Some Parallel Algorithms for Integer Factorisation

Richard P. Brent

Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK
rpb@comlab.ox.ac.uk
http://www.comlab.ox.ac.uk/
oucl/people/richard.brent.html

Abstract. Algorithms for finding the prime factors of large composite numbers are of practical importance because of the widespread use of public key cryptosystems whose security depends on the presumed difficulty of the factorisation problem. In recent years the limits of the best integer factorisation algorithms have been extended greatly, due in part to Moore's law and in part to algorithmic improvements. It is now routine to factor 100-decimal digit numbers, and feasible to factor numbers of 155 decimal digits (512 bits). We describe several integer factorisation algorithms, consider their suitability for implementation on parallel machines, and give examples of their current capabilities.

1 Introduction

Any positive integer N has a unique *prime power decomposition*

$$N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

($p_1 < p_2 < \cdots < p_k$ primes, $\alpha_j > 0$). This result is easy to prove, but the standard proof gives no hint of an efficient algorithm for computing the prime power decomposition. In order to compute it, we need –

1. An algorithm to test if an integer N is prime.
2. An algorithm to find a nontrivial factor f of a composite integer N.

Given these components there is a simple recursive algorithm to compute the prime power decomposition.

Fortunately or unfortunately, depending on one's point of view, problem 2 is generally believed to be hard. There is no known deterministic or randomised polynomial-time[1] algorithm for finding a factor of a given composite integer N. This empirical fact is of great interest because the most popular algorithm for

[1] For a polynomial-time algorithm the expected running time should be a polynomial in the length of the input, i.e. $O((\log N)^c)$ for some constant c.

public-key cryptography, the RSA algorithm [75], would be insecure if a fast integer factorisation algorithm could be implemented [62].

In this paper we survey some of the most successful integer factorisation algorithms. Since there are already several excellent surveys emphasising the number-theoretic basis of the algorithms, we concentrate on the computational aspects, and particularly on parallel/distributed implementations of the algorithms.

1.1 Primality testing

There are deterministic primality testing algorithms whose worst-case running time on a sequential computer is $O((\log N)^{c \log\log\log N})$, where c is a moderate constant. These algorithms are practical for numbers N of several hundred decimal digits [2, 20, 55]. If we are willing to accept a very small probability of error, then faster (polynomial-time) probabilistic algorithms are available [36, 57, 72]. Thus, in this paper we assume that primality testing is easy and concentrate on the more difficult problem of factoring composite integers.

1.2 Public key cryptography

As we already observed, large primes have a significant practical application – they can be used to construct *public key* cryptosystems[2]. The best-known is the *RSA system*, named after its inventors Rivest, Shamir and Adleman [75]. The security of RSA depends on the (assumed) difficulty of factoring the product of two large primes. This is the main practical motivation for the current interest in integer factorisation algorithms. Of course, mathematicians have been interested in factorisation algorithms for hundreds of years, but until recently it was not known that such algorithms were of "practical" importance.

In the RSA system we usually take $N = p_1 p_2$, where p_1, p_2 are large primes, each approximately equal, but not too close, to $n^{1/2}$. The product N is made public but the factors p_1, p_2 are kept secret. There is an implementation advantage in using a product of three large primes, $N = p_1 p_2 p_3$, where each p_i is approximately $N^{1/3}$. Some of the computations can be done mod p_i and the results (mod N) deduced via the Chinese remainder theorem. This is faster if we use three primes instead of two. On the other hand, the security of the system may be compromised because N, having smaller prime factors, may be easier to factor than in the two-prime case.

1.3 The discrete logarithm problem

The difficulty of the discrete logarithm problem [61, 46] was used by Diffie and Hellman [27] to construct the *Diffie-Hellman key agreement protocol*. This well-known protocol allows two parties to establish a secret key through an exchange

[2] A concept introduced by Diffie and Hellman [27]. Also known as *asymmetric* or *open encryption key* cryptosystems [77, 85].

of public messages. Related public-key algorithms, such as the El Gamal algorithm [30, 31, 77], also depend on the difficulty of the discrete logarithm problem. These public-key algorithms provide practical alternatives to the RSA algorithm. Although originally considered in the setting of the multiplicative group of $\mathbf{GF}(p)$ (the finite field with a prime number p of elements), they generalise to any finite group G. There may be advantages (increased speed or security for a fixed size) in choosing other groups. Neal Koblitz [37] and Victor Miller independently proposed using the group of points on an elliptic curve, and this is a subject of much current research.

We do not consider algorithms for discrete logarithms in this paper. However, it is interesting to note that in some cases integer factorisation algorithms have analogues which apply to the discrete logarithm problem [61, 88, 89]. This does not seem to be the case for discrete logarithms over elliptic curves, which is one reason for the popularity of elliptic curves in cryptographic applications [47, 48].

1.4 Parallel algorithms

When designing parallel algorithms we hope that an algorithm which requires time T_1 on a computer with one processor can be implemented to run in time $T_P \sim T_1/P$ on a computer with P independent processors. This is not always the case, since it may be impossible to use all P processors effectively. However, it is true for many integer factorisation algorithms, provided P is not too large.

The *speedup* of a parallel algorithm is $S = T_1/T_P$. We aim for a linear speedup, i.e. $S = \Theta(P)$.

2 Multiple-Precision Arithmetic

Before describing some integer factorisation algorithms, we comment on the implementation of multiple-precision integer arithmetic on vector processors and parallel machines. Multiple-precision arithmetic is necessary because the number N which we want to factor may be much larger than can be represented in a single computer word (otherwise the problem is trivial).

2.1 Carry propagation and redundant number representations

To represent a large positive integer N, it is customary to choose a convenient *base* or *radix* β and express N as

$$N = \sum_{0}^{t-1} d_j \beta^j,$$

where $d_0, \ldots, d_{t-1}$ are "base β digits" in the range $0 \le d_j < \beta$. We choose β large, but small enough that $\beta - 1$ is representable in a single word [4, 36]. Consider multiple-precision addition (subtraction and multiplication may be handled in a similar way). On a parallel machine there is a problem with *carry propagation* because a carry can propagate all the way from the least to the most significant digit. Thus an addition takes worst case time $\Theta(t)$, and average time $\Theta(\log t)$, independent of the number of processors.

The carry propagation problem can be reduced if we permit digits d_j outside the normal range. Suppose that we allow $-2 \le d_j \le \beta + 1$, where $\beta > 4$. Then possible carries are in $\{-1, 0, 1, 2\}$ and we need only add corresponding pairs of digits (in parallel), compute the carries and perform one step of carry propagation. It is only when comparisons of multiple-precision numbers need to be performed that the digits have to be reduced to the normal range by fully propagating carries. Thus, redundant number representation is useful for speeding up multiple-precision addition and multiplication. On a parallel machine with sufficiently many processors, such a representation allows addition to be performed in constant time.

