Massively parallel computing and factoring

Arjen K. Lenstra

Bellcore, 445 South Street, Morristown, NJ 07960, U.S.A; email: lenstra@flash.bellcore.com

1. Introduction

Let n be an odd positive composite integer that is not a prime power. It is well known that it is usually easy to prove n 's compositeness without revealing any information about its factors [7]. It is widely believed that factoring n is, in general, a hard problem. Indeed, if n has 200 decimal digits, and only prime factors of at least 60 digits, then factoring n is beyond the capabilities of current algorithms and technology. On the other hand, if n has 'only' 100 digits, it can quite easily be factored; 150 digits lie in the gray area that would require a major effort, but that should not be considered to be impossible.

Knowing the limits of our factoring capabilities is important for various reasons. A favorite example is the security of factoring based cryptosystems, which is based on our inability to solve the factoring problem. There are also cryptosystems, however, which are not only based on this inability, but which also depend on our ability to solve certain instances of the related *discrete logarithm problem*. Therefore we not only need to know what we *cannot* do, but we also need practical and efficient algorithms for what we supposedly *can* do. In this paper we consider how some of these issues are affected by a particular type of massively parallel computers, so-called *single instruction, multiple data* (SIMD) machines.

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 $x \rightarrow \infty$. 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 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. To evaluate how worrisome this might be for factoring based cryptosystems, implementations of the elliptic curve method have to be studied, and the consequences of all possible machine architectures have to be taken into account. In Section 2 we discuss our massively parallel implementation of the elliptic curve method. Using this implementation 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. Given how much computing time has been invested to this date in the elliptic curve method, and how fast its expected run time increases with growing p , it seems safe to say that it is unlikely that with present day technology we will ever be able to discover factors of 50 or more digits using the elliptic curve method.

The performance of general purpose factoring algorithms is not measured by the size of the factor discovered, but by the size of the numbers factored. Furthermore, unlike the elliptic curve method, their achievements can quite accurately be predicted, in the following sense. Suppose that a particular implementation of the double large prime multiple polynomial variation of the quadratic sieve algorithm (ppmpqs), currently the most practical general purpose factoring algorithm, takes time x to factor n . Then any number m which is approximately in the same digit range as n , can be factored in time $O(x \cdot L_m[1]/L_n[1])$, where the constant in the $O()$ is small and the $o(1)$'s in L_m and L_n are set to zero. An indication for the security margin offered by a 155 digit (512 bit) composite public key can immediately be obtained: factoring it is only about $L_{10^{155}}[1]/L_{10^{116}}[1] \approx 1300$ times harder than what can currently be achieved, because the current general purpose factoring record stands at 116 digits [11].

In Section 3 we discuss one aspect of this 116 digit record, the massively parallel solution of a huge sparse system of equations modulo 2. The method described there not only allowed us to achieve this result, it also shows that much larger systems can in principle be dealt with. As a consequence, the limits of our factoring capabilities will probably not be set by the maximum size of the system that can still be dealt with, at least not in the near future. In particular, it follows that the system that would be obtained during the ppmpqs factorization of a 155 digit number could be handled, though it would require a major effort. Another consequence of this work is that the generally much bigger systems that follow from large number field sieve factorizations can conveniently be solved [8]. Finally, this work is useful for the study of the feasibility of various cryptosystems that are based on the solution of certain medium-sized discrete logarithm problems [14; 17], because algorithms for the computation of discrete logarithms require the solution of large sparse linear systems.

Solving these systems is usually referred to as the 'second stage' of the factoring (or the discrete logarithm) algorithm. The 'first stage,' during which the system is constructed, is asymptotically just as hard as the second stage, but in practice by far the most time

consuming step. In [10] it was shown how vast amounts of computing time for this first step can be obtained from otherwise idle workstations all over the world; the 116 digit number referred to above was factored with this approach. Using an entirely different approach, we discuss in Section 4 some experiments that we have carried out with a massively parallel implementation of the first stage of ppmpqs. On current massively parallel machines this implementation will not be able to compete with the network approach, but it looks more promising than we had expected. It is likely that similar implementations on future generations of massively parallel machines will produce much more impressive results, also compared to the more cumbersome network approach.

We conclude this introductory section with a short overview of the 16K MasPar, the massively parallel computer that we have used for the implementations that will be described in sections 2 through 4. 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 [13].

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.

Two distinct advantages of the 16K MasPar compared to other massively parallel machines or supercomputers, are its price, and the fact that programming it is relatively easy. For our implementations we used the MasPar Parallel Application Language MPL, which is, from our perspective, a simple extension of C.

