Factoring Integers Using SIMD Sieves

Brandon Dixon[1] and Arjen K. Lenstra[2]

[1] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA,
E-mail: bdd@cs.princeton.edu
[2] Room MRE-2Q334, Bellcore, 445 South Street, Morristown, NJ 07960, USA,
E-mail: lenstra@bellcore.com

Abstract. We describe our single-instruction multiple data (SIMD) implementation of the multiple polynomial quadratic sieve integer factoring algorithm. On a 16K MasPar massively parallel computer, our implementation can factor 100 digit integers in a few days. Its most notable success was the factorization of the 110-digit RSA-challenge number, which took about a month.

1 Introduction

Usually one distinguishes two types of integer 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. To evaluate the security of factoring-based cryptosystems, it is important to study the practical behavior of general purpose factoring algorithms. In this paper we present an efficient SIMD-implementation of the multiple polynomial quadratic sieve (QS) factoring algorithm [12], still the most practical general purpose method for integers in the range from 80 to 120 digits.

The largest number factored by QS is a 116-digit number. This factorization was carried out in a few months on a widely distributed network of workstations, and took a total computation time of approximately 400 mips-years (1 mips-year is about $3.15 \cdot 10^{13}$ instructions) [10]. The previous QS record for a single-machine implementation had 101 digits, and was carried out on one processor of a four processor Cray Y-MP4/464 in 475 CPU-hours [14]. This record was broken by our factorization of the 110-digit RSA-challenge number.

As will be explained in Section 3, QS consists of two main steps: the sieving step, and the matrix elimination step. All successful parallel implementations of QS that we know of have followed the approach described in [2]: distribute the sieving step over any number of available processors, which work independently of each other, and collect their results and perform the matrix elimination at a central location. In [2] the sieving step was done on a local network of workstations using the Ethernet for the communication, in [9] the workstations are scattered around the world, and communicate with the central location using electronic mail. In both of these implementations the network of participating machines can be viewed as a loosely coupled multi-processor machine, where the

T. Helleseth (Ed.): Advances in Cryptology - EUROCRYPT '93, LNCS 765, pp. 28-39, 1994.

processors work asynchronously in *multiple-instruction multiple data* (MIMD) mode; i.e., each processor carries out its own set of instructions on its own set of data. Very powerful and fairly expensive massively parallel MIMD machines, with more computational power than was ever achieved using the approach from [9], are currently available. It should not be hard to break the 116-digit QS record on such a machine, but getting enough computing time might be prohibitively expensive, and is certainly more expensive than it was to use the donated cycles from the internet network.

Another type of massively parallel machine is the *single-instruction multiple data* (SIMD) machine. These machines usually consist of some particular network (hyper-cube, mesh) of several thousand small processors, each with its own fairly limited amount of memory. Unlike the processors on MIMD machines which can work more or less on their own, the SIMD processors simultaneously carry out the same instructions, but each processor on its own data. Furthermore, arbitrary subsets of processors can be made inactive or reactivated at any time. Although the rough computational power of large SIMD machines is comparable to that of supercomputers or large MIMD machines, SIMD machines are only efficient for computations that can be cast as a SIMD process.

SIMD machines have proven to be very useful for the matrix elimination step of QS [6; 8]. For that application, as well as for various other factoring applications (cf. [6]), it suffices to have a fairly restrictive type of SIMD machines, namely machines that only allow direct addressing (where the memory address is part of the instruction). For efficient sieving applications, however, it is essential that indirect addressing is available as well, i.e., the memory address depends on a value local to the processor. Although this is not the case for all SIMD machines, we will assume throughout this paper that this requirement is met (as it is for the 16K MasPar SIMD machine[3] that we used for our work, cf. Section 2).

Nevertheless, at first sight the sieving step does not look like the type of operation that would run well on any SIMD machine (cf. Section 3). Indeed, a MasPar-implementation of the QS-sieving step which was attempted in [5] seems to support this supposition. In this paper we describe a different approach to the SIMD-implementation of the sieving step which works quite efficiently: on a 16K MasPar 100-digit integers can be factored within three CPU-days. A 110-digit number took one CPU-month, where we used only 5/8 of the total available memory; using all the memory this factorization would have taken about 20

[3] 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.

days. With a later, faster version of the program we were able to do a 105-digit number in 5.3 CPU-days, using half of the available memory. This shows that relatively inexpensive SIMD machines are much better for general purpose factoring than was previously expected. For SIMD-implementations of special purpose factoring algorithms (like the elliptic curve method) we refer to [4].

