remainder computation modulo n, where the instruction stream is more dependent on the values involved. We

used the so-called *Montgomery representation* throughout our program, because it allows a modular multiplication that is oblivious of the data involved, without affecting the addition or the subtraction; see Section 4 or [9].

Although these and various other improvements considerably enhanced the performance of our first implementation, there was not much we could do about the fact that a 0.2 MIPS PE is a fairly slow processor: it took about a day to complete 16K curves with $B_1 = 50000$ and $B_2 = 10^6$ for a 100 digit n. With these parameters one can almost guarantee that all factors up to 28 digits will be found, but this is not an optimal parameter choice. To find 28 digit factors far fewer curves with much larger B_1 would be better, whereas the optimal B_1 that corresponds to 16K trials is more than 10^7. The latter would be a good choice if one wants to look for 45 digit factors, but it would require an inordinate amount of time on this machine.

Consequently, this parallel elliptic curve program is not ideally suited for the present MasPar. For future generations of SIMD machines, however, our program might turn out to be useful: if future PE's run at speeds comparable to that of current workstations, then 16K parallel trials with matching bounds could be processed in at most a few days.

Given the current PE speed, the only way to get a better parallel elliptic curve program on the 16K MasPar seems to be to divide the work per curve over r PE's, for some r. This would decrease the number of trials by r and hopefully increase the speed per curve by the same factor, thus allowing parameter choices which are closer to optimal for factors in the 30 digit range. We achieved this by designing a multi precision integer arithmetic that is entirely different from the one mentioned above. This arithmetic will be described in the next section.

4. Block-wise modular arithmetic

In this section we describe an alternative integer arithmetic for SIMD machines, that essentially reconfigures the machine into a machine with fewer but faster processors, at least for arithmetic modulo a fixed integer n.

Let b be some small integer such that arithmetic operations on b bit integers can be carried out efficiently on a PE. Because multiplication of 32 bit integers with a 64 bit result is the most efficient multiplication that a PE can perform, we used $b = 30$, so that we could also add without overflow problems. Let r be the smallest integer with $2^{b \cdot r} > n$, where n is the odd number being factored. Each of the 128 rows of 128 PE's is divided into $u = \lfloor 128/(r+1) \rfloor$ disjoint *blocks* of $r+1$ consecutive PE's, and $128 - (r+1) \cdot u$ idle

PE's. Thus, there are $128 * u$ blocks. The active set consists of the PE's that are contained in a block.

Suppose that the consecutive PE's in a block are numbered from 0 to r. If the ith PE contains a b bit integer $v_i \geq 0$, then $v_0, v_1, \ldots, v_r$ together represent the number $v = \sum_{i=0}^{r} v_i 2^{b \cdot i}$. Since we use the integers in $\{0, 1, \ldots, n-1\}$ to represent residue classes modulo n, we usually have that $v < n$ and therefore $v_r = 0$. This extra PE is used in the multiplication modulo n. Let $v = \sum_{i=0}^{r} v_i 2^{b \cdot i}$ and $w = \sum_{i=0}^{r} w_i 2^{b \cdot i}$ be two integers modulo n. To compute the sum $s = v + w$ the $r+1$ PE's compute $\bar{s}_i = v_i + w_i$ and $c_i = \bar{s}_i / 2^b$, next c_i is sent by the ith PE to the neighboring $(i+1)$th PE, and finally s_i is computed as $\bar{s}_i - c_i \cdot 2^b + c_{i-1}$. In the unlikely event that one of the s_i is still $\geq 2^b$ the carry propagation is repeated until all s_i are $< 2^b$. Here we note that it requires only one fast instruction on the MasPar to check if there are still any carries to be propagated. To complete the addition modulo n, we simply subtract n from s, using a similar technique; if $s - n$ is negative then s is the final outcome, otherwise it is $s - n$.

Examples that require r carry propagation steps are easy to construct, and a depth $O(\log r)$ carry propagation tree would give a better worst case performance. Our simplistic approach, however, works on average much faster because the second carry propagation step hardly ever occurs.

Multiplication modulo n within blocks is more complicated. As in the first elliptic curve implementation we used the Montgomery representation to avoid divisions. Let $R = 2^{b \cdot r}$, the smallest power of 2^b larger than n. The Montgomery representation $\tilde{x}$ of an integer x modulo n is the integer $x \cdot R \bmod n \in \{0, 1, \ldots, n-1\}$. Addition and subtraction of numbers in Montgomery representation is not different from ordinary addition and subtraction modulo n, and is carried out as described above. Multiplication, however, becomes much simpler than ordinary multiplication modulo n. Let z be such that $z \equiv x \cdot y \bmod n$. Then $\tilde{z}$ equals $\tilde{x} \cdot \tilde{y} / R \bmod n$. This $\tilde{z}$ can be computed efficiently as follows. First compute $v = \tilde{x} \cdot \tilde{y}$. Let $v = \sum_{i=0}^{2r} v_i 2^{b \cdot i}$ and d be such that $d \cdot n \equiv -1 \bmod 2^b$, which is well-defined because n is odd. Next, for $i = 0, 1, \ldots, r-1$ in succession replace v by $v + 2^{b \cdot i} \cdot n \cdot (v_i \cdot d \bmod 2^b)$, where the v_i for $i > 0$ are the radix 2^b digits of the v that was computed in the previous iteration. Notice that after iteration j the new v_j is zero, and that the new v is congruent to the old v modulo n. Consequently, the resulting v_0 through v_{r-1} are all zero, and the division by R can be carried out by simply shifting the resulting v to the right. The result might be $\geq n$, in which case it suffices to subtract n once to

