Factoring by electronic mail

Arjen K. Lenstra
Department of Computer Science
The University of Chicago
1100 E 58th Street
Chicago, IL 60637
arjen@gargoyle.uchicago.edu

Mark S. Manasse
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
msm@src.dec.com

Abstract. In this paper we describe our distributed implementation of two factoring algorithms, the elliptic curve method (ecm) and the multiple polynomial quadratic sieve algorithm (mpqs).

Since the summer of 1987, our ecm-implementation on a network of MicroVAX processors at DEC's Systems Research Center has factored several most and more wanted numbers from the Cunningham project. In the summer of 1988, we implemented the multiple polynomial quadratic sieve algorithm on the same network. On this network alone, we are now able to factor any 100 digit integer, or to find 35 digit factors of numbers up to 150 digits long within one month.

To allow an even wider distribution of our programs we made use of electronic mail networks for the distribution of the programs and for inter-processor communication. Even during the initial stage of this experiment, machines all over the United States and at various places in Europe and Australia contributed 15 percent of the total factorization effort.

At all the sites where our program is running we only use cycles that would otherwise have been idle. This shows that the enormous computational task of factoring 100 digit integers with the current algorithms can be completed almost for free. Since we use a negligible fraction of the idle cycles of all the machines on the worldwide electronic mail networks, we could factor 100 digit integers within a few days with a little more help.

1. Introduction

It is common practice to begin a paper on integer factoring algorithms with a paragraph emphasizing the importance of the subject because of its connection to public-key cryptosystems; so do we. We refer to [3] for more information on this point.

This paper deals with the practical question

how big are the integers we can factor with our present algorithms?

This question is rather vague, because we did not specify how much time and/or money we are willing to spend per factorization. Before making our question more precise, let us illustrate its vagueness with four examples which, in the summer of 1988, represented the state of the art in factoring.

(i) In [7, 24] Bob Silverman *et al.* describe their implementation of the multiple polynomial quadratic sieve algorithm (mpqs) on a network of 24 SUN-3 workstations. Using the idle cycles on these workstations, 90 digit integers have been factored in about six weeks (elapsed time).

(ii) In [21] Herman te Riele *et al.* describe their implementation of the same algorithm on two different supercomputers. They factored a 92 digit integer using 95 hours of CPU time on a NEC SX-2.

(iii) 'Red' Alford and Carl Pomerance implemented mpqs on 100 IBM PC's; it took them about four months to factor a 95 digit integer.

(iv) In [20] Carl Pomerance *et al.* propose to build a special purpose mpqs machine 'which should cost about $20,000 in parts to build and which should be able to factor 100 digit integers in a month.'

J.J. Quisquater and J. Vandewalle (Eds.): Advances in Cryptology - EUROCRYPT '89, LNCS 434, pp. 355-371, 1990.

In all these examples mpqs was being, or will be used as factoring algorithm, as it is the fastest general purpose factoring algorithm that is currently known [19]. In order to compare (i) through (iv), we remark that for numbers around 100 digits adding three digits to the number to be factored roughly doubles the computing time needed for mpqs; furthermore mpqs has the nice feature that the work can be evenly distributed over any number of machines. It follows that 100 digit integers could be factored in about one month, using

- the idle time of a network of 300 SUN-3 workstations, or
- one NEC SX-2, or
- 1200 IBM PC's, or
- one $20,000 special purpose processor.

Returning to our above question let us, in view of these figures, fix one of the resources by stipulating that we want to spend at most one month of elapsed time per factorization. Apparently, the answer then depends on the amount of money we are willing to spend. Clearly, if we use mpqs, and if we start from scratch, then the last alternative is definitely the cheapest of the four possibilities listed above. However, there is no reason at all to start from scratch.

If we really want to find out where our factorization limits lie nowadays, we should take into account that the natural habitat of the average computer scientist has changed considerably over the last few years: many people have access to some small number of small machines, and many of those small machines can communicate with each other by means of electronic mail. What would happen if someone exploited the full possibilities of his environment? The current factorization algorithms, when parallelized, do not require much inter-processor communication. Therefore, electronic mail could easily take care of the distribution of programs and data and the collection of results. So, if someone writes a factorization program, mails it to his friends along with instructions how to run it on the background, and convinces his friends to do the same with their friends up to an appropriate level of recursion, then the originator of the message could end up with a pretty powerful factorization machine. It is not unlikely that he will be able to factor 100 digit integers in much less than one month, and *without spending one single penny*.

Thus we rephrase our question as follows:

> *how big are the integers we can factor within one month of elapsed time, if we only want to use computing time that we can get for free?*

So far, we have only used the multiple polynomial quadratic sieve algorithm (mpqs) in our running time estimates. As we noted above, the reason for this is that mpqs is the fastest *general purpose* factoring algorithm that we currently know of, *i.e.*, it is the fastest algorithm whose running time is, roughly speaking, completely determined by the size of n, and not by any other properties that n might have. Thus, for mpqs we can fairly accurately predict the precise moment at which a factorization will be found, once the computation has been set up. This implies that, if we decide to use mpqs, the answer to the above question depends solely on the amount of computational power we are able to get. Furthermore, if we are able to factor some integer of some given size within a month, then we can factor *any* integer of about that size in about the same amount of time. This holds irrespective of how 'difficult' the number might be considered to factor, like RSA keys which are usually chosen as products of two primes of about the same size [22].

This does not imply that, given an arbitrary integer n to be factored which is not 'too big' for us, we immediately apply mpqs. That only makes sense if one knows that the number in question is 'difficult.' Ordinarily, one should first try methods that are good in finding factors with special properties; examples of such methods are trial division (small factors), Pollard's

'rho'-method (bigger small factors) [15, 18], the $p-1$ and $p+1$-methods and their variants (factors having various smoothness properties) [2, 16], and the elliptic curve method (factors of up to about 35 digits) [13]. The more time one invests in one of these methods, the higher its probability of success. Unfortunately it is difficult to predict how much time one should spend, because the properties of the factors are in general unknown. And, the time invested in an unsuccessful factorization attempt using one of these methods is completely wasted; it only gives a weak conviction that n does not have a factor of the desired property, but no proof (with the exception of trial division, which yields a proof that no factor less than or equal to the trial division bound exists if nothing has been found).

To return again to our question, let us assume that we will use mpqs as the method of last resort. As remarked above, the answer then depends on the free computational power we can organize, since mpqs cannot be lucky (up to a certain minor point fow which we refer to [24]). So, we should concentrate on methods to get as much free computing time as we can, which we do as described above: try to get volunteers on the electronic mail networks to run our program at times that their machines would otherwise be idle. To attract the attention of the maximum number of possible contributors, and to make them enthusiastic about the project, we adopted the following strategy:

- Write an mpqs-program that is as portable as we can make it, and ask some friends and colleagues working in the same field to experiment with it. In that way we should get a good impression of what is possible, and what should be avoided, and we get some experience in running an 'electronic mail multiprocessor' on a small scale.
- Use this program, the forces that we can organize at DEC's Systems Research Center (SRC), and the (still) relatively small external power to achieve some moderately impressive factorization results.
- Publish those results in the sci.math newsgroup to attract attention from possible contributors, and ask for their help.
- Use electronic mail to distribute the program to people who express interest.
- Get more impressive factorization results.
- Repeat the last three steps as long as we are still interested in the project.

