Recent Progress and Prospects for Integer Factorisation Algorithms

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/

26 April 2000

Abstract. The integer factorisation and discrete logarithm problems are of practical importance because of the widespread use of public key cryptosystems whose security depends on the presumed difficulty of solving these problems. This paper considers primarily the integer 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 outline several integer factorisation algorithms, consider their suitability for implementation on parallel machines, and give examples of their current capabilities. In particular, we consider the problem of parallel solution of the large, sparse linear systems which arise with the MPQS and NFS methods.

1 Introduction

There is no known deterministic or randomised polynomial-time algorithm for finding a factor of a given composite integer N. This empirical fact is of great interest because the most popular algorithm for public-key cryptography, the RSA algorithm [54], would be insecure if a fast integer factorisation algorithm could be implemented.

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.

This paper can be regarded as an update of [8], which was written just before the factorisation of the 512-bit number RSA155. Thus, to avoid duplication, we refer to [8] for primality testing, multiple-precision arithmetic, the use of factorisation and discrete logarithms in public-key cryptography, and some factorisation algorithms of historical interest.

1.1 Randomised algorithms

All but the most trivial algorithms discussed below are *randomised algorithms*, i.e. at some point they involve pseudo-random choices. Thus, the running times are *expected* rather than *worst case*. Also, in most cases the expected running times are conjectured, not rigorously proved.

1.2 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. $S = \Theta(P)$.

1.3 Quantum algorithms

In 1994 Shor [57, 58] showed that it is possible to factor in polynomial expected time[1] on a quantum computer [20, 21]. 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. The reader interested in quantum computers could start by reading [50, 60].

1.4 Moore's law

Moore's "law" [44, 56] predicts that circuit densities will double every 18 months or so. Of course, Moore's law is not a theorem, and must eventually fail, but it has been surprisingly accurate for many years. As long as Moore's law continues to apply and results in correspondingly more powerful parallel computers, we expect to get a steady improvement in the capabilities of factorisation algorithms, without any algorithmic improvements. Historically, improvements over the past thirty years have been due both to Moore's law and to the development of new algorithms (e.g. ECM, MPQS, NFS).

[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.

2 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 [28], which has worst-case run time $O(N^{1/3})$.
- The Continued Fraction algorithm [39] and the Multiple Polynomial Quadratic Sieve (MPQS) algorithm [46, 59], which under plausible assumptions have expected run time $O(\exp(\sqrt{c \ln N \ln\ln N}))$, where c is a constant (depending on details of the algorithm). For MPQS, $c \approx 1$.
- The Number Field Sieve (NFS) algorithm [29, 30], 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 [45], which under plausible assumptions has expected run time $O(f^{1/2} \cdot (\log N)^2)$.
- Lenstra's Elliptic Curve (ECM) algorithm [34], which under plausible assumptions has expected run time $O(\exp(\sqrt{c \ln f \ln\ln f}) \cdot (\log N)^2)$, 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 [19, 24] can be used to reduce the $(\log N)^2$ term.

Our survey of integer factorisation algorithms is necessarily cursory. For more information the reader is referred to [8, 35, 48, 53].

3 Lenstra's Elliptic Curve Algorithm

Lenstra's *elliptic curve* method/algorithm (abbreviated ECM) was discovered by H. W. Lenstra, Jr. about 1985 (see [34]). It is the best known algorithm in class B. To save space, we refer to [7, 8, 34] and the references there for a description of ECM, and merely give some examples of its successes here.

3.1 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

$$
\begin{aligned}
F_{10} = {} & 45592577 \cdot 6487031809 \cdot \\
& 4659775785220018543264560743076778192897 \cdot p_{252}
\end{aligned}
$$

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 [7], took about 240 Mips-years. (A Mips-year is the computation performed by a hypothetical machine performing 10^6 instructions per second for one year, i.e. about 3.15×10^{13} instructions. It is a convenient measure but should not be taken too seriously.)