make it $< n$. Montgomery multiplication can be carried out using $2r^2 + r$ multiplications on b-bit integers.

By merging the iterations for the (ordinary) multiplication of $\tilde{x}$ and $\tilde{y}$, and the division modulo n by R, the multiplication of numbers in Montgomery representation can be done quite easily in a block of PE's. Straightforward application of the above algorithm leads to a block-wise modular multiplication that can be made to fit in blocks of only r consecutive PE's, with 3 multiplications per iteration: two on all PE's in the block (on different data), but one (the computation of $v_i \cdot d \bmod 2^b$) that operates on identical data for all PE's in the block or that could be carried out by only one PE and sent to the others. This is inefficient, because it requires time for $3r$ multiplications on r PE's in parallel, instead of the $2r + 1 = (2r^2 + 1)/r$ multiplications we hoped for. As shown in [2] the $3r$ can be improved to $2r + 2$, giving a ratio which is close to optimal. Using this method we got an acceptable speed for the modular multiplication: for a 95 digit n one modular multiplication takes about 0.003 seconds, and because $r = 11$ implies $[128/(11 + 1)] = 10$ blocks per row, 1280 of these multiplications can be carried out simultaneously.

An additional advantage of the block-wise arithmetic is that all relevant values can be kept in registers, so that costly memory fetches and stores can be avoided. A detailed description can be found [2].

5. A second implementation and results

Incorporation of the modular arithmetic from the previous section into our first SIMD elliptic curve implementation led to a second version of our program that runs one curve per block instead of one curve per processing element. This implies that the number of trials depends on the size of n. For a 95 digit n we get 1280 trials, for 80 digits r becomes 9, and the number of trials goes up to 1536.

We used the following elementary method to lower the number of group operations needed to compute the kth power of the point x on the curve. According to the definition of k given in Section 3, we have $k = \prod_{i=1}^{l} q_i$, where $\{q_1, q_2, \ldots, q_l\}$ is the set of prime powers $< B_1$. The usual way of computing x^k is to first raise x to the power q_1, next raise the result to the power q_2, etc., until all q_i have been processed. Let for some integer m the *weight* $w(m)$ be defined as the number of ones in the binary representation of m. If ordinary repeated squaring and multiplication is used for the exponentiation, then the cost of the computation of $x^k \in G$ is $\sum_{i=1}^{l} \lfloor \log_2 q_i \rfloor$ squarings in G and $\sum_{i=1}^{l} (w(q_i) - 1)$ multiplications in G.

Let $S = S_1 \cup S_2 \cup \ldots S_s$ be a partition of $\{1, 2, \ldots, l\}$, and let $\bar{q}_j = \prod_{i \in S_j} q_i$. Clearly, x^k can also be computed by first raising x to the power $\bar{q}_1$, next the result to the power $\bar{q}_2$, etc., up to $\bar{q}_s$. The number of squarings in G needed for this computation is approximately the same as the number of squarings given above. The number of multiplications in G, however, can be made substantially smaller by choosing a partition S for which $\sum_{j=1}^{r} w(\bar{q}_j)$ is small. Finding the best partition with respect to this metric is in general a hard problem. In practice we will have to do with what can be found in a reasonable amount of time. Using a simple greedy algorithm (and $B_1 = 100000$) we found a partition in subsets of cardinality at most two that had approximately half the original weight; Bill Cook used this solution to derive an optimal partition under the same restrictions, but the resulting weight was not significantly lower. Next, we considered subsets of cardinality at most three. This resulted in approximately a third of the original weight. Given how much time it took us to find these triples, this is probably the best we may hope to achieve. The triples were found by means of a greedy-type algorithm on the 16K MasPar. To make the run times acceptable we processed the primes in intervals. More precisely, for each $i \in \{1, 2, \ldots, 20\}$ we determined partitions into triples of the prime powers $< 2 \cdot 10^6$ for the primes in the interval $[(i-1) \cdot 10^5, i \cdot 10^5]$. This separation into intervals is also useful because it allows a choice of bounds. The total run time of the program was reduced by 18% using this technique. To give an example, the primes 1028107, 1030639, and 1097101 have weights 10, 16, and 11, but their product has weight 8; in binary

$$11111011000000001011 \cdot 11111011100111101111 \cdot 10000101111010001101 =$$

$$10000001000100000010000001000000000000010000010000000000000001.$$

This example is remarkable but by no means exceptional.