The success of this implementation prompted work on a SIMD-implementation of the general number field sieve factoring method [1]. With this number field sieve implementation we broke the record set by the factorization of the ninth Fermat number, by factoring the 151-digit number $(2^{503}+1)/3$ and the 158-digit number $2^{523}-1$.

The remainder of this paper is organized as follows. In Section 2 a short description of the SIMD machine that we used is given. Section 3 contains a general description of QS. A simple algorithm for the redistribution of data on a SIMD machine is given in Section 4. An overview of our SIMD QS implementation is presented in Section 5, and Section 6 contains an example.

2 The Hardware

This section contains a short overview of the 16K MasPar, the massively parallel computer that we have used for the implementation to be described in this paper. Our description is 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, such as [11].

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. PE's cannot address each other's memory, but as mentioned in the introduction PE's can do indirect addressing. Each processor can communicate efficiently 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 less efficient global router that allows any processor to communicate with any other processor, but we never needed it.

Each job has a size between 4K 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 The Quadratic Sieve Factoring Method

Let $n > 1$ be an odd positive integer that is not a prime power. For each random integer x satisfying

$$x^2 \equiv 1 \bmod n \tag{3.1}$$

there is a probability of at least $1/2$ that $\gcd(n, x-1)$ is a non-trivial factor of n. To factor n it therefore suffices to construct several such x's in a more-or-less random manner.

In many factoring algorithms, solutions to (3.1) are sought by collecting integers v such that

$$v^2 \equiv \prod_{p \in P} p^{e_p(v)} \bmod n, \tag{3.2}$$

where the *factor base* P is some finite set of integers that are coprime to n, and $e_p(v) \in \mathbf{Z}$ for $p \in P$. A pair $(v, e(v))$ satisfying (3.2), with $e(v) = (e_p(v))_{p \in P} \in \mathbf{Z}^{\#P}$, is called a *relation*, and will be denoted by v for short. If V is a set of relations with $\#V > \#P$, then there exist at least $2^{\#V - \#P}$ distinct subsets W of V with $\sum_{v \in W} e(v) = (2w_p)_{p \in P}$ and $w_p \in \mathbf{Z}$; these subsets can be found using Gaussian elimination modulo 2 on the set of vectors $e(v)$ mod 2. Each such W leads to an $x \equiv (\prod_{v \in W} v) \cdot (\prod_{p \in P} p^{-w_p}) \bmod n$ satisfying (3.1).

In the original quadratic sieve factoring algorithm [12] relations are collected as follows. Let P consist of -1 and the primes $\leq B$ with Legendre symbol $\left(\frac{n}{p}\right) = 1$, for some bound B. An integer is called B-smooth if it can be written as a product over P. Relations are collected by looking for small integers i such that $f(i) = (i + [\sqrt{n}])^2 - n$ is B-smooth; for such i we have that $v = i + [\sqrt{n}]$ satisfies (3.2). Because a prime p divides $f(i)$ if and only if it divides $f(i+kp)$ for any integer k, smooth values can be found efficiently using a sieve, if the roots of f modulo the primes in P are known. For this reason, the relation collecting step is called the *sieving step*. Notice that only primes p with $\left(\frac{n}{p}\right) = 1$ can divide $f(i)$; this explains the definition of P.

The second step, finding subsets W as above, is called the *matrix elimination step*. In this paper we will not pay any further attention to this step, as it is well known how it can be dealt with for $\#P$ up to, say, 200000 (cf. [6; 8]).

Let

$$L[\gamma] = \exp((\gamma + o(1))\sqrt{\log n \log\log n}), \tag{3.3}$$

for a real number γ, and $n \to \infty$. With $B = L[1/2]$ it can be shown on loose heuristic grounds that both steps of the quadratic sieve algorithm can be completed in expected time $L[1]$.