Once such a distributed mpqs implementation has been set up it is a minor effort to include some other useful factorization features. For instance, we plan to include the elliptic curve method (ecm) in the package we distribute; we already have considerable experience with a distributed ecm program at SRC, but at the time of writing this paper we had not included it in the program we have distributed worldwide.

Here we should remark that ecm can easily be distributed over any number of machines, as it consists of a number of independent factorization trials. Any ecm trial can be lucky, and find a factorization, independent of any other trial. The probability of success per trial depends on the size of the factor to be found, and is therefore difficult to predict; see Section 2 for details. The mpqs algorithm works completely differently. There the machines compute so-called relations, which are sent to one central location. Once sufficiently many relations have been received, the factorization can be derived at the central location; see Section 3 for details.

Given such an extended package, a typical factorization effort would proceed as follows. Upon receipt of a new number, all machines on the factorization network do some specified number of ecm trials. A successful trial is immediately reported to us, and we broadcast a message to stop the current process. If all ecm trials have failed and the number in question is not too big, the machines move to mpqs and start sending us relations. We then wait until we have sufficiently many relations to be able to derive the factorization.

At the time of writing this paper we have factored two 93, one 96, one 100, one 102, and

one 106 digit number using mpqs, and we are working on a 103 digit number; for all these numbers extensive ecm attempts had failed. The 100 digit number took 26 days (of elapsed time). About 85% of the work of this factorization was carried out at SRC. For the 106 digit number the external machines contributed substantially more than 15% of the total factorization effort, namely 30%. The 103 digit number will be done exclusively by external machines, while the machines at SRC are working on other factorizations. As the factorization network is growing constantly, it is difficult to predict how many machines we will eventually be able to get. Consequently, at the present moment we are still unable to give a reliable answer to our question, but we have the impression that the present approach should enable us to factor 110 digit numbers.

It is a natural question to ask what consequences this could have for the security of the RSA cryptosystem. The well-known RSA challenge is to factor a certain 129 digit number concocted by Rivest, Shamir and Adleman. At the time they presented this challenge, and generously offered to pay $100 for the factorization, it appeared to be far out of reach. The analysis of mpqs however, shows that factoring 129 digit numbers is 'only' about 400 times as hard as factoring 100 digit numbers. Now that 100 digit numbers can be factored, this does not seem like a very secure safety margin. The $100 prize will not be much of an incentive, however: postage costs and the fees for currency conversion will make dividing the spoils counterproductive.

RSA cryptosystems are being used nowadays with keys of 512 bits, which amounts to 155 digits. Factoring 155 digit numbers is about 40000 times harder than factoring 100 digit numbers, which sounds pretty safe. But even given the present factoring algorithms, it is unclear how long this will remain safe. There are zillions of idle cycles around that can be used, as we have seen, by anyone who has access to the electronic mail network; this network is growing rapidly, and its future computational power is difficult to predict. More worrisome is that determined and powerful adversaries could in principle organize some gigantic factoring effort by linking together all their machines. Factoring 155 digit numbers is then not as impossible as many people would like it to be.

These estimates do not even take into account that the average processor will get much faster than it is now. Currently an average workstation operates at 1 to 3 million instructions per second (mips); a fast workstation operates in the range from 6 to 10 mips. Soon new workstations will be released running at 20 to 25 mips, and it is to be expected that within five years a moderately priced workstation will run at 100 mips. The reader can easily figure out what consequences this will have for the safety of 512 bit RSA keys.

Of course, we are not factoring actual RSA keys. For one thing, when we report our previous successes to try to gather helpers to work on the project, we might inadvertently attract the attention of the owner of the key, who might be unhappy at discovering this. Secondly, the only reason to factor RSA keys is to impersonate the owner. The people helping us factor the number might demand their share and we do not know anyone with a small enough key and a large enough bank balance to make this worthwhile. To avoid any suspicion, we factor only numbers from the Cunningham project [6], or other numbers with short algebraic descriptions and of some mathematical significance [5].

Another concern of our helpers might be that they have accepted a Trojan horse [25]. For this reason our implementations are relatively straightforward; helpers might not be able to verify the correctness of the algorithm, but they can certainly see that nothing too strange is going on. We could have written something that is very clever, which can find helpers by itself. Such a program would be a virus [1, 10], and while it might help us in conquering a single large number, it would probably hurt our standing in the community in the long run. Someone who is not concerned about propriety might be less scrupulous.

With all these negative possibilities, why should anyone take the risks of helping us? So

far, the people who helped us are our friends and are not inclined to ascribe base motives to our work. Additionally, we have managed to help them get their names in the newspapers. In the future we will have to make sure that the inconveniences involved in running the program are outweighed by the good feelings generated by whatever share of the limelight we can place our helpers in.

We do not claim that in the long run our approach is the most cost-effective way of factoring large integers on a regular basis. For that purpose we believe that Pomerance's machine mentioned in (iv) above, or a couple of them, is the most promising attack. For the security of RSA cryptosystems, however, we think that our approach of building an ad hoc network is more threatening; there it counts what we *can* do, not what we can do on a regular basis. As we mentioned above, the power of such a network is difficult to estimate, and our setup has the advantage that it can almost immediately profit from any advances in technology (faster machines) or theory (faster algorithms).

The rest of this paper is organized as follows. In Section 2 we give a rough outline of the elliptic curve method and its expected behavior, and we present some of the results we obtained using this method. The same will be done for the multiple polynomial quadratic sieve algorithm in Section 3. Some details of the distribution techniques that we have used and that we are planning to use are given in Section 4.

2. The elliptic curve method

The elliptic curve method (ecm) consists of a number of independent factorization trials. Any trial can be lucky and find a factorization, independent of any other trial. The larger the smallest factor of the number to be factored, the smaller the probability of success per trial. The elliptic curve method is a *special purpose* factoring algorithm in the sense that it can only be expected to work if the number to be factored has a reasonably small factor.

Indeed, the elliptic curve method is a very useful method to find small factors of large numbers. For instance, we will see that if a 100 digit number has a 38 digit factor, it is probably faster to find this factor using ecm than using the multiple polynomial quadratic sieve algorithm. A problem, however, is that one does not know beforehand whether the number to be factored indeed has a small factor. If it has a small factor, then applying ecm has a reasonable probability of success; if there is no small factor, then ecm will have virtually no chance. This naturally leads to the question how much time one should spend on an attempt to factor a number using ecm.