2.2 High level parallelism

Rather than trying to perform individual multiple-precision operations rapidly, it is often more convenient to implement the multiple-precision operations in bit or word-serial fashion, but perform many independent operations in parallel. For example, a *trial* of the elliptic curve algorithm (§7) involves a predetermined sequence of additions and multiplications on integers of bounded size. Our implementation on a Fujitsu VPP 300 performs many trials concurrently (on one or more processors) in order to take advantage of each processor's vector pipelines.

2.3 Use of real arithmetic

Most supercomputers were not designed to optimise the performance of exact (multiple-precision) integer arithmetic. On machines with fast floating-point hardware, e.g. pipelined 64-bit floating point units, it may be best to represent base β digits in *floating-point* words. The upper bound on β is imposed by the multiplication algorithm – we must ensure that β^2 is exactly representable in a (single or double-precision) floating-point word. In practice it is convenient to allow some slack – for example, we might require $8\beta^2$ to be exactly representable. On machines with IEEE standard arithmetic, we could use $\beta = 2^{24}$.

2.4 Redundant representations mod N

Many integer factorisation algorithms require operations to be performed modulo N, where N is the number to be factored. A straightforward implementation would perform a multiple-precision operation and then perform a division by N to find the remainder. Since N is fixed, some precomputation involving N (e.g. reciprocal approximation) may be worthwhile. However, it may be faster to avoid explicit divisions, by taking advantage of the fact that it is not usually necessary to represent the result uniquely.

For example, consider the computation of $x * y \bmod N$. The result is $r = x * y - q * N$ and it may be sufficient to choose q so that $0 \le r < 2N$

(a weaker constraint than the usual $0 \le r < N$). To compute r we multiply x by the digits of y, most significant digit first, but modify the standard "shift and add" algorithm to subtract single-precision multiples of N in order to keep the accumulated sum bounded by $2N$. Formally, a partial sum s is updated by $s \leftarrow \beta * s + y_j * x - q_j * N$, where q_j is obtained by a division involving only a few leading digits of $\beta * s + y_j * x$ and N.

Alternatively, a technique of Montgomery [49] can be used to speed up modular arithmetic.

2.5 Computing inverses mod N

In some factorisation algorithms we need to compute inverses mod N. Suppose that x is given, $0 < x < N$, and we want to compute z such that $xz = 1 \bmod N$. The extended Euclidean algorithm [36] applied to x and N gives u and v such that

$$ux + vN = GCD(x, N).$$

If $GCD(x, N) = 1$ then $ux = 1 \bmod N$, so $z = u$. If $GCD(x, N) > 1$ then $GCD(x, N)$ is a nontrivial factor of N. This is a case where failure (in finding an inverse) implies success (in finding a factor) !

3 Integer Factorisation Algorithms

There are many algorithms for finding a nontrivial factor f of a composite integer N. The most useful algorithms fall into one of two classes –

A. The run time depends mainly on the size of N, and is not strongly dependent on the size of f. Examples are –
   - Lehman's algorithm [39], which has worst-case run time $O(N^{1/3})$.
   - The Continued Fraction algorithm [56] and the Multiple Polynomial Quadratic Sieve (MPQS) algorithm [67], which under plausible assumptions have expected run time $O(\exp(c(\ln N \ln\ln N)^{1/2}))$, where c is a constant (depending on details of the algorithm). For MPQS, $c \approx 1$.
   - The Number Field Sieve (NFS) algorithm [41, 40], which under plausible assumptions has expected run time $O(\exp(c(\ln N)^{1/3}(\ln\ln N)^{2/3}))$, where c is a constant (depending on details of the algorithm and on the form of N).

B. The run time depends mainly on the size of f, the factor found. (We can assume that $f \le N^{1/2}$.) Examples are –
   - The trial division algorithm, which has run time $O(f \cdot (\log N)^2)$.
   - Pollard's "rho" algorithm [66], which under plausible assumptions has expected run time $O(f^{1/2} \cdot (\log N)^2)$.
   - Lenstra's Elliptic Curve (ECM) algorithm [45], which under plausible assumptions has expected run time $O(\exp(c(\ln f \ln\ln f)^{1/2}) \cdot (\log N)^2)$, where $c \approx 2$ is a constant.

In these examples, the time bounds are for a sequential machine, and the term $(\log N)^2$ is a generous allowance for the cost of performing arithmetic operations on numbers which are $O(N^2)$. If N is very large, then fast integer multiplication algorithms [24, 36] can be used to reduce the $(\log N)^2$ term.

Our survey of integer factorisation algorithms in §§4–10 below is necessarily cursory. For more information the reader is referred to the literature [7, 13, 52, 69, 74].

3.1 Quantum factorisation algorithms

In 1994 Shor [79, 80] showed that it is possible to factor in polynomial expected time on a quantum computer [25, 26]. However, despite the best efforts of several research groups, such a computer has not yet been built, and it remains unclear whether it will ever be feasible to build one. Thus, in this paper we restrict our attention to algorithms which run on classical (serial or parallel) computers [86]. The reader interested in quantum computers could start by reading [71, 87].

4 Pollard's "rho" Algorithm

Pollard's "rho" algorithm [5, 66] uses an iteration of the form

$$x_{i+1} = f(x_i) \bmod N, \quad i \ge 0,$$

where N is the number to be factored, x_0 is a random starting value, and f is a nonlinear polynomial with integer coefficients, for example

$$f(x) = x^2 + a \quad (a \ne 0 \bmod N)\ .$$

Let p be the smallest prime factor of N, and j the smallest positive index such that $x_{2j} = x_j \pmod p$. Making some plausible assumptions, it is easy to show that the expected value of j is $E(j) = O(p^{1/2})$. The argument is related to the well-known "birthday" paradox – the probability that $x_0, x_1, \ldots, x_k$ are all distinct mod p is approximately

$$(1 - 1/p) \cdot (1 - 2/p) \cdots (1 - k/p) \sim \exp\left(\frac{-k^2}{2p}\right)\ ,$$

and if $x_0, x_1, \ldots, x_k$ are not all distinct mod p then $j \le k$.

In practice we do not know p in advance, but we can detect x_j by taking greatest common divisors. We simply compute GCD $(x_{2i} - x_i, N)$ for $i = 1, 2, \ldots$ and stop when a GCD greater than 1 is found.

4.1 Pollard rho examples

An early example of the success of a variation of the Pollard "rho" algorithm is the complete factorisation of the Fermat number $F_8 = 2^{2^8} + 1$ by Brent and Pollard [11]. In fact

$$F_8 = 1238926361552897 \cdot p_{62},$$

where p_{62} is a 62-digit prime.

The *Cunningham project* [12] is a collaborative effort to factor numbers of the form $a^n \pm 1$, where $a \leq 12$. The largest factor found by the Pollard "rho" algorithm during the Cunningham project is a 19-digit factor of $2^{2386}+1$ (found by Harvey Dubner on a Dubner Cruncher [14]). Larger factors could certainly be found, but the discovery of ECM (§7) has made the Pollard "rho" algorithm uncompetitive for factors greater than about 10 decimal digits [6, Table 1].