2. Elliptic curve method

In this section we discuss two massively parallel implementations of the elliptic curve factoring method. This work was done with Brandon Dixon. For a detailed description of the elliptic curve method, hints for its implementation and parameter choices, we refer to [7; 12; 16; 20]. 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, if some integer y for which $\gcd(n, y) \neq 1$ has to be inverted modulo n in the course of a group operation, then the group operation breaks down, and a factor of n has been found.

This is employed 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 that 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 , so 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.

This 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 . A common approach therefore is to do a few trials with fairly low B_1 (and corresponding B_2), and 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 , for say $1/2 < c < 2$. The optimal parameter choices depend on the implementation. For instance, for the implementation described in [10], we got $t = 300$ and $B_1 = 65000$ as the optimal choice 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 supposed to be 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 by 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, randomly generated using its own unique random seed. For this purpose the multi precision integer arithmetic used in [10] was adapted to the MasPar, in such a way that each PE independently operates 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

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), more interesting factors are usually found on only one PE.

Concerning the other operations involved in the group operation, 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 depends more on the values involved. For that reason 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 below or [15].

Although these and various other improvements considerably enhanced the performance of our initial program, there was not much we could do about the fact that a 0.2 MIPS PE is a fairly slow processor for these types of operations: 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 by no means 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 a 16K MasPar.

Consequently, this parallel elliptic curve program is not ideally suited for our 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 a number of PE's. This would decrease the number of trials and increase the speed per curve, 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. A rough description of this arithmetic follows.

Let b be some small integer such that arithmetic operations on b bit integers can be carried out efficiently on a PE. Because MPL, the MasPar parallel language, supports multiplication of 32 bit integers with a 64 bit result we used $b = 30$, so that we could also add without overflow problems. Lower choices would lead to fewer but possibly faster trials. 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 $128 \cdot u$ 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 i th PE contains a b bit integer $v_i \geq 0$, then $v_0, v_1, \dots, v_r$ together represent the number $v = \sum_{i=0}^r v_i 2^{b \cdot i}$. Since we use the integers in $\{0, 1, \dots, n-1\}$ 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 i th 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 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 other elliptic curve program 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 $x \cdot R \bmod n \in \{0, 1, \dots, n-1\}$. 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 efficiently computed 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 $i = 0, 1, \dots, r-1$ 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 $< n$.

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 quite easily be done 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. Clearly, this is inefficient. We have chosen for a variation, where we precompute the integer $(d \cdot n) / 2^b$, and distribute its r radix 2^b digits over the block. This allows us to reduce the number of multiplications to two for the first $r-2$ iterations. For the last two iterations the original approach is used, to keep the result within reasonable bounds. The disadvantage of this variation is that we need $r+1$ instead of r PE's per block, and that it is conceivable that we need two subtractions at the end. Apparently the probability that this second subtraction is needed is very small: at the time of writing, after more than 10^{12} modular multiplications, it has not yet occurred.

Another disadvantage of the block-wise multi precision integer arithmetic is the need for interprocessor communication. An advantage, however, is that all relevant values can be kept in registers, so that costly memory fetches and stores can be avoided. 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 $\lceil 128 / (11 + 1) \rceil = 10$ blocks per row, 1280 of these multiplications can be carried out simultaneously. Notice that the number of trials in the implementation of the elliptic curve method based on this block-wise arithmetic 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.

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 $34 \cdot 12/11$ 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 better 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:

$$p(11279) = 2^6 \cdot 5 \cdot 8418\,735626\,949973\,617503 \cdot$$

$$1232\,079689\,567662\,686148\,201863\,995544\,247703 \cdot$$

$$78\,507734\,924917\,342278\,622201\,969372\,653526\,213641\,483293.$$

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 by hitting on a 0.7% probability of finding it at this point. The smallest factor that we know we missed till now has 35 digits, although with 1536 trials with $B_1 = 700000$ and $B_2 = 15 \cdot 10^6$ we had a 26% probability of finding it. We refer to [4] for more information on the probability estimates.

In both our parallel elliptic curve implementations we used the following elementary method to lower the number of group operations needed to compute x^k . According to the definition of k given above, we have $k = \prod_{i=1}^l q_i$, where $\{q_1, q_2, \dots, q_l\}$ 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 $\sum_{i=1}^l [\log_2 q_i]$ squarings in G and $\sum_{i=1}^l (w(q_i) - 1)$ multiplications in G .