This question would probably be easier to answer if the results from a failed ecm attempt would be useful for other purposes. Unfortunately however, a failed ecm trial does not contribute anything that is useful for the rest of the computation, or that might be helpful for other factorization attempts. Consequently, the time invested in an unsuccessful ecm factorization attempt is completely wasted. It is even the case that, if ecm fails to detect a small factor after some number of trials, this does not guarantee that there *is* no small factor, although the existence of a small factor becomes less 'likely'. Also, the expected remaining computing time grows with the time that has been spent already.

Those less desirable properties are not exactly fitted to make the ecm a very popular method. Who is, after all, willing to spend his valuable cycles on a lengthy computation that will probably not produce anything that is useful. This would not be much of a problem if those cycles are *not* valuable. On big mainframes such cycles might be difficult to find, but on the average workstation they are abundant, as workstations are usually idling at least half of the time. A disadvantage is that one workstation is probably slower than a big mainframe. Many workstations together, however, should give at least the same computational power as one big machine, when applied to a method that can easily be parallelized like ecm.

Therefore, in order to find out what can still be done using ecm, and what is out of reach,

and without getting complaints about wasted computing time, we should run ecm in the background on a large number of workstations. Because we cannot predict the sizes of the factors of the numbers we attempt to factor, our attempts will quite often be fruitless; there is not much of a difference between wasting cycles on unsuccessful ecm attempts or idling, so nobody will complain too seriously. Every now and then we will hit upon a lucky number and find a factorization, thus making ourselves, the contributors of the cycles (we hope), and the authors of [6] (in any case) happy.

In the rest of this section we will discuss some aspects of our MicroVAX implementation which is written in Pascal with VAX assembly language for the multiprecision integer arithmetic (for a wider distribution of the ecm program we have a C-version as well). The ecm is described in detail in [13]; descriptions that focus more on practical aspects and implementations of ecm can be found in [4, 15]. For the purposes of this paper the following rough description of the elliptic curve method suffices.

Elliptic curve method. Given an integer n to be factored, randomly select an elliptic curve modulo n and a point x in the group of points of this elliptic curve.
First stage: Select an integer m_1, and raise x to the power k, where k is the product of all prime powers $\leq m_1$. If this computation fails because a non-trivial factor has been found, then terminate. Otherwise, continue with the second stage.
Second stage: Select an integer $m_2 > m_1$, and try to compute x^{kq} for the primes q between m_1 and m_2 in succession. If this computation fails because a non-trivial factor has been found, then terminate. Otherwise, start all over again.

Given the sizes of the m_i and the smallest factor p of n, the probability that one iteration is successful in factoring n can be derived. Given a choice for the m_i, the expected number of iterations to find a factor of a certain size with a certain probability then easily follows. Asymptotically the expected running time of ecm to find a factor p of n is $O((\log n)^2 e^{(1+o(1))\sqrt{2}\log p \log\log p})$, summed over all trials. Every iteration, or trial, of ecm is completely independent of every other trial. This means that we can expect to achieve an s fold speed-up by running the ecm program on s independent identical machines, as long as we make sure that those machines make different random choices.

The second stage as formulated here has the disadvantage that it needs a table containing the differences between the consecutive primes up to m_2. For huge values of m_2 this might become problematic, especially in a set-up where memory efficiency is an important aspect. Therefore we use the so-called *birthday paradox* version for the second stage as described in [4, section 6]. Our implementation also incorporates other ideas mentioned in the same paper which make it even more time and space efficient. We refer to [4, sections 7, 9.1, 9.3, and 9.4] for a description of these improvements.

Given the program and given the relative speeds of the two stages, it remains to analyze how the parameters should be chosen (and modified) during execution. For this purpose it is useful first to optimize those choices, *given* the size of the smallest prime factor p of n. For a fixed size of p the optimum parameter choice can be found by slightly changing the analysis given in [4] to take the various improvements into account. The resulting analysis is somewhat different from the one in [4], but it is sufficiently similar that we do not give any details of our computation and instead refer to [4].

In this paper we are only interested in the resulting running time estimates on a MicroVAX II processor. Our unit of work is one multiplication in $\mathbf{Z}/n\mathbf{Z}$, i.e., one multiplication of two numbers in $\{0, 1, \ldots, n-1\}$ followed by a reduction modulo n. This operation can be performed at the cost of roughly two multiplications of integers of about the same size as n by representing the numbers as suggested in [14]. This representation of integers resulted in a 20 percent speed-up, as compared to the ordinary representation. On a MicroVAX II processor,

which operates at about 1 mips, one unit of work for a 100 digit number takes about 0.0045 seconds.

The amount of work needed to find a factor of a certain size with probability at least 60 percent is given in Table 1. We have also indicated how many millions of seconds this would take on a 1 mips computer for various sizes of n; notice that 30 million seconds is about one year. Furthermore, the table lists the optimal values for m_1 in the first stage, and the number of iterations to achieve a 60 percent probability of success; it appeared that the effective value for m_2 for our implementation is about $30 \cdot m_1$.

Table 1

Amount of work, optimal parameter choice, and millions of seconds on a 1 mips computer to find smallest factor p of n with a success probability of 60 percent

$\log_{10}p$	$\log_{10}$work	m_1	trials	millions of seconds for $\log_{10}n$ =			
				80	90	100	110
25	8.6	65000	300	1	2	2	2
26	8.9	85000	400	2	3	3	4
27	9.1	115000	500	3	4	5	6
28	9.3	155000	650	6	7	9	11
29	9.5	205000	750	9	12	15	18
30	9.7	275000	950	15	20	24	28
31	9.9	360000	1100	25	30	37	45
32	10.1	480000	1350	40	50	60	72
33	10.3	625000	1600	60	75	95	115
34	10.5	825000	1950	95	120	150	180
35	10.7	1100000	2300	150	190	230	280
36	10.9	1400000	2800	230	290	360	430
37	11.1	1800000	3300	350	450	550	670
38	11.3	2350000	3900	540	680	840	1020
39	11.5	3000000	4700	830	1040	1290	1560
40	11.6	3850000	5600	1300	1600	1960	2400

What does this mean for a network of approximately 100 workstations, each consisting of five MicroVAX processors? Assuming that the average workstation is idle from 5.00 PM to 9.00 AM, we should be able to get about 28 million seconds per weekday. This should enable us to find 30 digit factors of 100 digit numbers in about one day. During the weekend we expect to work 1.5 times faster.

The numbers we attempted to factor all came from the appendix of unfactored numbers from the Cunningham Tables [6, 8]. In Table 2 we list some of the most and more wanted numbers from the Cunningham Tables we factored with ecm. As the 'expected time' in Table 2 is based on a 60 percent probability of success, it is only a very rough indication. Furthermore, the term *expected time* is misleading, because we can only expect that time *given* the resulting factorization.