4.2 Parallel rho

Parallel implementation of the "rho" algorithm does not give linear speedup[3]. A plausible use of parallelism is to try several different pseudo-random sequences (generated by different polynomials f). If we have P processors and use P different sequences in parallel, the probability that the first k values in each sequence are distinct mod p is approximately $\exp(-k^2P/(2p))$, so the speedup is $\Theta(P^{1/2})$. Recently Crandall [23] has suggested that a speedup $\Theta(P/(\log P)^2)$ is possible, but his proposal has not yet been tested.

5 The Advantages of a Group Operation

The Pollard rho algorithm takes $x_{i+1} = f(x_i) \bmod N$ where f is a polynomial. Computing x_n requires n steps. Suppose instead that $x_{i+1} = x_0 \circ x_i$ where "$\circ$" is an associative operator, which for the moment we can think of as multiplication. We can compute x_n in $O(\log n)$ steps by the *binary powering* method [36].

Let m be some bound assigned in advance, and let E be the product of all maximal prime powers q^e, $q^e \leq m$. Choose some starting value x_0, and consider the cyclic group $< x_0 >$ consisting of all powers of x_0 (under the associative operator "$\circ$"). If this group has order g whose prime power components are bounded by m, then $g|E$ and $x_0^E = I$, where I is the group identity.

We may consider a group defined mod p but work mod N, where p is an unknown divisor of N. This amounts to using a redundant representation for the group elements. When we compute the identity I, its representation mod N may allow us to compute p via a GCD computation (compare Pollard's rho algorithm). We give two examples below: Pollard's $p-1$ algorithm and Lenstra's elliptic curve algorithm.

6 Pollard's $p - 1$ Algorithm

Pollard's "$p-1$" algorithm [65] may be regarded as an attempt to generate the identity in the multiplicative group of $\mathbf{GF}(p)$. Here the group operation "$\circ$" is just multiplication mod p, so (by Fermat's theorem) $g|p-1$ and

$$x_0^E = I \Rightarrow x_0^E = 1 \pmod p \Rightarrow p|\mathrm{GCD}\ (x_0^E - 1, N)$$

[3] Variants of the "rho" algorithm can be used to solve the discrete logarithm problem. Recently, van Oorschot and Wiener [63, 64] have shown that a linear speedup is possible in this application.

6.1 $p-1$ example

The largest factor found by the Pollard "$p-1$" algorithm during the Cunningham project is a 32-digit factor

$$p_{32} = 49858990580788843054012690078841$$

of $2^{977}-1$. In this case

$$p_{32} - 1 = 2^3 \cdot 5 \cdot 13 \cdot 19 \cdot 977 \cdot 1231 \cdot 4643 \cdot 74941 \cdot 1045397 \cdot 11535449$$

6.2 Parallel $p-1$

Parallel implementation of the "$p-1$" algorithm is difficult, because the inner loop seems inherently serial. At best, parallelism can speed up the multiple precision operations by a small factor (depending on $\log N$ but not on p).

6.3 The worst case for $p-1$

In the worst case, when $(p-1)/2$ is prime, the "$p-1$" algorithm is no better than trial division. Since the group has fixed order $p-1$ there is nothing to be done except try a different algorithm. In the next section we show that it is possible to overcome the main handicaps of the "$p-1$" algorithm, and obtain an algorithm which is easy to implement in parallel and does not depend on the factorisation of $p-1$.

7 Lenstra's Elliptic Curve Algorithm

If we can choose a "random" group G with order g close to p, we may be able to perform a computation similar to that involved in Pollard's "$p-1$" algorithm, working in G rather than in $\mathbf{GF}(p)$. If all prime factors of g are less than the bound m then we find a factor of N. Otherwise, repeat with a different G (and hence, usually, a different g) until a factor is found. This is the motivation for H. W. Lenstra's *elliptic curve algorithm (or method)* (ECM).

A curve of the form

$$y^2 = x^3 + ax + b \tag{1}$$

over some field F is known as an *elliptic curve*. A more general cubic in x and y can be reduced to the form (1), which is known as the Weierstrass normal form, by rational transformations, provided $\mathrm{char}(F) \neq 2$ or 3.

There is a well-known way of defining an Abelian group $(G, \circ)$ on an elliptic curve over a field. Formally, if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are points on the curve, then the point $P_3 = (x_3, y_3) = P_1 \circ P_2$ is defined by –

$$(x_3, y_3) = (\lambda^2 - x_1 - x_2,\ \lambda(x_1 - x_3) - y_1), \tag{2}$$

where

$$\lambda = \begin{cases} (3x_1^2 + a)/(2y_1) & \text{if } P_1 = P_2 \\ (y_1 - y_2)/(x_1 - x_2) & \text{otherwise.} \end{cases}$$

The identity element in G is the "point at infinity", (∞, ∞).

From now on we write "$\circ$" as "$+$", since this is standard in the elliptic curve literature. Thus (∞, ∞) is the "zero" element of G, and is written as 0.

The geometric interpretation of $P_1 + P_2$ is straightforward: the straight line P_1P_2 intersects the elliptic curve at a third point $P_3' = (x_3, -y_3)$, and P_3 is the reflection of P_3' in the x-axis. We refer the reader to a suitable text [19, 35, 38, 81] for an introduction to the theory of elliptic curves.

In Lenstra's algorithm [45] the field F is the finite field $\mathbf{GF}(p)$ of p elements, where p is a prime factor of N. The multiplicative group of $\mathbf{GF}(p)$, used in Pollard's "$p-1$" algorithm, is replaced by the group G defined by (1–2). Since p is not known in advance, computation is performed in the ring Z/NZ of integers modulo N rather than in $\mathbf{GF}(p)$. We can regard this as using a redundant group representation.

A *trial* is the computation involving one random group G. The steps involved are –

1. Choose x_0, y_0 and a randomly in $[0, N)$. This defines $b = y_0^2 - (x_0^3 + ax_0) \bmod N$. Set $P \leftarrow P_0 = (x_0, y_0)$.
2. For prime $q \le m$ set $P \leftarrow q^e P$ in the group G defined by a and b, where e is an exponent chosen as in §5. If $P = 0$ then the trial succeeds as a factor of N will have been found during an attempt to compute an inverse mod N. Otherwise the trial fails.

The work involved in a trial is $O(m)$ group operations. There is a tradeoff involved in the choice of m, as a trial with large m is expensive, but a trial with small m is unlikely to succeed.