Because in the original quadratic sieve only one polynomial f is used to generate all $> \#P$ relations, the interval of i-values to be inspected is rather large. Since $f(i) \approx 2i[\sqrt{n}]$ grows linearly with i, the probability of $f(i)$ being B-smooth decreases with increasing i. Davis [3] suggested using more than one polynomial, thus allowing a smaller i-interval per polynomial which should increase the yield (per i) of the sieving step. In our implementation we used Montgomery's version of this same idea [16]. Let P be as above, and let

$$f(i) = a^2i^2 + bi + (b^2 - n)/(4a^2), \tag{3.4}$$

for integers a, b with $b^2 \equiv n \bmod 4a^2$. This requires n to be $1 \bmod 4$, which can be achieved by replacing n by $3n$ if necessary; in practice it might even be advantageous to use some other multiplier, cf. [16]. If $f(i)$ is B-smooth, then $v = (ai + b/(2a)) \bmod n$ satisfies (3.2), and B-smooth $f(i)$'s can again be found using a sieve once the roots of f modulo the primes in P are known. Thus, polynomials satisfying (3.4) can be used to generate relations efficiently. The expected run-time of the resulting factoring algorithm, however, is still $L[1]$.

3.5 Constructing Polynomials

(cf. [7; 16]). We show how polynomials as in (3.4) can be constructed. Let M be such that $f(i)$'s with $i \in [-M, M)$ will be tested for smoothness in the sieve. Let $a \equiv 3 \bmod 4$ be a probable prime with $a^2 \approx \sqrt{n/2}/M$ and Jacobi symbol $\left(\frac{a}{n}\right) = 1$. Since a is free of primes in P, the polynomial f has two roots modulo all primes in the factor base P. To find b such that $b^2 \equiv n \bmod 4a^2$, we first set $b = n^{(a+1)/4} \bmod a$ so that $b^2 \equiv n \bmod a$. Next we replace b by

$$b + a\big((2b)^{-1}((n - b^2)/a) \bmod a\big),$$

and finally, if b turns out to be even we replace b by $b - a^2$. It follows that $b^2 \equiv n \bmod 4a^2$, and $|f(i)| = O(i\sqrt{n})$ for $i \in [-M, M)$. Notice that the roots of $f \bmod p$ are $(-b \pm \sqrt{n})/(2a^2) \bmod p$, so that computation of the roots requires one inversion modulo p for all primes p in P; the values of $r_p \equiv (\sqrt{n}) \bmod p$ should be precomputed and stored in a table.

Another method to generate polynomials is presented in [13], and has the advantage that the roots of a polynomial modulo the primes in the factor base can be derived easily from the roots of the previous polynomial. We have no practical experience with this method.

In implementations of QS one usually also collects values of i for which $f(i)$ factors almost completely using the elements of P, i.e., except for one (or two) larger primes. If the large primes in these so-called *partial relations* match, they can be combined to form relations as in (3.2). In practice this enhancement leads to a speed-up factor of 4 to 6. For a detailed description of how the number of

useful combinations among the partial relations can be counted, and how the combinations can be actually formed, we refer to [10].

3.6 The Sieving Step

Summarizing, for some fixed choice of factor base P, sieving bound M and $n \boxminus 1 \bmod 4$, the QS-sieving step can be carried out by performing steps (a) through (h).

(a) For all primes p in P compute $r_p \equiv (\sqrt{n}) \bmod p$;

(b) Set $a_{\text{low}} = [\sqrt[4]{n/2}/\sqrt{M}]$;

(c) Compute the smallest $a > a_{\text{low}}$ that satisfies the requirements in (3.5), compute the corresponding b as in (3.5), and let f be as in (3.4);

(d) For all primes p in P compute the roots r_{p1} and r_{p2} of $f \bmod p$ as $(-b \pm r_p)/(2a^2) \bmod p$;

(e) For all integers i with $i \in [-M, M)$ set $s(i)$ to zero;

(f) For all primes p in P and $v = 1, 2$ replace $s(i)$ by $s(i) + [\log p]$ for all $i \in [-M, M)$ which are equal to r_{pv} modulo p (this is the actual sieving step);

(g) For all $i \in [-M, M)$ for which $s(i)$ is sufficiently close to the report bound $\log |f(i)|$, try to factor $f(i)$ using the elements of P, and store the resulting relations and partial relations (an i for which $s(i)$ is close to the report bound is called a *report*);

(h) If more relations and partial relations are needed, replace a_{low} by a and go back to step (c); otherwise terminate.