Table 2
Some factorizations obtained with ecm
(time in millions of seconds on a 1 mips computer)

$\log_{10} n$	$\log_{10} p$	time expected	time observed	elapsed days	n is factor of
99.2	26.1	3	47	2.4	$10^{137}-1$
96.3	26.3	3	5	0.3	$11^{97}+1$
103.4	27.6	7	3	0.3	$3^{229}+1$
100.0	27.7	8	6	0.5	$10^{101}-1$
100.8	28.8	14	15	0.5	$7^{137}-1$
98.0	30.7	32	25	1.4	$6^{137}+1$
89.7	31.6	40	15	0.8	$2^{361}-1$
105.2	31.7	60	21	0.9	$10^{143}-1$
100.3	33.5	120	21	1.2	$2^{353}-1$
88.6	33.9	110	130	5.7	$5^{157}-1$
93.7	34.2	160	31	1.8	$10^{116}+1$
87.8	35.6	240	89	4.1	$2^{353}+2^{127}+1$

As might be clear from this table, the elliptic curve method is indeed very useful to find small factors, and it does so in a reasonable amount of time. Except for the first two entries, we have been quite lucky several times.

Of course, there were numerous failures as well. Some of the failures for which the factorization was later found by others (or by ourselves) are listed in Table 3. The meaning of the 'expected time' column in Table 3 is: 'the time ecm should have spent for a 60 percent probability of success'. As usual, p denotes the smallest prime factor of n.

Table 3
Some failed ecm attempts
(time in millions of seconds on a 1 mips computer)

$\log_{10} n$	$\log_{10} p$	time expected	time spent	elapsed days	n is factor of	factor found by
79.4	29.4	11	23	1.4	$2^{483}+1$	Silverman, mpqs
91.4	33.1	83	130	7	$6^{131}-1$	te Riele, mpqs
92.1	33.1	84	140	7.4	$2^{368}+1$	Ruby, ecm
105.8	36.6	530	130	7	$2^{353}+1$	see below, mpqs
92.4	37.1	570	130	7	$10^{109}+1$	see below, mpqs
89.5	40.7	>1500	160	9	$5^{160}+1$	Silverman, mpqs
100.0	40.9	>2000	145	8	$11^{104}+1$	see below, mpqs
94.6	43.2	>10000	130	7	$2^{332}+1$	Alford, mpqs
101.1	46.4	>10000	130	7	$2^{391}-1$	see below, mpqs
95.3	47.5	>10000	270	14	$11^{107}-1$	see below, mpqs

For the first three entries of Table 3 we clearly have had some bad luck; failure of ecm does not at all imply that there is no small factor. The last five entries were not actual failures, in the sense that the smallest factor was too big for ecm to find.

In practice it appears that ecm is relatively insensitive to the choice of the parameters. If m_1 is chosen too large for the (unknown) smallest prime factor of n, the probability of success per curve increases, and consequently the number of trials decreases. And vice versa, if m_1 is too

small, the probability per curve decreases, and the number of trials increases. In both cases the product of m_1 and the number of trials, which is up to a constant a good indication for the work done, will be close to optimal. This practical observation is supported by the theoretical analysis. For instance, if one uses the parameters that are optimal for $\log_{10} p = 30$ for an n that happens to have a 32 digit smallest factor, the optimal $\log_{10}$work of 10.12 increases to only 10.13 (i.e., 2400 trials with $m_1 = 275000$ instead of 1350 trials with $m_1 = 480000$).

Consequently, the parameter choices do not matter too much, at least within certain limits. In our experience $m_1 = 1.001^{i-1} \cdot 300000$ at the ith trial worked satisfactorily, as did slightly larger (and smaller) choices for the growth rate and m_1.

3. The multiple polynomial quadratic sieve algorithm

The quadratic sieve algorithm is described in [19]. Descriptions of the multiple polynomial variation of the quadratic sieve algorithm (mpqs) can be found in [20, 24].

Like the elliptic curve method, the quadratic sieve algorithms are probabilistic algorithms. But unlike ecm, the quadratic sieve algorithms do not depend on certain properties of the factors to be found. Furthermore, their success does not depend on a *single* lucky choice, which, as we have seen in Section 2, is sometimes unlikely ever to occur. Instead, they work by combining many small instances of luck, each of which is much more likely to take place.

Once the computation has been set up, one can easily see how 'lucky' the method is on the average, the progress can easily be monitored, and the moment it will be completed can fairly accurately be predicted. This is completely independent of how 'difficult' the number to be factored is considered to be.

The quadratic sieve algorithms are called *general purpose* factoring algorithms because, up to a minor detail (cf. [24]), their run time is solely determined by the size of the number to be factored. Their expected running time to factor a number n is $e^{(1+o(1))\sqrt{\log n \log\log n}}$; this is independent of the sizes of the factors to be found. Notice that this is the same as the asymptotic running time of the elliptic curve method if the smallest factor p is close to $\sqrt{n}$.

The quadratic sieve algorithms consist of two stages, a time consuming first stage to collect so-called *relations*, and a relatively easy second stage where the relations are combined to find the factorization. In the multiple polynomial variation of the quadratic sieve algorithm the first stage can easily be distributed over almost any number of processors, in such a way that running the algorithm on s identical processors at the same time results in an s fold speed-up. In the second stage the combination is found using Gaussian elimination, which is usually done on one machine.

The following rough description might be helpful to understand the factorization process. Let n be the number to be factored. First one chooses an integer $B > 0$ and a factor base $\{p_1, p_2, ..., p_B\}$, consisting of $p_1 = -1$ and the first $B-1$ primes p for which n is a square modulo p. Next one looks for relations, which are expressions of the form

$$v^2 \equiv q^t \cdot \prod_{j=1}^{B} p_j^{e_j} \bmod n,$$

for $t \in \{0,1\}$, $e_j \in \mathbf{Z}_{\geq 0}$, and q a prime not in the factor base. Below we will explain how these relations can be found. We will call a relation with $t = 0$ a *small* relation, and a relation with $t = 1$ a *partial* relation. Two partial relations with the same q can be combined to yield a relation of the form

$$w^2 \equiv q^2 \cdot \prod_{j=i}^{B} p_j^{e_j} \bmod n,$$

(where w is the product of the v's of the two partial relations, and the e_j are the sums of the exponents of the two partial relations); such a relation will be called a *big* relation. For n

having 90 to 110 digits B will approximately range from 17000 to 100000.

Given a total of more than B small and big relations, we will be able to find a dependency among the exponent vectors $(e_j)_{j=1}^{B}$, and therefore certainly a dependency modulo 2 among the exponent vectors modulo 2; this can be done by means of Gaussian elimination on a bit matrix having B columns and $> B$ rows. Such a dependency modulo 2 then leads to a solution x, y to $x^2 \equiv y^2 \bmod n$. Given this solution, one has a reasonable probability of factoring n by computing $\gcd(x-y, n)$. More relations lead to more dependencies, which will lead to more pairs x, y. Therefore we can virtually guarantee to factor n, if the relations matrix is slightly over-square.