Given $x \in \mathbf{GF}(p)$, there are at most two values of $y \in \mathbf{GF}(p)$ satisfying (1). Thus, allowing for the identity element, we have $g = |G| \le 2p + 1$. A much stronger result, the *Riemann hypothesis for finite fields*, is known –

$$|g - p - 1| < 2p^{1/2} .$$

Making a plausible assumption about the distribution of prime divisors of g, one may show that the optimal choice of m is $m = p^{1/\alpha}$, where

$$\alpha \sim (2 \ln p / \ln \ln p)^{1/2} .$$

It follows that the expected run time is

$$T = p^{2/\alpha + o(1/\alpha)} . \tag{3}$$

For details, see Lenstra [45]. The exponent $2/\alpha$ in (3) should be compared with 1 (for trial division) or 1/2 (for Pollard's "rho" method). Because of the overheads involved with ECM, a simpler algorithm such as Pollard's "rho" is preferable for finding factors of size up to about 10^{10}, but for larger factors the asymptotic advantage of ECM becomes apparent. The following examples illustrate the power of ECM.

7.1 ECM examples

1. In 1995 we completed the factorisation of the 309-decimal digit (1025-bit) Fermat number $F_{10} = 2^{2^{10}} + 1$. In fact

$$F_{10} = 45592577 \cdot 6487031809 \cdot \\ 4659775785220018543264560743076778192897 \cdot p_{252}$$

   where $46597 \cdots 92897$ is a 40-digit prime and $p_{252} = 13043 \cdots 24577$ is a 252-digit prime. The computation, which is described in detail in [10], took about 240 Mips-years.
2. The largest factor known to have been found by ECM is the 53-digit factor

$$53625112691923843508117942311516428173021903300344567$$

   of $2^{677} - 1$, found by Conrad Curry in September 1998 using a program written by George Woltman and running on 16 Pentiums (for more details see [9]). Note that if the RSA system were used with 512-bit keys and the three-prime variation, as described in §1.2, the smallest prime would be less than 53 decimal digits, so ECM could be used to break the system.

7.2 The second phase

Both the Pollard "$p-1$" and Lenstra elliptic curve algorithms can be speeded up by the addition of a second phase. The idea of the second phase is to find a factor in the case that the first phase terminates with a group element $P \neq 0$, such that $|\langle P \rangle|$ is reasonably small (say $O(m^2)$). (Here $\langle P \rangle$ is the cyclic group generated by P.) There are several possible implementations of the second phase. One of the simplest uses a pseudorandom walk in $\langle P \rangle$. By the birthday paradox argument, there is a good chance that two points in the random walk will coincide after $O(|\langle P \rangle|^{1/2})$ steps, and when this occurs a nontrivial factor of N can usually be found. Details of this and other implementations of the second phase may be found in [6, 10, 28, 50, 51, 83].

The use of a second phase provides a significant speedup in practice, but does not change the asymptotic time bound (3). Similar comments apply to other implementation details, such as ways of avoiding most divisions and speeding up group operations, ways of choosing good initial points, and ways of using preconditioned polynomial evaluation [6, 50, 51].

7.3 Parallel/distributed implementation of ECM

Unlike the Pollard "rho" and "$p-1$" methods, ECM is "embarrassingly parallel", because each trial is independent. So long as the expected number of trials is much larger than the number P of processors available, linear speedup is possible by performing P trials in parallel. In fact, if T_1 is the expected run time on one processor, then the expected run time on a MIMD parallel machine with P processors is

$$T_P = T_1/P + O(T_1^{1/2+\epsilon}) \tag{4}$$

The bound (4) applies on a SIMD machine if we use the Montgomery-Chudnovsky form [18, 50]

$$by^2 = x^3 + ax^2 + x$$

instead of the Weierstrass normal form (1) in order to avoid divisions.

In practice, it may be difficult to perform P trials in parallel because of storage limitations. The second phase requires more storage than the first phase. Fortunately, there are several possibilities for making use of parallelism during the second phase of each trial. One parallel implementation performs the first phase of P trials in parallel, but the second phase of each trial sequentially, using P processors to speed up the evaluation of the high-degree polynomials which constitute most of the work during the second phase.

8 Quadratic Sieve Algorithms

Quadratic sieve algorithms belong to a wide class of algorithms which try to find two integers x and y such that $x \neq \pm y \pmod N$ but

$$x^2 = y^2 \pmod N\ . \tag{5}$$

Once such x and y are found, then GCD $(x - y, N)$ is a nontrivial factor of N.

One way to find x and y satisfying (5) is to find a set of *relations* of the form

$$u_i^2 = v_i^2 w_i \pmod N, \tag{6}$$

where the w_i have all their prime factors in a moderately small set of primes (called the *factor base*). Each relation (6) gives a row in a matrix M whose columns correspond to the primes in the factor base. Once enough rows have been generated, we can use sparse Gaussian elimination in $\mathbf{GF}(2)$ to find a linear dependency (mod 2) between a set of rows of M. Multiplying the corresponding relations now gives an expression of the form (5). With probability at least $1/2$, we have $x \neq \pm y \bmod N$ so a nontrivial factor of N will be found. If not, we need to obtain a different linear dependency and try again.

In quadratic sieve algorithms the numbers w_i are the values of one (or more) polynomials with integer coefficients. This makes it easy to factor the w_i by *sieving*. For details of the process, we refer to [15, 43, 67, 70, 73, 82]. The inner loop of the sieving process has the form

while $j < bound$ **do**
  begin
  $s[j] \leftarrow s[j] + c;$
  $j \leftarrow j + q;$
  end

Here *bound* depends on the size of the (single-precision real) sieve array s, q is a small prime or prime power, and c is a single-precision real constant depending on q ($c = \Lambda(q) = \log p$ if $q = p^e$, p prime). The loop can be implemented efficiently on a pipelined vector processor. It is possible to use scaling to avoid floating point additions, which is desirable on a small processor without floating-point hardware.

In order to minimise cache misses on a machine whose memory cache is too small to store the whole array s, it may be desirable to split the inner loop to perform sieving over cache-sized blocks.

The best quadratic sieve algorithm, the *multiple polynomial quadratic sieve algorithm* MPQS [67, 82] can, under plausible assumptions, factor a number N in time $\Theta(\exp(c(\ln N \ln\ln N)^{1/2}))$, where $c \sim 1$. The constants involved are such that MPQS is usually faster than ECM if N is the product of two primes which both exceed $N^{1/3}$. This is because the inner loop of MPQS involves only single-precision operations.

Use of "partial relations", i.e. incompletely factored w_i, in MPQS is analogous to the second phase of ECM and gives a similar performance improvement [3]. In the "one large prime" (P-MPQS) variation w_i is allowed to have one prime factor exceeding m (but not too much larger than m). In the "two large prime" (PP-MPQS) variation w_i can have two prime factors exceeding m – this gives a further performance improvement at the expense of higher storage requirements [44], and does not seem to have an analogue applicable to ECM.

8.1 Parallel/distributed implementation of MPQS

Like ECM, the sieving stage of MPQS is ideally suited to parallel implementation. Different processors may use different polynomials, or sieve over different intervals with the same polynomial. Thus, there is a linear speedup so long as the number of processors is not much larger than the size of the factor base. The computation requires very little communication between processors. Each processor can generate relations and forward them to some central collection point. This was demonstrated by A. K. Lenstra and M. S. Manasse [43], who distributed their program and collected relations via electronic mail. The processors were scattered around the world – anyone with access to electronic mail and a C compiler could volunteer to contribute[4]. The final stage of MPQS – Gaussian elimination to combine the relations – was not so easily distributed. In practice it is only a small fraction of the overall computation, but it may become

[4] This idea of using machines on the Internet as a "free" supercomputer has recently been adopted by several other computation-intensive projects

a limitation if very large numbers are attempted by MPQS (a similar problem is discussed below in connection with NFS).