3.7 Practical Remarks

In step (f) one often does not sieve with the small primes in P, or replaces them by sufficiently large powers, to increase the speed. This lowers the number of reports in step (g), so that the report bound has to be lowered accordingly. For a 100-digit n, the factor base P will have approximately 50000 elements, and the largest element of P will be about $1.3 \cdot 10^6$. Because a small multiple of this largest prime is a good choice for M, several million $s(i)$'s have to be stored. Although each $s(i)$ is usually represented by a single byte (8 bits), the $s(i)$'s together might not fit in memory. In that case, the interval $[-M, M)$ is broken into smaller subintervals, which are processed consecutively as above. In practice the report bound $\log |f(i)|$ is often replaced by some appropriately chosen fixed bound.

3.8 Parallelization

The sieving step can easily be parallelized on any number of independent processors, by restricting each of them to a unique interval of candidate a-values, disjoint from the intervals assigned to other processors. Notice that two different identical processors that run the same sieving program and that started at the same time, each on its own interval of candidate a-values, are most likely to be at entirely different points in the program, even after a very short run: one processor might find a 'good' a-value earlier than the other in step (c), and thus begin earlier with the next steps, or one processor might find more reports in step (g) and spend more time on the trial divisions of the corresponding $f(i)$'s. Also at other points the precise instruction stream that gets executed might differ (for instance, in the inversions modulo p in step (d)), but these differences are minor compared to the entirely different things that might happen in steps (c) and (g). In a situation where several copies of (3.6) are processed simultaneously in SIMD-mode, this might lead to major inefficiencies, because the process that happens to be the slowest for a particular step sets the pace for that step. In the next section it is shown how these inefficiencies can be avoided at the expense of some memory.

4 Redistributing Data

In our SIMD-implementations of (3.6)(c) and (3.6)(g) we find ourselves in the following situation. We have a toroidal mesh M of m SIMD processing elements (PE's) that allows fast communication between each PE and its eight nearest neighbors. For the 16K MasPar described in Section 2, for instance, M would be the array of PE's, and m would be 16K. Furthermore, there is an inexpensive SIMD-process G, such that each PE running G has a fairly small probability p, independent from the other PE's, to generate a useful packet of information. These packets have to be processed by a time-consuming SIMD-process B. The goal is to process as many packets as possible, by repeating G and B indefinitely.

Clearly, since p is small it is quite inefficient to perform B right after G because then only the few PE's that have found a packet would be processing B. Fortunately, in our situation we can take advantage of the following.

(i) G can be repeated an arbitrary number of times before B is executed (i.e., packets do not have to be used immediately after they have been generated);

(ii) It is irrelevant for B on which PE a particular packet was found (i.e., packets may be generated and processed on different PE's);

(iii) It is not crucial that *all* packets that have been generated are also actually processed by B, but of course generating packets that will not be used leads to inefficiencies.

Using (i) we could keep a stack of packets per PE, and apply G until each stack contains at least one packet. At that point all m top of stack elements could be popped and processed by B, after which G is again applied, and so on. For

small p and large m this approach would require rather large stacks on the PE's unless many packets are discarded, using (iii).

A better solution that uses much smaller stacks and that avoids discarding too many packets redistributes the packets after every application of G, thus making use of (ii) as well. There are many ways to do this; for us the following worked entirely satisfactorily.

4.1 Random Redistribution

On all m PE's simultaneously, do the following in SIMD-mode. Fix some arbitrary ordering N_1, N_2, ..., N_8 of the eight nearest neighbors (the same ordering for all PE's). For $i = 1, 2, \ldots, 8$, set $N = N_i$ and perform steps (a) and (b).

(a) Get the number of packets n on N's stack;
(b) If $n + 1$ is smaller than the number of packets on the PE's own stack, then perform steps (b1) through (b3);
(b1) Set e equal to the top of stack packet, and pop this packet from the stack;
(b2) Push e on the top of the stack of N;
(b3) Go back to step (a).

This approach resulted in the following behavior. Starting from empty stacks on all PE's it took on average $2/p$ applications of G (each followed by (4.1)) until none of the PE's had an empty stack. From that point on it takes, after each execution of B on all m PE's, on average $1/p$ applications of G (plus (4.1)), with a very small variance, before B can be applied again to process m packets. Except for the start-up stage, this is the best one could hope for.