There are various ways to generate relations. We will show how small relations can be generated; how partial relations can be generated easily follows from this. In the original description of the quadratic sieve algorithm, Pomerance proposes to use the quadratic polynomial $f(X) = ([\sqrt{n}\,]+X)^2-n$, and to look for at least $B+1$ integers m such that $f(m)$ is smooth; here we say that an integer is smooth if it can completely be factored over the factor base $p_1, p_2, \ldots, p_B$. Because $f(m) \equiv ([\sqrt{n}\,]+m)^2 \bmod n$, a smooth $f(m)$ produces a relation. From $f(m) \approx 2m\sqrt{n}$ (for small m), and the heuristic assumption that the $f(m)$'s behave approximately as random numbers with respect to smoothness properties, we can derive an integer m_B such that we expect that there are at least $B+1$ distinct m's with $|m| \leq m_B$ for which the corresponding $f(m)$'s are smooth.

Given a list of values of $f(m)$ for $|m| \leq m_B$, those that are smooth can be found as follows. Let $p > 2$ be a prime in the factor base that does not divide n, then the equation $f(X) \equiv 0 \bmod p$ has two solutions $m_1(p)$ and $m_2(p)$ (because n is a square modulo p), which can easily be found. But $f(m_i(p)) \equiv 0 \bmod p$ implies that $f(m_i(p)+kp) \equiv 0 \bmod p$ for any integer k. To find the $f(m)$ on our list that are divisible by p it suffices therefore to consider the locations $m_i(p)+kp$ for all integers k such that $|m_i(p)+kp| \leq m_B$ and $i = 1, 2$. Finding the $f(m)$ that are smooth can therefore be done by performing this so-called *sieving* for all p in the factor base.

Here we should remark that in practice one does **not** set up a list of $f(m)$'s for $m = -m_B, \ldots, -1, 0, 1, \ldots, m_B$, to divide the values at appropriate locations by the primes. Instead a list of zero's is set up, to which approximations to the logarithms of the primes are added. For locations m which contain sufficiently big numbers after sieving with all primes, one computes and attempts to factor $f(m)$; how big this value should be also depends on the maximal size of the big prime (the q) one allows in a partial factorization (see above), as $f(m)$'s that do not factor completely possibly lead to a partial relation. Of course, the whole interval will not be processed at the same time, but it will be broken up into pieces that fit in memory.

A heuristic analysis of this algorithm leads to the expected asymptotic running time mentioned above. The algorithm could in principle be parallelized by giving each machine an interval to work on. Several authors (Davis in [9] and Montgomery in [24]) suggested a variation of the original quadratic sieve algorithm which is not only better suited for parallelization, but which also runs faster (although the asymptotic running time remains $e^{(1+o(1))\sqrt{\log n \log\log n}}$). Instead of using one polynomial to generate the smooth residues, they suggest using many different suitably-chosen polynomials. Evidently, this makes parallelization easier, as each machine could work on its own polynomial(s). Another advantage of using many polynomials is that one can avoid the performance degradation that mars the original quadratic sieve algorithm ($|f(m)|$ grows linearly with $|m|$, so that the probability of smoothness decreases as $|m|$ grows), by considering only a fixed interval per polynomial before moving to the next polynomial. This more than outweighs the disadvantage of having to initialize many polynomials. We used the multiple polynomial variation as suggested by Montgomery; for details we refer to [11] and [24].

To determine the optimal value of B, the size of the factor base, we have to balance the advantages of large B's (many smooth residues) against the disadvantages (we need at least $B+1$ smooth residues, they are quite costly to find, and the final reduction gets more expensive). The crossover point is best determined experimentally; in practice anything that is reasonably close to the optimal value works fine.

This does not imply that we will always use a B that is close to optimal. For n around 103 digits, the optimal B gets so large that it would cause various memory problems. In the first place the memory required by the program to generate relations grows linearly with B. In our setup we should also be able to run the program on small workstations, which implies that B cannot get too big. In the second place, storing all relations that we receive for large B (in particular the partial relations) gets problematic. And finally, storing the matrix during the Gaussian elimination becomes a problem (apart from the fact that the elimination gets quite slow, see below). As a consequence we will be using suboptimal values of B for n having more than 103 digits, at least until we have solved these problems. See below for the values of B that we have used for various n.

During the first stage of the algorithm, one not only collects small relations, residues that factor completely over the factor base, but one looks for partial relations as well, i.e., residues that factor over the factor base, except for a factor $q > p_B$. The reason for this is, as we have seen above, that two partial relations with the same q can be combined into one so-called big relation, which is just as useful as one small relation during the final combination stage. The more partials we can get, the more bigs we will find, thus speeding up the first stage of the algorithm. But evidently, one needs quite some number of partial relations to have a reasonable probability to find two with the same q (cf. birthday paradox).

In principle we could easily keep all partials with q prime and $q < p_B^2$, and collect those during the first stage. Doing that would however not be very practical, as we would get an enormous number of partial relations. Furthermore, most matches will be found among the smaller q's. For that reason, we only keep partial relations for which q is not too big. An upper bound of 10^8 on q works quite satisfactorily. With this bound we found about fifteen times as many partials as smalls for $B = 50000$, which at the end of the first stage resulted in about three big relations for every two smalls. For $B = 65500$ and the same bound on q we got about thirteen partials per small relation, and at the end we had five bigs for every three smalls.

The smalls and partials we receive (and generate) at SRC are processed in the following straightforward way. About once every few days the new relations are verified for correctness. This verification is done for obvious reasons: who knows what people send us, and who knows what mailers do with the messages they are supposed to send; we never received faulty relations that were in the right format, but we got quite some totally incomprehensible junk, badly mutilated by some mailer. After verification, the smalls are sorted by their weight, and merged into the file of 'old' small relations, thereby deleting the double relations (sometimes we get one message several times). The partials are sorted by their q value, and merged into the sorted file of 'old' partials. If we find two relations with the same q, one is kept in or inserted in the sorted file of partial relations, and the combination of the two is computed and merged into the file of big relations. Again, doubles are deleted during the merging, and a combination of two identical partial relations is rejected. Of course, during these processes, we check to see if we are so lucky as to have found a small or big relation with even exponents, as such a relation could immediately lead to a factorization; we have not been so lucky yet, as was to be expected.

Figure 1 illustrates the progress of the first stage for the factorization of a 100 digit number with $B = 50000$ and $q \leq 10^8$. The first stage started at noon, September 15, 1988, and it was completed on October 9, when we reached a total of more than 50000 small and big relations.

It can be seen that the number of big relations grows faster and faster: the more partial relations we get, the higher probability we have to find a double q, and therefore a big relation. The number of small relations grows more or less linearly, as was to be expected. Among the about 320000 relations we had found on October 9 there were about 20500 smalls, and the remaining partials produced about 29500 big relations. For the other numbers we factored the graphs of the numbers of small and big relations behaved similarly to Figure 1.