8.2 MPQS examples

MPQS has been used to obtain many impressive factorisations [12, 73, 82]. Arjen Lenstra and Mark Manasse [43] (with many assistants scattered around the world) have factored several numbers larger than 10^{100}. For example, a recent factorisation was the 116-decimal digit number $(3^{329}+1)/(\text{known small factors})$ into a product of 50-digit and 67-digit primes. The final factorisation is

$$\begin{aligned} 3^{329}+1 = 2^2 \cdot 547 \cdot 16921 \cdot 256057 \cdot 36913801 \cdot 177140839 \cdot 1534179947851 \cdot \\ 24677078822840014266652779036768062918372697435241 \cdot p_{67} \end{aligned}$$

Such factorisations require many years of CPU time, but a real time of only a month or so because of the number of different processors which are working in parallel.

At the time of writing, the largest number factored by MPQS is the 129-digit "RSA Challenge" [75] number RSA129. It was factored in 1994 by Atkins *et al* [1]. It is certainly feasible to factor larger numbers by MPQS, but for numbers of more than about 110 decimal digits GNFS is faster [32–34]. For example, it is estimated in [21] that to factor RSA129 by MPQS required 5000 Mips-years, but to factor the slightly larger number RSA130 by GNFS required only 1000 Mips-years [22].

9 The Special Number Field Sieve (SNFS)

The *number field sieve* (NFS) algorithm was developed from the *special number field sieve* (SNFS), which we describe in this section. The *general number field sieve* (GNFS or simply NFS) is described in §10.

Most of our numerical examples have involved numbers of the form

$$a^e \pm b \,, \tag{7}$$

for small a and b, although the ECM and MPQS factorisation algorithms do not take advantage of this special form.

The *special number field sieve* (SNFS) is a relatively new algorithm which does take advantage of the special form (7). In concept it is similar to the quadratic sieve algorithm, but it works over an algebraic number field defined by a, e and b. We refer the interested reader to Lenstra *et al* [40, 41] for details, and merely give an example to show the power of the algorithm. For an introduction to the relevant concepts of algebraic number theory, see Stewart and Tall [84].

9.1 SNFS examples

Consider the 155-decimal digit number

$$F_9 = N = 2^{2^9} + 1$$

as a candidate for factoring by SNFS. Note that $8N = m^5 + 8$, where $m = 2^{103}$. We may work in the number field $Q(\alpha)$, where α satisfies

$$\alpha^5 + 8 = 0,$$

and in the ring of integers of $Q(\alpha)$. Because

$$m^5 + 8 = 0 \pmod{N},$$

the mapping $\phi : \alpha \mapsto m \bmod N$ is a ring homomorphism from $Z[\alpha]$ to Z/NZ.

The idea is to search for pairs of small coprime integers u and v such that both the algebraic integer $u + \alpha v$ and the (rational) integer $u + mv$ can be factored. (The factor base now includes prime ideals and units as well as rational primes.) Because

$$\phi(u + \alpha v) = (u + mv) \pmod{N},$$

each such pair gives a relation analogous to (6).

The prime ideal factorisation of $u + \alpha v$ can be obtained from the factorisation of the *norm* $u^5 - 8v^5$ of $u + \alpha v$. Thus, we have to factor simultaneously two integers $u + mv$ and $|u^5 - 8v^5|$. Note that, for moderate u and v, both these integers are much smaller than N, in fact they are $O(N^{1/d})$, where $d = 5$ is the degree of the algebraic number field. (The optimal choice of d is discussed below.)

Using these and related ideas, Lenstra *et al* [42] factored F_9 in June 1990, obtaining

$$F_9 = 2424833 \cdot 7455602825647884208337395736200454918783366342657 \cdot p_{99},$$

where p_{99} is an 99-digit prime, and the 7-digit factor was already known (although SNFS was unable to take advantage of this). The collection of relations took less than two months on a network of several hundred workstations. A sparse system of about 200,000 relations was reduced to a dense matrix with about 72,000 rows. Using Gaussian elimination, dependencies (mod 2) between the rows were found in three hours on a Connection Machine. These dependencies implied equations of the form $x^2 = y^2 \bmod F_9$. The second such equation was nontrivial and gave the desired factorisation of F_9.

More recently, considerably larger numbers have been factored by SNFS. The current record is the 211-digit number $10^{211} - 1$, factored early in 1999 by a collaboration called "The Cabal" [17]. In fact, $10^{211} - 1 = 3^2 \cdot p_{93} \cdot p_{118}$, where

$$\begin{aligned} p_{93} = {} & 6926245573243896206627823226773367111381084825 \\ & 88281739734375570506492391931849524636731866879 \end{aligned}$$

and p_{118} may be found by division. The factorisation of $N = 10^{211} - 1$ used two polynomials

$$f(x) = x - 10^{35}$$

and

$$g(x) = 10x^6 - 1$$

with common root $m = 10^{35} \bmod N$. Details of the computation can be found in [17]. To summarise: after sieving and reduction a sparse matrix over $\mathbf{GF}(2)$ was obtained with about 4.8×10^6 rows and weight (number of nonzero entries) about 2.3×10^8, an average of about 49 nonzeros per row. Montgomery's block Lanczos program (see §10) took 121 hours on a Cray C90 to find 64 dependencies. Finally, the square root program needed 15.5 hours on one CPU of an SGI Origin 2000, and three dependencies to find the two prime factors.

10 The General Number Field Sieve (GNFS)

The *general number field sieve* (GNFS or just NFS) is a logical extension of the special number field sieve (SNFS). When using SNFS to factor an integer N, we require two polynomials $f(x)$ and $g(x)$ with a common root $m \bmod N$ but no common root over the field of complex numbers. If N has the special form (7) then it is usually easy to write down suitable polynomials with small coefficients, as illustrated by the two examples given in §9. If N has no special form, but is just some given composite number, we can also find $f(x)$ and $g(x)$, but they no longer have small coefficients.

Suppose that $g(x)$ has degree $d > 1$ and $f(x)$ is linear[5]. d is chosen empirically, but it is known from theoretical considerations that the optimum value is

$$d \sim \left(\frac{3 \ln N}{\ln \ln N} \right)^{1/3} .$$

We choose $m = \lfloor N^{1/d} \rfloor$ and write

$$N = \sum_{j=0}^{d} a_j m^j$$

where the a_j are "base m digits" and $a_d = 1$. Then, defining

$$f(x) = x - m, \quad g(x) = \sum_{j=0}^{d} a_j x^j ,$$

it is clear that $f(x)$ and $g(x)$ have a common root $m \bmod N$. This method of polynomial selection is called the "base m" method.