For n with $r = 11$ it takes about 34 hours to complete 1280 elliptic curve trials with $B_1 = 10^6$ and $B_2 = 2 \cdot 10^7$. For n with $r = 12$ it takes approximately $34 * 12/11$ hours to complete 1152 trials with the same bounds, and other timings can be derived similarly. Consequently, this second elliptic curve program allows a much more balanced choice of the parameters for a search of factors in the 30 digit range. We have used the program to factor various numbers from the list of composite numbers from [1] and many numbers from the 'Partition List' of the 'RSA Data Security Factoring Challenge.' To date the

largest factor we have found has 40 digits, which was a new elliptic curve factoring record:

$$\begin{aligned} p(11279) =& 2^6 \cdot 5 \cdot 8418\,735626\,949973\,617503 \cdot \\ & 1232\,079689\,567662\,686148\,201863\,995544\,247703 \cdot \\ & 78\,507734\,924917\,342278\,622201\,969372\,653526\,213641\,483293. \end{aligned}$$

The number factored was the 89 digit product of the last two factors, and p(11279) denotes the 11279th partition number. The factorization was found by one of 1408 trials with $B_1 = 10^6$ after $q = 1208269$ in the second phase, which means that we have been quite lucky since finding the factor at this point happens with probability 0.7%*.

We found another, unexpected, application of our SIMD elliptic curve program in [6]. In this paper a new class of pseudo random generators is introduced with the remarkable property that they have provably long periods. Such generators are based on integers b, r, and s such that $m = b^r \pm b^s \pm 1$ is prime, and have a period equal to the order of b modulo m. The size of b depends on the type of pseudo random number generator that is needed; typically it varies between 16 and 64 bits. Consequently, primes m as above can fairly easily be found, if r and s are not excessively large (typically they vary between 10 and 50). Establishing the order of b, however, requires factoring $m - 1$, which might be (and usually is) hard. We found, among others, that $b = 2^{53} - 1052$ has order $(m-1)/2$ modulo the 511 digit prime $m = b^{32} + b^{16} - 1$, where the factorization of the factor $b^8 + 1$ of $m - 1$ was found using our SIMD elliptic curve implementation:

$$\begin{aligned} b^8 + 1 =& 257 \cdot 16673 \cdot 275422\,758002\,613571\,762817 \cdot \\ & 29\,464604\,796724\,055573\,394001 \cdot 65726\,748717\,597552\,856331\,429857 \cdot \\ & 18\,955214\,139747\,587789\,390113\,133069\,907813\,629489. \end{aligned}$$

This was one of the few examples where the global inversion on the front end produced a composite factor: one block had found the 24 digit factor and simultaneously an other block had found the 26 digit factor. This b and m provide an attractive way to generate double-precision floating point pseudo random numbers using just addition.

---

* This corresponds to a 0.0006% probability of success per curve. For comparison, the 42 digit factor referred to above, was found with $B_1 = 2 \cdot 10^6$, and $B_2 \approx 10^8$, which leads to a 0.003% probability of success per curve. The number of curves used in that factorization is unknown to us.

References

1. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers*, second edition, Contemporary Mathematics **22**, Amer. Math. Soc., Providence, 1988.
2. B. Dixon, A. K. Lenstra, *Systolic Montgomery multiplication*, in preparation.
3. A. K. Lenstra, H. W. Lenstra, Jr., *Algorithms in number theory*, Chapter 12 in: J. van Leeuwen (ed.), *Handbook of theoretical computer science*, Volume A, *Algorithms and complexity*, Elsevier, Amsterdam, 1990.
4. A. K. Lenstra, M. S. Manasse, *Factoring by electronic mail*, Advances in Cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci. **434** (1990), 355–371.
5. H. W. Lenstra, Jr., *Factoring integers with elliptic curves*, Ann. of Math. **126** (1987), 649–673.
6. G. Marsaglia, A. Zaman, *A new class of random number generators*, Ann. of Appl. Prob. 1 (1991), 462–480.
7. *MasPar MP-1 principles of operation*, MasPar Computer Corporation, Sunnyvale, CA, 1989.
8. U.M. Maurer, Y. Yacobi, *Non-interactive public key cryptography*, Advances in Cryptology, Eurocrypt '91, Lecture Notes in Comput. Sci., to appear.
9. P. L. Montgomery, *Modular multiplication without trial division*, Math. Comp. **44** (1985), 519–521.
10. P. L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Math. Comp. **48** (1987), 243–264.
11. R. D. Silverman, S. W. Wagstaff, Jr., *A practical analysis of the elliptic curve factoring algorithm*, manuscript.