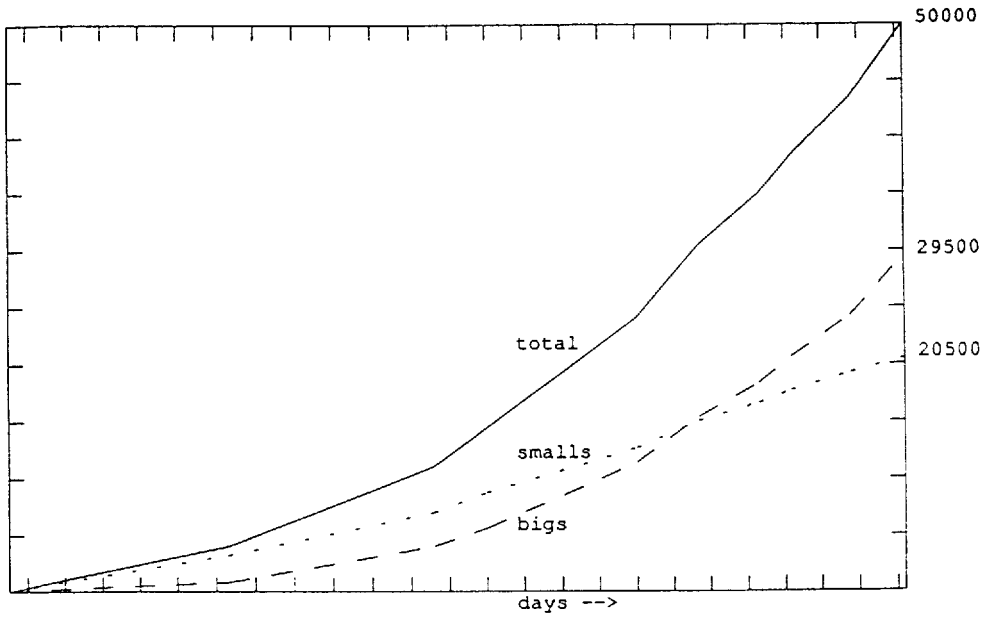

Figure 1

Given these figures, it will not come as a surprise that, once the first stage of the mpqs program has been set up on some number of machines (and keeps running on those machines!), the moment that the total number of small and big relations exceeds B can fairly accurately be predicted. As mentioned above, it then remains to find a dependency modulo 2 among the exponent vectors of the small and big relations. For our initial experiments we wrote a straightforward Gaussian elimination algorithm [12], which was perfectly able to handle the sparse matrices resulting from factor bases with B up to about 35000 that we had chosen for the smaller numbers we factored. For larger B our algorithm definitely gets very slow; for B = 50000 the Gaussian elimination took for instance about 1.5 days on a VAX 8800. We are still working on the implementation of more advanced methods for the Gaussian elimination, like Wiedemann's method [26] and extensions of Odlyzko's intelligent Gaussian elimination techniques [17]. We should note that we never needed $B+1$ rows in the matrix to produce a factorization; it appeared that about $0.99 \cdot B$ is enough in practice.

We conclude this section with some of the results we obtained with the multiple polynomial quadratic sieve algorithm. Because we wanted to be able to run the program at as many sites as possible, we decided to write our programs completely in C. Of course this causes some inefficiencies, because we did not use any assembly language code. We could have provided such code for various popular types of machines, but we did not do that yet.

While carrying out our experiments, we attempted bigger and bigger numbers. Bigger numbers have two important advantages: collecting relations takes at least a few weeks, so that

we do not have to change our (worldwide) inputs often, and the resulting factorizations attract more attention, and therefore more contributors. In this way we collected relations for the factorizations of the numbers mentioned in Table 4; as usual, p denotes the smallest prime factor of n. All numbers in Table 4 were most or more wanted in the Cunningham Tables [6]. At the time of writing this paper we are collecting relations for a 103 digit number, the last entry in Table 4.

Table 4
Some factorizations obtained with mpqs

$\log_{10} n$	$\log_{10} p$	B	elapsed days	external help	n is factor of
92.1	35.3	25000	13	≤ 1 %	$7^{139}+1$
92.4	37.1	25000	11.5	5 %	$10^{109}+1$
95.3	47.5	37900	24	5 %	$11^{107}-1$
100.0	40.9	50000	26	15 %	$11^{104}+1$
101.1	46.4	65500	45	21 %	$2^{391}-1$
105.8	36.6	65500	120	30 %	$2^{353}+1$
102.88	?	65500	?	100 %	$10^{122}+1$

4. Distributing the factorization process

In the summer of 1987 we implemented elliptic curve factorization, distributed over the network of approximately 100 Firefly workstations in use at SRC. The overall structure of the distribution was similar to Silverman's system for quadratic sieve: one central system coordinates the factorization, searching out idle systems and supplying them with tasks. The internal structure is quite different, however.

To support research in distributed computation and recompilation, Ellis [23] built 'mi' and 'dp' — the machine information and distant process servers. 'mi' keeps track of the utilization of every workstation, the period of time since the last keystroke or button press, and an availability predicate provided by the owner of the workstation. Most machines have the default predicate, which allows distant processing only if the machine has been unused for at least half an hour on weekends or in the evening, or two hours during the workday. The distant process mechanism starts a program on a selected machine, transparently connecting open file descriptors for standard input and output back to the originating workstation.

In addition, the standard SRC environment includes a stub compiler for remote procedure calls. One constructs the interface in our standard programming language and supplies an implementation of the service. 'flume', the stub compiler, and the RPC runtime system then allow client programs to make calls to the server just by calling procedures in the interface. The RPC runtime includes provisions for reasonably prompt notification in the event that the client address space or workstation crashes.

Using these facilities, our factoring program was easy to write. We constructed a standalone executable which, when supplied with a random seed and an exponent bound, runs a trial of the elliptic curve algorithm. If it discovered a factorization, it printed it and exited. If it received a Unix interrupt signal, it printed a snapshot of the current state of the computation. The program accepted a description of a partially completed curve as an input to resume factorization. We then needed to invoke this using the distant process machinery.

However, Fireflies are multiprocessors; each has five processors. Thus, we wanted our program to run multiple times on each Firefly.

To facilitate this, we wrote a driver program which spawns one factoring task per processor.

The driver program keeps the workstation busy by using RPC to request a factorization task whenever some processor is idle. It also reports factorizations and snapshots to the central server. Additionally, this program watches for the workstation to come back into use. Should this happen, we take one of two actions. If the owner of the workstation has returned and pressed a button on his mouse or keyboard, we signal the workers to exit, collect their final reports, and exit the driver. If another distant process appears on the workstation, we defer to it by suspending execution of the workers until it completes.