[5] This is not necessary. For example, Montgomery found a clever way (described in [32]) of choosing two quadratic polynomials.

In principle, we can proceed as in SNFS, but many difficulties arise because of the large coefficients of $g(x)$. For details, we refer the reader to [32, 33, 53, 54, 59, 68, 69, 91]. Suffice it to say that the difficulties can be overcome and the method works! Due to the constant factors involved it is slower than MPQS for numbers of less than about 110 decimal digits, but faster than MPQS for sufficiently large numbers, as anticipated from the theoretical run times given in §3.

Some of the difficulties which had to be overcome to turn GNFS into a practical algorithm are:

1. Polynomial selection. The "base m" method is not very good. Brian Murphy [58–60] has shown how a very considerable improvement (by a factor of more than ten for number of 140 digits) can be obtained.
2. Linear algebra. After sieving a very large, sparse linear system over $\mathbf{GF}(2)$ is obtained, and we want to find dependencies amongst the rows. It is not practical to do this by Gaussian elimination [90] because the "fill in" is too large. Montgomery [54] showed that the Lanczos method could be adapted for this purpose. (This is nontrivial because a nonzero vector x over $\mathbf{GF}(2)$ can be orthogonal to itself, i.e. $x^T x = 0$.) To take advantage of bit-parallel operations, Montgomery's program works with blocks of size dependent on the wordlength (e.g. 64).
3. Square roots. The final stage of GNFS involves finding the square root of a (very large) product of algebraic numbers[6]. Once again, Montgomery [53] found a way to do this.

At present, the main obstacle to a fully parallel implementation of GNFS is the linear algebra. Montgomery's block Lanczos program runs on a single processor and requires enough memory to store the sparse matrix. In principle it should be possible to distribute the block Lanczos solution over several processors of a parallel machine, but the communication/computation ratio will be high.

It should be noted that if special hardware is built for sieving, as pioneered by Lehmer and recently proposed (in more modern form) by Shamir [78], the linear algebra will become relatively more important[7].

10.1 RSA140

At the time of writing, the largest number factored by GNFS is the 140-digit RSA Challenge number RSA140. It was split into the product of two 70-digit primes in February, 1999, by a team coordinated from CWI, Amsterdam. For details see [16]. To summarise: the amount of computer time required to find the factors was about 2000 Mips-years. The two polynomials used were

[6] An idea of Adleman, using quadratic characters, is essential to ensure that the square root exists.

[7] The argument is similar to Amdahl's law: no matter how fast sieving is done, we can not avoid the linear algebra.

$$f(x) = x - 34435657809242536951779007$$

and

$$\begin{aligned} g(x) = &+439682082840x^5 + 390315678538960x^4 \\ &-7387325293892994572x^3 - 19027153243742988714824x^2 \\ &-63441025694464617913930613x + 318553917071474350392223507494 \,. \end{aligned}$$

The polynomial $g(x)$ was chosen to have a good combination of two properties: being unusually small over the sieving region and having unusually many roots modulo small primes (and prime powers). The effect of the second property alone makes $g(x)$ as effective at generating relations as a polynomial chosen at random for an integer of 121 decimal digits (so in effect we have removed at least 19 digits from RSA140 by judicious polynomial selection). The polynomial selection took 2000 CPU-hours on four 250 MHz SGI Origin 2000 processors. Sieving was done on about 125 SGI and Sun workstations running at 175 MHz on average, and on about 60 PCs running at 300 MHz on average. The total amount of CPU time spent on sieving was 8.9 CPU-years.

The resulting matrix had about 4.7×10^6 rows and weight about 1.5×10^8 (about 32 nonzeros per row). Using Montgomery's block Lanczos program, it took almost 100 CPU-hours and 810 MB of memory on a Cray C916 to find 64 dependencies among the rows of this matrix. Calendar time for this was five days.

10.2 RSA155

At the time of writing, an attempt to factor the 512-bit number RSA155 is well underway. We confidently predict that it will be factored before the year 2000!

11 Conclusion

We have sketched some algorithms for integer factorisation. The most important are ECM, MPQS and NFS. The algorithms draw on results in elementary number theory, algebraic number theory and probability theory. As well as their inherent interest and applicability to other areas of mathematics, advances in public key cryptography have lent them practical importance.

Despite much progress in the development of efficient algorithms, our knowledge of the complexity of factorisation is inadequate. We would like to find a polynomial time factorisation algorithm or else prove that one does not exist. Until a polynomial time algorithm is found or a quantum computer capable of running Shor's algorithm [79, 80] is built, large factorisations will remain an interesting challenge.

A survey similar to this one was written in 1990 (see [8]). Comparing the examples there we see that significant progress has been made. This is partly due to Moore's law, partly due to the use of many machines on the Internet, and partly due to improvements in algorithms (especially GNFS). The largest number factored by MPQS at the time of writing [8] had 111 decimal digits. According to [21], the 110-digit number RSA110 was factored in 1992, and the 120-digit number RSA120 was factored in 1993 (both by MPQS). In 1996 GNFS was used to factor RSA130, and in February 1999 GNFS also cracked RSA140. We have predicted that the 512-bit number RSA155 will be factored before the year 2000. Progress seems to be accelerating, but this is due in large part to algorithmic improvements which are unlikely to be repeated.

512-bit RSA keys are now definitely insecure. 1024-bit keys should remain secure for many years, barring the unexpected (but unpredictable) discovery of a completely new algorithm which is better than GNFS, or the development of a practical quantum computer.

Acknowledgements

Thanks are due to Peter Montgomery, Brian Murphy, Herman te Riele, Sam Wagstaff, Jr. and Paul Zimmermann for their assistance.

References

1. D. Atkins, M. Graff, A. K. Lenstra and P. C. Leyland, The magic words are squeamish ossifrage, *Advances in Cryptology: Proc. Asiacrypt'94*, *LNCS* **917**, Springer-Verlag, Berlin, 1995, 263–277.
2. A. O. L. Atkin and F. Morain, Elliptic curves and primality proving, *Math. Comp.* **61** (1993), 29–68. Programs available from `ftp://ftp.inria.fr/INRIA /ecpp.V3.4.1.tar.Z` .
3. H. Boender and H. J. J. te Riele, *Factoring integers with large prime variations of the quadratic sieve,* Experimental Mathematics, **5** (1996), 257–273.
4. R. P. Brent, A Fortran multiple-precision arithmetic package, *ACM Transactions on Mathematical Software* **4** (1978), 57–70.
5. R. P. Brent, An improved Monte Carlo factorisation algorithm, *BIT* **20** (1980), 176–184.
6. R. P. Brent, Some integer factorisation algorithms using elliptic curves, *Australian Computer Science Communications* **8** (1986), 149–163. `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb102.dvi.gz` .
7. R. P. Brent, Parallel algorithms for integer factorisation, in *Number Theory and Cryptography* (edited by J. H. Loxton), London Mathematical Society Lecture Note Series **154**, Cambridge University Press, 1990, 26–37.
8. R. P. Brent, Vector and parallel algorithms for integer factorisation, *Proceedings Third Australian Supercomputer Conference* University of Melbourne, December 1990, 12 *pp.* `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb122.dvi.gz` .
9. R. P. Brent, *Large factors found by ECM,* Oxford University Computing Laboratory, May 1999. `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/champs.txt` .