Let $S = S_1 \cup S_2 \cup \dots \cup S_s$ be a partition of $\{1, 2, \dots, l\}$, 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 $\sum_{j=1}^s w(\bar{q}_j)$ 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 $i \in \{1, 2, \dots, 20\}$ we determined partitions into triples of the prime powers $< 2 \cdot 10^6$ for the primes in the interval $[(i-1) \cdot 10^5, i \cdot 10^5]$. To give an example, the primes 1028107, 1030639, and 1097101 have weights 10, 16, and 11, but their product has weight 8.

Another approach of the partition problem might be to try to factor low-weight numbers, covering as many small primes as possible. The triples approach led to a saving of 18% in both elliptic curve programs. Notice that it can be used in any implementation of the elliptic curve method, and in any lengthy exponentiation where the exponent consists of many small factors. There exist various other exponentiation techniques that lower the number of squarings in G . Because squaring in G is a more complicated operation than multiplication, this might lead to even better results. We plan to investigate this in the near future. A complete description of the implementations discussed in this section can be found in [4; 5].

3. Gaussian elimination

Let M be a matrix over $\mathbb{Z}/2\mathbb{Z}$ with slightly more rows than columns. In this section we discuss the problem of finding linear dependencies modulo 2 among the rows of M . This problem occurs, among others, in the second stage of several general purpose factoring algorithms. There it is usually the case that M is fairly large, but sparse. For example, for a 116 digit number we factored, M had 120000 columns, and for the factorization of the ninth Fermat number [9] we had 200000 columns. In both cases the average number of non-zero entries per row was approximately fifty, with a rather large standard deviation. Furthermore the non-zero entries are *not* uniformly distributed per row, but instead the columns to the left are much denser than the columns to the right.

Let m denote the number of columns. It is well known that dependencies can be found in $O(m^3)$ bit operations by means of ordinary Gaussian elimination, where the pivot-search can be done from the sparse (right) side. In this way some profit can be gained from the sparseness, but not much. For our type of matrices fill-in usually occurs after approximately 1/3 of the pivots have been selected and processed. Thus, on ordinary workstations straightforward Gaussian elimination becomes problematic for large m . The largest m we have ever processed in this way was 80000, which took six weeks on an 8 MIPS workstation. We found that such long runs are in general unreliable: the frequency of uncorrected bit read or write errors is too high, and this can render the entire computation worthless.

Clearly, for $m \geq 10^5$ different methods or different machines have to be employed. There are various methods that can take advantage of the sparseness of M . Here we will discuss only one of these methods, *structured Gaussian elimination* [6; 18]. We have no experience yet with newer methods like Coppersmith's version of the Lanczos algorithm or his blocked Wiedemann algorithm [2; 3]. This last method in particular looks very promising, and we intend to investigate the possibilities of a MasPar implementation.

Structured Gaussian elimination is a preprocessing step during which the problem is reduced to a much smaller dense matrix by means of a sequence of clever pivot choices. The columns are partitioned into *heavy* and *sparse* columns, where initially all columns are considered sparse. Roughly speaking, each pivot is chosen in a sparse column of a row that causes only fill-in in the heavy columns of the matrix, after which the pivot row and column are removed from the matrix. When such a pivot choice is impossible, either some of the columns are moved from the sparse to the heavy part, or some excess rows are removed, if there are any, until a suitable pivot can be found. This is repeated until no sparse columns are left. For reasons that are not yet understood it seems to be beneficial to have many excess rows initially.

During this process, the heavy part of the matrix is not updated, but only the history of the eliminations that have been carried out is remembered. This information can then be used to build the smaller but much denser matrix corresponding to the heavy columns, and to convert dependencies among its rows into dependencies among the rows of the original matrix. Using this approach the 200000 column matrix (with 230000 rows) referred to above could be reduced to a dense matrix of 'only' 72000 columns (and slightly more rows). The 120000 matrix (with 142000 rows) could be reduced to a dense matrix of 45000 columns. In general we are able to reduce matrices that are obtained from factorizations to approximately 36% of their original number of columns. Structured Gaussian elimination, including the construction of the dense matrix, can easily be done on a workstation. For the above sizes this takes only a few hours. We are looking into a MasPar implementation, however, in particular for the construction of the dense matrix, to avoid heavy disk usage.

It remains to find linear dependencies among the rows of the 'small' dense matrix, in a way that is more reliable and faster than simply running ordinary Gaussian elimination on a workstation. For the 72000 matrix this was done on a 64K Connection Machine (another type of SIMD machine) using a program written by Roger Frye and Mike McKenna at Thinking Machines. The entire computation took three hours. For the 45000 matrix we used our own MasPar Gaussian elimination program, which took half an hour. Based on various experiments we estimate that this MasPar implementation can handle a dense 90000 matrix in less than three hours. Given how fast these fairly large dense matrices can be handled on SIMD machines, the problem of dealing with these matrices seems to be solved, at least for the near future.