The main program services requests for tasks and records reports of factorizations and snapshots. Periodically, it queries the 'mi' server for the list of all machines which are available, and some statistics about the machines. For each machine which is not currently running a factoring driver, and which has enough free virtual memory that we won't cause the machine to abort user programs, 'dp' is used to invoke a driver.

Using this setup, we accumulated an average computing power roughly equivalent (for our purposes) to a dedicated Cray-1. Following a processor upgrade to the Fireflies, we find ourselves running at a continuous rate of roughly 500 mips. This is sufficient, as reported above, for the discovery of 30+ digit factors in a day.

Subsequently, we came to the realization that our needs could easily be satisfied by a much looser communications system. Our bandwidth requirements are small for elliptic curve -- we just need to send out the number to be factored, and collect the factorization if it ever appears. In particular, we realized that the current electronic mail networks should easily handle the traffic needed to run an elliptic curve factorization.

Political and technical realities made it preferable to attempt this with quadratic sieve, instead. As described above, elliptic curve is much more sensitive to the quality of implementation, and needs to be carefully tuned to run well on a given machine. Moreover, although the rate of progress is slower with quadratic sieve, it is certain, and the factorizations are impressive.

In the summer of 1988, we implemented a new worker program that runs quadratic sieve, printing the relations it finds. The driver and central server were modified to handle both elliptic curve and quadratic sieve. We used this setup to factor a 93 digit number and gain experience with the complete quadratic sieve process.

The worker was also capable of being run by hand and sending results by electronic mail. We distributed the program to a few of our friends, and asked them to tell us how it failed to be sufficiently portable.

We then refined the program, and factored a 96 digit number with more help from outside SRC. We then factored a 100 digit number, with the assistance of most of the prominent members of the factoring community. At this point, everything seemed sufficiently stable that we started to develop an analogue of the driver program that was simple and portable, which we announced the availability of on netnews, along with our most recent results.

The current implementation consists of the quadratic sieving program, some shell scripts, and the utilities we need to check the relations we receive for correctness and to find linear combinations that lead to factorizations.

At SRC, we have a special user account, factor@src.dec.com, that receives all the mail containing relations, requests for copies of the program, and requests for lists of tasks. A shell script examines the mail as it arrives, processing it as appropriate. This takes advantage of the feature of the Berkeley mail delivery program, which allows mail to be delivered as input to a user program.

At helper sites, we run a script that is a simple infinite loop, waking up once each minute. It checks that the factoring process is still running. If not, it waits a little while (in case it failed due to some persistent hardware problem) and starts a new one. It fetches the task to perform out of a list which can be shared by many computers, if the list is on a file system

which can be read and written by those computers. The script also checks that the load average on the machine isn't too high, and that there's no recent typing activity by users. If either of these fails, the worker is suspended. On Unix systems, the script periodically enqueues itself in the 'at' queue, so that the program will survive a crash of the machine.

While these scripts are not particularly elegant, they are sufficiently portable that we have workers running on almost every known type of Unix system. A simple variant of these scripts also runs on VMS. As of this writing, we find that we have access to roughly 1000 mips of sustained computing power for factoring. We have not yet run up against the limits of what the mail systems will handle; we estimate that we could easily handle a factor of 10 more mail without imposing a burdensome load on our mail transport system. This would allow us to factor a 103 digit number in about a week. Since the number of messages is largely independent of the number being factored, we can factor larger numbers in roughly the same amount of time, if we can get our hands on enough computers; we might need to stretch the computation out to a month to factor 130 digit numbers if we didn't want to add load to the mail system.

Acknowledgments. Many thanks to John Ellis for building 'dp' and 'mi'. Without 'dp' and 'mi' we would not even have started this factorization project. Also many thanks to the people at SRC who made their machines available or shut them off for distant processes; they provided us with lots of cycles, and feedback about things we should **not** do. The first author would like to thank SRC for its great hospitality and support during the summers of 1987 and 1988.

Acknowledgments are due to Sam Wagstaff who tirelessly provided us with his endless lists of 'most wanted', 'more wanted', and 'most wanted unwanted' numbers.

And then it is our pleasure to acknowledge the support of our colleagues on the worldwide factorization network. At the time of writing this paper, the following people had donated their cycles: Dean Alvis, Indiana University at South Bend, South Bend, Indiana; Bean Anderson, Peter Broadwell, Dave Ciemiewicz, Tom Green, Kipp Hickman, Philip Karlton, Andrew Myers, Thant Tessman, Michael Toy, Silicon Graphics, Inc., Mountain View, California; Per Andersson, Christopher Arnold, Lennart Augustsson, Chalmers University of Technology, Gothenburg, Sweden; Butch Anton, Hewlett-Packard, Cupertino, California; Greg Aumann, Telecom Australia Research Laboratories, Melbourne, Australia; Gregory Bachelis, Robert Bruner, Wayne State University, Detroit, Michigan; Bob Backstrom, Australian Nuclear Science & Technology Organization, Lucas Heights, Australia; Richard Beigel, The Johns Hopkins University, Baltimore, Maryland; Anita Borg, Sam Hsu, Richard Hyde, Richard Johnsson, Bob Mayo, Joel McCormack, Louis Monier, Donald Mullis, Brian Reid, Michael Sclafani, Richard Swan, Terry Weissman, DEC, Palo Alto, California; Richard Brent, Australian National University, Canberra, Australia; Brian Bulkowski, Brown University, Providence, Rhode Island; John Carr, Jonathan Young, MIT, Cambridge, Massachusetts; Enrico Chiarucci, Shape Technical Centre, The Hague, The Netherlands; Ed Clarke, United Kingdom; Derek Clegg, Island Graphics Corporation, San Rafael, California; Robert Clements, BBN Advanced Computers Inc., Cambridge Massachusetts; Chris Cole, Peregrine Systems, Inc., Irvine, California; Leisa Condie, Australia; Bob Devine, DEC, Colorado Springs, Colorado; John ffitch, Dave Hutchinson, University of Bath, Bath, United Kingdom; Seth Finkelstein, MathSoft, Inc., Cambridge, Massachusetts; Mark Giesbrecht, University of Toronto, Toronto, Canada; Peter Graham, University of Manitoba, Winnipeg, Canada; Ruud Harmsen, ?, The Netherlands; Paw Hermansen, Odense University, Odense, Denmark; Phil Hochstetler, ?; Robert Horn, Memotec Datacom, North Andover, Massachusetts; Scott Huddleston, Tektronix, Beaverton, Oregon; Bob Johnson, Logic Automation Incorporated, Beaverton, Oregon; Arun Kandappan, University of Texas, Austin, Texas; Jarkko Kari, Valtteri Niemi, University of Turku, Turku, Finland; Gerard Kindervater, Erasmus Universiteit, Rotterdam, The Netherlands; Albert Koelmans, University of Newcastle upon Tyne, Newcastle, United Kingdom; Emile LeBlanc, Hendrik Lenstra, Kenneth Ribet, Jim Ruppert, University of California, Berkeley, California; Andries Lenstra, Amsterdam, The Netherlands; Paul Leyland, Oxford University, Oxford, United Kingdom; Walter Lioen, Herman te Riele, Dik Winter, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands; David Lilja, Soren Lundsgaard, Kelly Sheehan, University of Illinois, Urbana-Champaign, Illinois; Barry Lustig, Advanced Decision Systems, Mountain View,