Although we tried several orderings of the neighbors, we never noticed a significant difference in the performance. With stacks of at most 5 packets we occasionally lost packets but this introduced only a minor inefficiency. A simpler variant of (4.1) would be to remove the jump back to step (a) in the case that at least one packet has been moved to a neighbor. Similarly, G can be repeated a few times, before (4.1) is applied (with the jump). We have no experience with these simplifications, but we suspect that they work equally well. Notice that (4.1) uses only communication with nearest neighbors, with toroidal wraparound, which keeps communication costs to a minimum.

5 A SIMD Implementation of the QS-Sieving Step

Given the redistribution algorithm from the previous section, there are various ways to implement the QS-sieving step efficiently on a SIMD-machine, as long as the machine provides reasonably fast communication between neighbors. The simplest approach would be to let each PE generate polynomials as in (3.6)(c) (making sure that they try different a-values), until each PE got at least one polynomial (using a stack of polynomials and (4.1)), after which each PE performs steps (3.6)(d)-(g) using the polynomial it ended up with (on the top of its stack of polynomials). This works efficiently in SIMD mode and without further inter-processor communication, except for the trial divisions in (3.6)(g); the

$f(i)$'s, however, can again be redistributed using (4.1), so that trial division too can be performed on all PE's simultaneously.

Although these applications of (4.1) solve the synchronization problems caused by SIMD execution of (3.6)(c) and (3.6)(g), this approach is inefficient on the 16K MasPar, because there is not enough memory per PE to store a sufficiently large chunk of the interval $[-M, M)$. Furthermore, the roots in (3.6)(d) would have to be recomputed for each subinterval of $[-M, M)$ to be processed, because there is not enough memory on a PE to store them.

The opposite approach would be to process one polynomial at a time, and to spread the interval $[-M, M)$ over the PE's. This is a feasible approach if there if an ordering PE_0, PE_1, ..., PE_{m-1} such that PE_i and PE_{i+1} can communicate quickly (with indices modulo m). On the 16K MasPar this would not be impossible, but it would lead to a fairly small subinterval per PE with a very small hit-probability during the sieving step, unless the combined interval $[-M, M)$ is chosen exceedingly long.

One row of 128 PE's on a 16K MasPar has a total amount of memory of $128 \times 64\text{K} = 8\text{MBytes}$, which is just about the right amount of memory to store the sieving interval $[-M, M)$ for the factorization of a 100-digit n. This observation suggests that on the 16K MasPar it might be a good idea to process 128 polynomials at a time, with each of the 128 rows of 128 processors taking care of one polynomial. This is the approach that we opted for. Since there will be no communication between processors in different rows, except for the redistributions (cf. (4.1)) during the polynomial generation and trial division steps, we restrict our description to what happens in a single row of 128 processors.

Let PE_0, PE_1, ..., PE_{127} be a row of 128 processors, such that PE_j and PE_{j+1} can communicate quickly (with indices modulo 128, i.e., with toroidal wraparound). We remove -1 and the small primes from the factor base P, choose the remaining P such that $\#P = 128 \cdot k$, for some integer k, and we partition P over the row of processors in such a way that each processor stores k primes (but see Remark (5.2)(f)). Furthermore, each processor contains k square roots of n modulo its k primes (the r_p from (3.6)(a)). Finally, we choose M such that $M = 64 \cdot L$ for some integer L, and we divide the sieving interval $[-M, M)$ over the processors in such a way that PE_j stores the length L interval $I_j = [(j - 64)L, (j - 63)L)$.

Suppose that we have repeatedly applied (3.6)(c) combined with (4.1) on the entire 128×128 processor array simultaneously until each of the 16K processors has a non-empty stack of polynomials. In particular, we suppose that each PE_j contains a unique polynomial f_j, for $0 \leq j \leq 127$. These polynomials are processed one at a time in 128 iterations. To process f_j processor PE_j first broadcasts f_j to the other processors in the row, so that all 128 processors share the same polynomial, after which the pipelined sieving from (5.1) is performed.

5.1 Pipelined Sieving

Suppose that PE_0 through PE_{127} all have the same polynomial f, represented by a and b as in (3.4). The values for m and l below are the same on all processors and can thus be taken care of by the ACU (cf. Section 2). Perform steps (a) through (c) on PE_0 through PE_{127} simultaneously.