We conclude this section with a short description of our MasPar Gaussian elimination program. Let D be the (dense) 0-1-matrix to be processed, and assume that D fits in core, i.e., has fewer than approximately 92000 columns. This is not a major restriction, but it simplifies the description. There are various ways to map D into the PE array, each with its own (dis)advantages compared to the others. Our first attempt, which certainly looks quite natural, worked entirely satisfactorily, so that we never bothered to consider the alternatives; the results will probably be similar.

Roughly speaking, we superimposed the PE array on the dense representation of D . More precisely, let D have r non-zero rows and c columns, where we assume that both r and c are divisible by 128 which can be achieved by adding zero rows and/or columns to D if necessary. Distribute D over the PE array such that the i th row of PE's stores the i th block of $r/128$ consecutive rows of D , and similarly such that the j th column of PE's stores the j th block of $c/128$ consecutive columns, for $i, j = 1, 2, \dots, 128$. So, each row of D is divided into $c/128$ chunks over a particular row of PE's. In our MPL implementation we made c divisible by 4096, so that the matrix elements per PE could be stored in an $(r/128) \times (c/4096)$ array of 32 bit words.

Using the processor interconnection features available on the MasPar, implementation of Gaussian elimination with this matrix layout is straightforward. For $i = 1, 2, \dots, r$ in succession, do the following. First select a new pivot column in the i th row of D , where only the corresponding row of PE's is active, and send the pivot information (the entire pivot row and the column number of the pivot element) to all PE rows straight to the south. So, each PE to the south now contains the chunk of the pivot row that corresponds to its chunks of the rows of D . Next, for all PE's corresponding to the row chunks that contain the pivot column, check which later rows have a one in the pivot column, and

send this information to all other PE columns, both west and east. Each PE now knows to which of its row chunks of D it should add (modulo 2) the corresponding pivot row chunk. Finally, these additions are carried out simultaneously on all PE's, and the next pivot can be selected. If no new pivot can be found a linear dependency has been found. The process can easily be organized in such a way that the coefficients of the dependency can be read off from the entries in the 'zero' row; for this it suffices to set the pivot element in the pivot row to zero, before doing the additions.

Notice that in this description the PE's are on average idle half of the time; the first row of PE's, for instance, is inactive after the first $r/128$ pivots have been processed. By selecting the i th pivot row in the $((i-1) \bmod 128 + 1)$ th row of PE's, and by sending the pivot information both north and south, this can be avoided.

4. Quadratic sieve algorithm

In this section we describe some of the experiments that we have carried out on the 16K MasPar to implement the first stage of the multiple polynomial quadratic sieve factoring algorithm (mpqs). For a description of mpqs we refer to [11, 19]. Roughly speaking, during the first stage of mpqs a sequence of quadratic polynomials depending on n is processed, independent of each other and in any order, to collect a certain type of data, until sufficiently many data have been obtained. This sequence of polynomials is computed using a sequence of primes close to $\sqrt[3]{n}$. For each polynomial f the following has to be done.

- (i) For all $p \in P$ compute the roots r_{1p} and r_{2p} of f modulo p , where P is some fixed collection of primes, the factor base;
- (ii) For all $i \in I$ set s_i to zero, where I is some interval of integers;
- (iii) For all $p \in P$ and $j = 1, 2$ replace s_i by $s_i + [\log p]$ for all $i \in I$ which are equal to r_{jp} modulo p ;
- (iv) For all $i \in I$ for which s_i is sufficiently large, try to factor $f(i)$ using the elements of P . The $f(i)$ that can (almost) completely be factored are the data being sought.

The s_i are usually represented by a single byte (8 bits). For the 116 digit n mentioned in the introduction, we had $\#P = 120000$, and $\#I = 33500000$; reasonable choices for a 95 digit n are for instance $\#P = 30000$ and $\#I = 4000000$. In many implementations the interval I is broken into smaller subintervals that fit in memory, and steps (ii), (iii), and (iv) are carried out for each of the subintervals.

Although it is in principle possible to process 16K polynomials in parallel on the MasPar, the PE memory is far too small to make this approach efficient: the interval I would have to be split up into many subintervals, and for each of them the roots have to be recomputed because there is not enough memory to store them. For a 95 digit n it is possible, however, to process 128 polynomials in parallel in a fairly efficient manner. Actually, also for any $d < 128$ dividing 128 we can process d polynomials in parallel, which might be better for larger n .