10. R. P. Brent, Factorization of the tenth Fermat number, *Math. Comp.* **68** (1999), 429-451. Preliminary version available as *Factorization of the tenth and eleventh Fermat numbers,* Technical Report TR-CS-96-02, CSL, ANU, Feb. 1996, 25*pp.* `ftp://ftp.comlab.ox.ac.uk:/pub/Documents/techpapers/Richard.Brent/rpb161tr.dvi.gz` .
11. R. P. Brent and J. M. Pollard, Factorisation of the eighth Fermat number, *Math. Comp.* **36** (1981), 627–630.
12. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman and S. S. Wagstaff, Jr., *Factorisations of* $b^n \pm 1, b = 2, 3, 5, 6, 7, 10, 11, 12$ *up to high powers*, American Mathematical Society, Providence, Rhode Island, second edition, 1988. Updates available from `http://www/cs/purdue.edu/homes/ssw/cun/index.html` .
13. D. A. Buell, Factoring: algorithms, computations, and computers, *J. Supercomputing* **1** (1987), 191–216.
14. C. Caldwell, The Dubner PC Cruncher – a microcomputer coprocessor card for doing integer arithmetic, review in *J. Rec. Math.* **25** (1), 1993.
15. T. R. Caron and R. D. Silverman, Parallel implementation of the quadratic sieve, *J. Supercomputing* **1** (1988), 273–290.
16. S. Cavallar, B. Dodson, A. K. Lenstra, P. Leyland, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele and P. Zimmermann, *Factorization of RSA-140 using the number field sieve*, announced 4 February 1999. Available from `ftp://ftp.cwi.nl/pub/herman/NFSrecords/RSA-140` .
17. S. Cavallar, B. Dodson, A. K. Lenstra, P. Leyland, W. Lioen, P. L. Montgomery, H. te Riele and P. Zimmermann, *211-digit SNFS factorization*, announced 25 April 1999. Available from `ftp://ftp.cwi.nl/pub/herman/NFSrecords/SNFS-211` .
18. D. V. and G. V. Chudnovsky, Sequences of numbers generated by addition in formal groups and new primality and factorization tests, *Adv. in Appl. Math.* **7** (1986), 385–434.
19. H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, Berlin, 1993.
20. H. Cohen and H. W. Lenstra, Jr., Primality testing and Jacobi sums, *Math. Comp.* **42** (1984), 297–330.
21. S. Contini, The factorization of RSA-140, RSA Laboratories Bulletin **10**, 8 (March 1999). Available from `http://www.rsa/com/rsalabs/html/bulletins.html` .
22. J. Cowie, B. Dodson, R. M. Elkenbracht-Huizing, A. K. Lenstra, P. L. Montgomery and J. Zayer, A world wide number field sieve factoring record: on to 512 bits, *Advances in Cryptology: Proc. Asiacrypt'96*, *LNCS* **1163**, Springer-Verlag, Berlin, 1996, 382–394.
23. R. E. Crandall, *Parallelization of Pollard-rho factorization*, preprint, 23 April 1999.
24. R. Crandall and B. Fagin, Discrete weighted transforms and large-integer arithmetic, *Math. Comp.* **62** (1994), 305–324.
25. D. Deutsch, Quantum theory, the Church-Turing principle and the universal quantum computer, *Proc. Roy. Soc. London, Ser. A* **400** (1985), 97–117.
26. D. Deutsch, Quantum computational networks, *Proc. Roy. Soc. London, Ser. A* **425** (1989), 73–90.
27. W. Diffie and M. Hellman, New directions in cryptography, *IEEE Trans. Inform. Theory* **22** (1976), 472–492.
28. B. Dixon and A. K. Lenstra, Massively parallel elliptic curve factoring, *Proc. Eurocrypt '92*, *LNCS* **658**, Springer-Verlag, Berlin, 1993, 183–193.
29. B. Dodson and A. K. Lenstra, NFS with four large primes: an explosive experiment, *Proc. Crypto'95*, *LNCS* **963**, Springer-Verlag, Berlin, 1995, 372–385.

30. T. El Gamal, A public-key cryptosystem and a signature scheme based on discrete logarithms, *Advances in Cryptology: Proc. CRYPTO'84*, Springer-Verlag, Berlin, 1985, 10–18.
31. T. El Gamal, A public-key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. on Information Theory* **31** (1985), 469–472.
32. M. Elkenbracht-Huizing, An implementation of the number field sieve, *Experimental Mathematics*, **5** (1996), 231–253.
33. M. Elkenbracht-Huizing, *Factoring integers with the number field sieve*, Doctor's thesis, Leiden University, 1997.
34. M. Elkenbracht-Huizing, A multiple polynomial general number field sieve *Algorithmic Number Theory – ANTS III, LNCS* **1443**, Springer-Verlag, Berlin, 1998, 99–114.
35. K. F. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag, Berlin, 1982.
36. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison Wesley, third edition, 1997.
37. N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, New York, 1994.
38. S. Lang, *Elliptic Curves – Diophantine Analysis*, Springer-Verlag, Berlin, 1978.
39. R. S. Lehman, Factoring large integers, *Math. Comp.* **28** (1974), 637–646.
40. A. K. Lenstra and H. W. Lenstra, Jr. (editors), The development of the number field sieve, *Lecture Notes in Mathematics* **1554**, Springer-Verlag, Berlin, 1993.
41. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard, *The number field sieve*, Proc. 22nd Annual ACM Conference on Theory of Computing, Baltimore, Maryland, May 1990, 564–572.
42. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, The factorization of the ninth Fermat number, *Math. Comp.* **61** (1993), 319–349.
43. A. K. Lenstra and M. S. Manasse, Factoring by electronic mail, *Proc. Eurocrypt '89, LNCS* **434**, Springer-Verlag, Berlin, 1990, 355–371.
44. A. K. Lenstra and M. S. Manasse, Factoring with two large primes, *Math. Comp.* **63** (1994), 785–798.
45. H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Annals of Mathematics (2)* **126** (1987), 649–673.
46. K. S. McCurley, The discrete logarithm problem, in *Cryptography and Computational Number Theory*, C. Pomerance, ed., *Proc. Symp. Appl. Math.*, Amer. Math. Soc., 1990.
47. A. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, Boston, 1993.
48. A. Menezes, Elliptic curve cryptosystems, *CryptoBytes* **1**, 2 (1995), 1–4. Available from `http://www.rsa.com/rsalabs/pubs/cryptobytes` .
49. P. L. Montgomery, Modular multiplication without trial division, *Math. Comp.* **44** (1985), 519–521.
50. P. L. Montgomery, Speeding the Pollard and elliptic curve methods of factorisation, *Math. Comp.* **48** (1987), 243–264.
51. P. L. Montgomery, *An FFT extension of the elliptic curve method of factorization,* Ph. D. dissertation, Mathematics, University of California at Los Angeles, 1992. `ftp://ftp.cwi.nl/pub/pmontgom/ucladissertation.psl.Z` .
52. P. L. Montgomery, A survey of modern integer factorization algorithms, *CWI Quarterly* **7** (1994), 337–366. `ftp://ftp.cwi.nl/pub/pmontgom/cwisurvey.psl.Z` .
53. P. L. Montgomery, Square roots of products of algebraic numbers, *Mathematics of Computation 1943 – 1993, Proc. Symp. Appl. Math.* **48** (1994), 567–571.