(a) For all $i \in I_j$ set $s(i)$ to zero (cf. (3.6)(e));

(b) For $m = 1, 2, \ldots, k$ in succession (where $\#P = 128 \cdot k$), perform steps (b1) through (b5);

(b1) Let p be the mth prime on PE_j and r_p the corresponding squareroot of n (with different p's for different j's);

(b2) Compute r_{p1} and r_{p2} as the smallest integers $\geq (j - 64)L$ which are equal to $(-b + r_p)/(2a^2)$ and $(-b - r_p)/(2a^2)$ modulo p, respectively (cf. (3.6)(d)), and compute $\tilde{r}_{p1}$ and $\tilde{r}_{p2}$ as the smallest integers $\geq -64L$ which are equal to r_{p1} and r_{p2} modulo p, respectively;

(b3) Set $l = 0$;

(b4) For $v = 1, 2$, as long as $r_{pv} < (j - 63)L$, replace $s(r_{pv})$ by $s(r_{pv}) + [\log p]$ and next r_{pv} by $r_{pv} + p$ (cf. (3.6)(f));

(b5) Replace l by $l + 1$. If $l < 128$, replace the 5-tuple $(p, r_{p1}, r_{p2}, \tilde{r}_{p1}, \tilde{r}_{p2})$ by the corresponding 5-tuple from the left neighbor (with wraparound), on PE_0 only replace r_{p1} and r_{p2} by $\tilde{r}_{p1}$ and $\tilde{r}_{p2}$, respectively, and return to step (b4);

(c) For all $i \in I_j$ for which $s(i)$ is sufficiently close to the report bound $\log |f(i)|$, push i, a, and b on the top of a stack of reports (cf (3.6)(g)).

This finishes the description of (5.1). Notice that in (5.1) 128 polynomials are processed simultaneously on 128 rows of processors. After every execution of (5.1) the elements of the stack built in step (c) will be redistributed using (4.1). As soon as all 16K processors have at least one report, the $f(i)$'s are computed and the actual trial divisions (using the original factor base including the small primes) are carried out on 16K processors simultaneously. And after 128 applications of (5.1) new polynomials are generated using (3.6)(c) and (4.1).

5.2 Practical Remarks

(a) In our implementation we removed the first 50 primes from P, to speed up the sieving step; the threshold is best determined experimentally.

(b) We gained considerable additional efficiency by distributing P over the PE_i in such a way that its smallest 128 elements are taken care of by $m = 1$ in step (b), the next smallest 128 by $m = 2$, etc.: as soon as for some m all p's are $> L$, at most one sieve location per PE has to be updated in step (b4), which means that for larger m simpler code can be used. An additional advantage of this approach is that $[\log p]$ in (5.1)(b4) can be replaced by a sufficiently close approximation which is the same over the entire array of processing elements, and which can be changed depending on the value of m. As usual, instead of $\log |f(i)|$ we used some fixed value for the report bound in (5.1)(c).

(c) In our implementation we only kept track of r_{p1}, $\tilde{r}_{p1}$ and $r_{p1} - r_{p2}$. Although this leads to slightly more complicated computations, it also resulted in fewer communications (a 4-tuple instead of a 5-tuple in (5.1)(b5)) and thus greater speed.

(d) If the interval $[-M, M)$ is too large, (5.1) can be applied repeatedly to subintervals of $[-M, M)$ simply by changing the definition of I_j appropriately (but see Remark (f) below). In this way we could decrease the memory requirements of our implementation considerably, at the expense of an acceptable slow-down. So, instead of the full 64K per PE, we normally use only about 28K, thus allowing other people to share the machine with us.

(e) In the trial division, the PE's simultaneously process the primes in the complete factor base to build a list of primes occurring in their $f(i)$. After that, each PE works on this list of primes to determine the multiplicities and the remaining factor. All PE's for which the remaining factor is 1, or a sufficiently small prime $> B$, or a sufficiently small composite, dump their trial division results to disk. The remaining composites (which lead to partial relations with two large primes) are factored by a separate program that runs independently on the front end.