California; Ken Mandelberg, Emory University, Atlanta, Georgia; Peter Montgomery, University of California, Los Angeles, California; Francois Morain, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France; Marianne Mueller, Sun, Mountain View, California; Duy-Minh Nhieu, John Rogers, University of Waterloo, Waterloo, Canada; Gary Oberbrunner, Thinking Machines Corp., Cambridge, Massachusetts; Andrew Odlyzko, AT&T Bell Laboratories, Murray Hill, New Jersey; Michael Portz, Rheinisch-Westfaelische Technische Hochschule Aachen, Aachen, West-Germany; Jim Prescott, Rochester?; Ulf Rehmann, Universität Bielefeld, Bielefeld, West Germany; Mike Rimkus, The University of Chicago, Chicago, Illinois; Mark Riordan, Michigan State University, East Lansing, Michigan; Michael Rutenberg, Reed College, Portland, Oregon; Rick Sayre, Pixar, San Rafael, California; Charles Severance, ?; Jeffrey Shallit, Dartmouth College, Hanover, New Hampshire; Hiroki Shizuya, Tohoku University, Sendai, Japan; Robert Silverman, the MITRE Corporation, Bedford, Massachusetts; Malcolm Slaney, Apple, Cupertino, California; Ron Sommeling, Victor Eijkhout, Ben Polman, Katholieke Universiteit Nijmegen, Nijmegen, The Netherlands; Bill Sommerfeld, Apollo Division of Hewlett Packard, Chelmsford, Massachusetts; Mitchell Spector, Seattle University, Seattle, Oregon; Pat Stephenson, Cornell University, Ithaca, New York; Nathaniel Stitt, ?, Berkeley-area?, California; Marc-Paul van der Hulst, Universiteit van Amsterdam, Amsterdam, The Netherlands; Jos van der Meer, Sun Nederland, Amsterdam, The Netherlands; Frank van Harmelen, Edinburgh University, Edinburgh, Scotland; Brick Verser, Kansas State University, Manhattan, Kansas; Charles Vollum, Cogent Research, Inc., Beaverton, Oregon; Samuel Wagstaff, Purdue University, West-Lafayette, Indiana;

Our sincere apologies to everybody who contributed but whose name does not appear on the above list. Please let us know! Keeping track of the names of all our helpers proved to be the hardest part of this project.

References

1. L. Adleman, "The theory of computer viruses," Proceedings Crypto 88, 1988.
2. E. Bach. J. Shallit, "Factoring with cyclotomic polynomials," *Proceedings 26th FOCS*, 1985, pp 443-450.
3. G. Brassard, *Modern Cryptology*, Lecture Notes in Computer Science, vol. 325, 1988, Springer Verlag.
4. R.P. Brent, "Some integer factorization algorithms using elliptic curves," Australian Computer Science Communications v. 8, 1986, pp 149-163.
5. R.P. Brent, G.L. Cohen, "A new lower bound for odd perfect numbers," Math. Comp., *to appear.*
6. 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, vol. 22, Providence: A.M.S., 1988.
7. T.R. Caron, R.D. Silverman, "Parallel implementation of the quadratic sieve," J. Supercomputing, v. 1, 1988, pp 273-290.
8. A.J.C. Cunningham, H.J. Woodall, *Factorisation of* $(y^n \mp 1)$. $y = 2, 3, 5, 6, 7, 10, 11, 12$ *up to high powers (n)*, London: Hodgson (1925).
9. J.A. Davis, D.B. Holdridge. "Factorization using the quadratic sieve algorithm," Sandia National Laboratories Tech Rpt. SAND 83-1346, December 1983.
10. P.J. Denning, "The Science of Computing: Computer Viruses," *American Scientist*, v. 76, May-June 1988.
11. A.K. Lenstra, H.W. Lenstra, Jr, "Algorithms in number theory," in: J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, D. Perrin (eds.), *Handbook of theoretical computer science*, to appear; report 87-8, The University of Chicago, Department of Computer Science, May 1987.
12. A.K. Lenstra, M.S. Manasse, "Compact incremental Gaussian elimination over Z/2Z," report 88-16, The University of Chicago, Department of Computer Science, October 1988.

13. H.W. Lenstra, Jr., "Factoring integers with elliptic curves," *Ann. of Math.*, v. 126, 1987, pp. 649-673.

14. P.L. Montgomery, "Modular multiplication without trial division," *Math. Comp.*, v. 44, 1985, pp 519-521.

15. P.L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Math. Comp.*, v. 48, 1987, pp 243-264.

16. P.L. Montgomery, R.D. Silverman, "An FFT extension to the p-1 factoring algorithm," manuscript, 1988.

17. A.M. Odlyzko, "Discrete logarithms and their cryptographic significance," pp. 224-314; in: T. Beth, N. Cot, I. Ingemarsson (eds), *Advances in cryptology*, Springer Lecture Notes in Computer Science, vol. 209, 1985.

18. J.M. Pollard, "A Monte Carlo method for factorization," *BIT*, v. 15, 1975, pp 331-334.

19. C. Pomerance, "Analysis and comparison of some integer factoring algorithms," pp. 89-139; in: H.W. Lenstra, Jr., R. Tijdeman (eds), *Computational methods in number theory*, Mathematical Centre Tracts 154, 155, Mathematisch Centrum, Amsterdam, 1982.

20. C. Pomerance, J.W. Smith, R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm," *SIAM J. Comput.*, v. 17, 1988, pp. 387-403.

21. H.J.J. te Riele, W.M. Lioen, D.T. Winter, "Factoring with the quadratic sieve on large vector computers," report NM-R8805, 1988, Centrum voor Wiskunde en Informatica, Amsterdam.

22. R.L. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM.*, v. 21, 1978, pp. 120-126.

23. E. Roberts, J. Ellis, "parmake and dp: Experience with a distributed, parallel implementation of make," Proceedings from the Second Workshop on Large-Grained Parallelism, Software Engineering Institute, Carnegie-Mellon University, Report CMU/SEI-87-SR-5, November 1987.

24. R.D. Silverman, "The multiple polynomial quadratic sieve," *Math. Comp.*, v. 48, 1987, pp. 329-339.

25. K. Thompson, "Reflections on Trusting Trust," *Commun. ACM*, v. 27, 1984, pp. 172-80.

26. D.H. Wiedemann, "Solving sparse linear equations over finite fields," *IEEE Transactions on Information Theory*, v. 32, 1986, pp. 54-62.