54. P. L. Montgomery, A block Lanczos algorithm for finding dependencies over $GF(2)$, *Advances in Cryptology: Proc. Eurocrypt'95*, *LNCS* **921**, Springer-Verlag, Berlin, 1995, 106–120.
55. F. Morain, *Courbes elliptiques et tests de primalité,* Ph. D. thesis, Univ. Claude Bernard – Lyon I, France, 1990. `ftp://ftp.inria.fr/INRIA/publication/Theses/TU-0144.tar.Z` .
56. M. A. Morrison and J. Brillhart, A method of factorisation and the factorisation of F_7, *Math. Comp.* **29** (1975), 183–205.
57. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
58. B. A. Murphy, Modelling the yield of number field sieve polynomials, *Algorithmic Number Theory – ANTS III*, *LNCS* **1443**, Springer-Verlag, Berlin, 1998, 137–150.
59. B. A. Murphy, *Polynomial selection for the number field sieve integer factorisation algorithm*, Ph. D. thesis, Australian National University, 1999.
60. B. A. Murphy and R. P. Brent, On quadratic polynomials for the number field sieve, *Australian Computer Science Communications* **20** (1998), 199–213.
61. A. M. Odlyzko, Discrete logarithms in finite fields and their cryptographic significance, *Advances in Cryptology: Proc. Eurocrypt '84*, *LNCS* **209**, Springer-Verlag, Berlin, 1985, 224–314.
62. A. M. Odlyzko, The future of integer factorization, *CryptoBytes* **1**, 2 (1995), 5–12. Available from `http://www.rsa.com/rsalabs/pubs/cryptobytes` .
63. P. C. van Oorschot and M. J. Wiener, Parallel collision search with application to hash functions and discrete logarithms, *Proc 2nd ACM Conference on Computer and Communications Security*, ACM, New York, 1994, 210–218.
64. P. C. van Oorschot and M. J. Wiener, Parallel collision search with cryptanalytic applications, *J. Cryptology* **12** (1999), 1–28.
65. J. M. Pollard, Theorems in factorisation and primality testing, *Proc. Cambridge Philos. Soc.* **76** (1974), 521–528.
66. J. M. Pollard, A Monte Carlo method for factorisation, *BIT* **15** (1975), 331–334.
67. C. Pomerance, The quadratic sieve factoring algorithm, *Advances in Cryptology, Proc. Eurocrypt '84, LNCS* **209**, Springer-Verlag, Berlin, 1985, 169–182.
68. C. Pomerance, The number field sieve, *Proceedings of Symposia in Applied Mathematics* **48**, Amer. Math. Soc., Providence, Rhode Island, 1994, 465–480.
69. C. Pomerance, A tale of two sieves, *Notices Amer. Math. Soc.* **43** (1996), 1473–1485.
70. C. Pomerance, J. W. Smith and R. Tuler, A pipeline architecture for factoring large integers with the quadratic sieve algorithm, *SIAM J. on Computing* **17** (1988), 387–403.
71. J. Preskill, *Lecture Notes for Physics 229: Quantum Information and Computation*, California Institute of Technology, Los Angeles, Sept. 1998. `http://www.theory.caltech.edu/people/preskill/ph229/` .
72. M. O. Rabin, Probabilistic algorithms for testing primality, *J. Number Theory* **12** (1980), 128–138.
73. H. J. J. te Riele, W. Lioen and D. Winter, Factoring with the quadratic sieve on large vector computers, *Belgian J. Comp. Appl. Math.* **27** (1989), 267–278.
74. H. Riesel, *Prime numbers and computer methods for factorization,* 2nd edition, Birkhäuser, Boston, 1994.
75. R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM* **21** (1978), 120–126.
76. RSA Laboratories, Information on the RSA challenge, `http://www.rsa/com/rsalabs/html/challenges.html` .

77. B. Schneier, *Applied Cryptography*, second edition, John Wiley and Sons, 1996.
78. A. Shamir, *Factoring large numbers with the TWINKLE device* (extended abstract), preprint, 1999. Announced at Eurocrypt'99.
79. P. W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, *Proc. 35th Annual Symposium on Foundations of Computer Science,* IEEE Computer Society Press, Los Alamitos, California, 1994, 124–134. CMP 98:06
80. P. W. Shor, Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Computing* **26** (1997), 1484–1509.
81. J. H. Silverman, *The arithmetic of elliptic curves,* Graduate Texts in Mathematics **106**, Springer-Verlag, New York, 1986.
82. R. D. Silverman, The multiple polynomial quadratic sieve, *Math. Comp.* **48** (1987), 329–339.
83. R. D. Silverman and S. S. Wagstaff, Jr., A practical analysis of the elliptic curve factoring algorithm, *Math. Comp.* **61** (1993), 445–462.
84. I. N. Stewart and D. O. Tall, *Algebraic Number Theory*, second edition, Chapman and Hall, 1987.
85. D. Stinson, *Cryptography – Theory and Practice*, CRC Press, Boca Raton, 1995.
86. A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.* (2) **42** (1936), 230–265. Errata *ibid* **43** (1937), 544–546.
87. U. Vazirani, Introduction to special section on quantum computation, *SIAM J. Computing* **26** (1997), 1409–1410.
88. D. Weber, Computing discrete logarithms with the number field sieve, *Algorithmic Number Theory – ANTS II*, *LNCS* **1122**, Springer-Verlag, Berlin, 1996, 99–114.
89. D. Weber, *On the computation of discrete logarithms in finite prime fields*, Ph. D. thesis, Universität des Saarlandes, 1997.
90. D. H. Wiedemann, Solving sparse linear equations over finite fields, *IEEE Trans. Inform. Theory* **32** (1986), 54–62.
91. J. Zayer, *Faktorisieren mit dem Number Field Sieve*, Ph. D. thesis, Universität des Saarlandes, 1995.

Postscript (7 September 1999)

This paper (rpb193sp) appeared in *LNCS* **1685** (1999), 1–22. We have corrected a small number of typographical errors. A revised and expanded version (rpb193) is available from the author's web site.