(f) In the above description each processor stores k elements of P and the corresponding roots, where $\#P = 128 \cdot k$, and processors in the same column store the same subsets. We found it more efficient to distribute these subsets over the columns, so that processors store at most $[(k-1)/128]+1$ elements of P plus the corresponding roots. This leads to some extra communication (in Step (5.1)(b1)), but it makes more memory available for the sieve. If the sieve is broken into smaller subintervals, as suggested in Remark (d), then we either need $O(k)$ memory per processor to remember various useful values for the next subinterval, or these values have to be recomputed for each subinterval. This leads to loss of memory (which could have been used for the sieve) or loss of time. In particular for large $\#P$ we found it more efficient *not* to use Remark (d) at all, even though the sieve gets so small compared to $\max P$ that many primes do not even hit it once.

6 Example

The largest number we have factored using our SIMD QS implementation is the 110-digit number from the RSA-challenge list [15]:

$$\begin{aligned}\mathrm{RSA}(110) =& 35794234179\,72586877499\,18078325684\,55403003778\,02422822619\\ & 35329081904\,84670252364\,67741151351\,61112045040\,60317568667\\ =& 58464182144\,06154678836\,55318297916\,23841986105\,05601062333\cdot\\ & 61224210904\,93547576937\,03731756141\,88412257585\,54253106999\end{aligned}$$

At the time of writing, this is the largest number factored on a single machine using a general purpose factoring method. This factorization took approximately a month of computing time on the 16K MasPar, where we used only 40K of the

64K bytes per PE. The factor base consisted of approximately 80,000 primes, and the sieving interval $[-M, M)$ was broken into two consecutive pieces (cf. (5.2)(d)). We did not use the suggestions from Remark (5.2)(f) for this factorization; we later found out that they would have saved us quite some time. They were used, however, for the factorization of a 105-digit number in 5.3 CPU-days using 30K per PE. Extrapolation to a 110-digit number would give at most 20 CPU-days, still with 30K per PE.

References

1. Bernstein, D. J., Lenstra, A. K.: A general number field sieve implementation (to appear)
2. Caron, T. R., Silverman, R. D.: Parallel implementation of the quadratic sieve. J. Supercomputing **1** (1988) 273–290
3. Davis, J. A., Holdridge, D. B.: Factorization using the quadratic sieve algorithm. Tech. Report SAND 83-1346, Sandia National Laboratories, Albuquerque, NM, 1983
4. Dixon, B., Lenstra, A. K.: Massively parallel elliptic curve factoring. Advances in Cryptology, Eurocrypt'92, Lecture Notes in Comput. Sci. **658** (1993) 183–193
5. Gjerken, A.: Faktorisering og parallel prosessering (in norwegian), Bergen, 1992
6. Lenstra, A. K.: Massively parallel computing and factoring. Proceedings Latin'92, Lecture Notes in Comput. Sci. **583** (1992) 344–355
7. Lenstra, A. K., Lenstra, H. W., Jr.: Algorithms in number theory. Chapter 12 in: van Leeuwen, J. (ed.): Handbook of theoretical computer science. Volume A, Algorithms and complexity. Elsevier, Amsterdam, 1990
8. Lenstra, A. K., Lenstra, H. W., Jr., Manasse, M. S., Pollard, J. M.: The factorization of the ninth Fermat number. Math. Comp. **61** (1993) (to appear)
9. Lenstra, A. K., Manasse, M. S.: Factoring by electronic mail. Advances in Cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci. **434** (1990) 355–371
10. Lenstra, A. K., Manasse, M. S.: Factoring with two large primes. Math. Comp. (to appear)
11. MasPar MP-1 principles of operation. MasPar Computer Corporation, Sunnyvale, CA, 1989
12. Pomerance, C.: Analysis and comparison of some integer factoring algorithms. 89–139 in: Lenstra, H. W., Jr., Tijdeman, R. (eds): Computational methods in number theory. Math. Centre Tracts **154/155**, Mathematisch Centrum, Amsterdam, 1983
13. Pomerance, C., Smith, J. W., Tuler, R.: A pipeline architecture for factoring large integers with the quadratic sieve algorithm. SIAM J. Comput. **17** (1988) 387–403
14. te Riele, H., Lioen, W., Winter, D.: Factorization beyond the googol with mpqs on a single computer. CWI Quarterly **4** (1991) 69–72
15. RSA Data Security Corporation Inc., sci.crypt, May 18, 1991; information available by sending electronic mail to challenge-rsa-list@rsa.com
16. Silverman, R. D.: The multiple polynomial quadratic sieve. Math. Comp. **84** (1987) 327–339