To process 128 polynomials in parallel, we proceed as follows. First, each PE attempts to find a prime close to $\sqrt[3]{n}$ satisfying certain conditions, in such a way that no duplicates are generated. This is continued until each PE row contains at least one prime, which can be efficiently done using random north/south redistributions. For each PE row, one prime is selected, and distributed among all PE's in that row. Each PE then derives the

polynomial corresponding to that prime, so that there are 128 different polynomials, with PE's in the same row having the same polynomial.

To each of the 128 PE's in a PE row we assign $\#P/128$ of the elements of the factor base, and a subinterval of I of length $\#I/128$ (initialized as zero), such that consecutive PE's get consecutive subintervals (where we assume that both $\#P$ and $\#I$ are divisible by 128, and that the PE memory is large enough). For each of the primes p in the PE's subset of the factor base, the following is done simultaneously by all PE's. First they compute the smallest roots u_1 and u_2 in I of their polynomial modulo p , and the smallest corresponding roots t_1 and t_2 in their own or a later subinterval of I . Next, for $t = t_1, t_2$, as long as t belongs to their own subinterval, s_t is replaced by $s_t + [\log p]$ and t by $t + p$. Finally, all relevant information concerning p is sent to the first PE to the right (with wraparound), and is used similarly to update the sieve intervals there, for that p ; this last step is repeated 127 times, after which step (iii) has been carried out for 128 primes in the factor base. Notice that the left most PE in each PE row does not use the t_i , but replaces them by the u_i .

To make step (iv) efficient in SIMD mode, each PE builds its own stack of candidate $f(i)$'s to factor, the elements of which are randomly distributed to other PE's, to keep the memory requirements per PE low, and to minimize the waiting time until all PE's have at least one $f(i)$ to trial divide. The actual trial division can then easily and efficiently be carried out on 16K different $f(i)$'s.

Initial experiments with this approach were encouraging. Although some time is needed for the communication in the pipelined sieve, this did not cause major inefficiencies. We expect that we will be able to factor numbers in the ninety digit range in one night; one hundred digits should take a few days.

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. D. Coppersmith: Solving linear equations over $GF(2)$. IBM Research Report RC 16997, 1991
3. D. Coppersmith: Solving linear equations over $GF(2)$ II: Block Wiedemann algorithm. Manuscript, 1991
4. B. Dixon, A. K. Lenstra: Massively parallel elliptic curve factoring. In preparation
5. B. Dixon, A. K. Lenstra: Systolic Montgomery multiplication. In preparation
6. B. A. LaMacchia, A. M. Odlyzko: Solving large sparse systems over finite fields. Proc. Crypto '90, to appear
7. A. K. Lenstra, H. W. Lenstra, Jr.: Algorithms in number theory. In: J. van Leeuwen (ed.) Handbook of theoretical computer science, Volume A, Algorithms and complexity. Elsevier, Amsterdam, 1990, Chapter 12
8. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard: The number field sieve. In preparation. Extended abstract: Proc. 22nd Annual ACM Symp. on Theory of Computing (STOC), Baltimore, May 14-16 (1990) 564-572
9. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard: The factorization of the ninth Fermat number. Submitted for publication
10. A. K. Lenstra, M. S. Manasse: Factoring by electronic mail. Advances in Cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci. 434 (1990) 355-371

11. A. K. Lenstra, M. S. Manasse: Factoring with two large primes. In preparation. Extended abstract: *Advances in Cryptology, Eurocrypt '90, Lecture Notes in Comput. Sci.* **473** (1991) 72-82
12. H. W. Lenstra, Jr.: Factoring integers with elliptic curves. *Ann. of Math.* **126** (1987) 649-673
13. MasPar MP-1 principles of operation. MasPar Computer Corporation, Sunnyvale, CA, 1989
14. U.M. Maurer, Y. Yacobi: Non-interactive public key cryptography. *Advances in Cryptology, Eurocrypt '91, Lecture Notes in Comput. Sci.*, to appear
15. P. L. Montgomery: Modular multiplication without trial division. *Math. Comp.* **44** (1985) 519-521
16. P. L. Montgomery: Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* **48** (1987) 243-264
17. Y. Murakami, M. Kasahara: The discrete logarithm problem under a composite modulus. *ISEC 90*
18. C. Pomerance, J. W. Smith: Reduction of large, sparse matrices over a finite field via created catastrophes. In preparation
19. R. D. Silverman: The multiple polynomial quadratic sieve. *Math. Comp.* **48** (1987) 329-339
20. R. D. Silverman, S. S. Wagstaff, Jr.: A practical analysis of the elliptic curve factoring algorithm. Manuscript