2. The largest factor known to have been found by ECM is the 54-digit factor

$$484061254276878368125726870789180231995964870094916937$$

of $(6^{43} - 1)^{42} + 1$, found by Nik Lygeros and Michel Mizony with Paul Zimmermann's GMP-ECM program [63] in December 1999 (for more details see [9]).

3.2 Parallel/distributed implementation of ECM

ECM consists of a number of independent pseudo-random trials, each of which can be performed on a separate processor. 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

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

3.3 ECM factoring records

Figure 1 shows the size D (in decimal digits) of the largest factor found by ECM against the year it was done, from 1991 (40D) to 1999 (54D) (historical data from [9]).

3.4 Extrapolation of ECM records

Let D be the number of decimal digits in the largest factor found by ECM up to a given date. From the theoretical time bound for ECM, assuming Moore's law, we expect $\sqrt{D}$ to be roughly a linear function of calendar year (in fact $\sqrt{D \ln D}$ should be linear, but given the other uncertainties we have assumed for simplicity that $\sqrt{\ln D}$ is roughly a constant). Figure 2 shows $\sqrt{D}$ versus year Y.

The straight line shown in the Figure 2 is

$$\sqrt{D} = \frac{Y - 1932.3}{9.3} \quad \text{or equivalently} \quad Y = 9.3\sqrt{D} + 1932.3\ ,$$

and extrapolation gives $D = 60$ in the year $Y = 2004$ and $D = 70$ in the year $Y = 2010$.

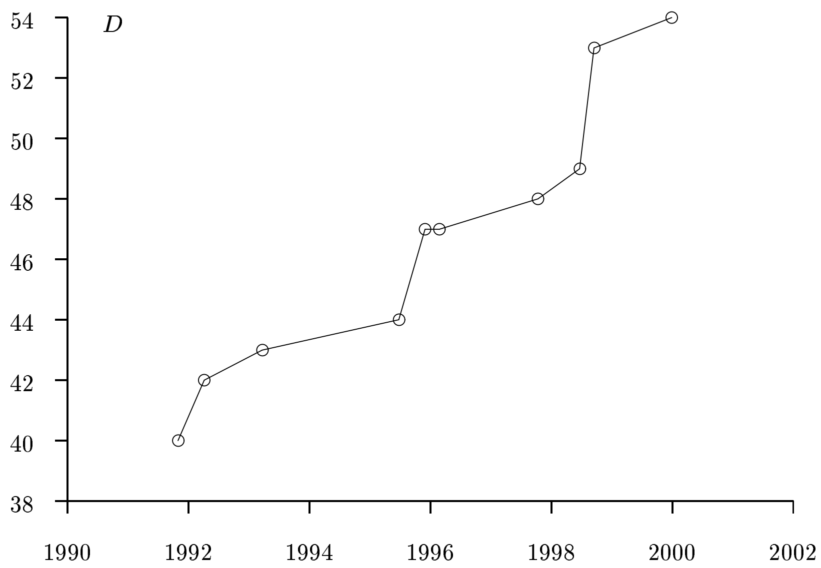

Figure 1: Size of factor found by ECM versus year

4 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

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

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 (2) is to find a set of *relations* of the form

$$u_i^2 = v_i^2 w_i \pmod N, \tag{3}$$

where the w_i have all their prime factors in a moderately small set of primes (called the *factor base*). Each relation (3) gives a colunm in a matrix R whose rows correspond to the primes in the factor base. Once enough columns have been generated, we can use Gaussian elimination in $\mathbf{GF}(2)$ to find a linear dependency (mod 2) between a set of columns of R. Multiplying the corresponding relations now gives an expression of the form (2). 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) quadratic polynomials with integer coefficients. This makes it easy to factor the w_i by *sieving*. For details of the process, we refer to [11, 32, 35, 46, 49, 52, 59].

The best quadratic sieve algorithm (MPQS) 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

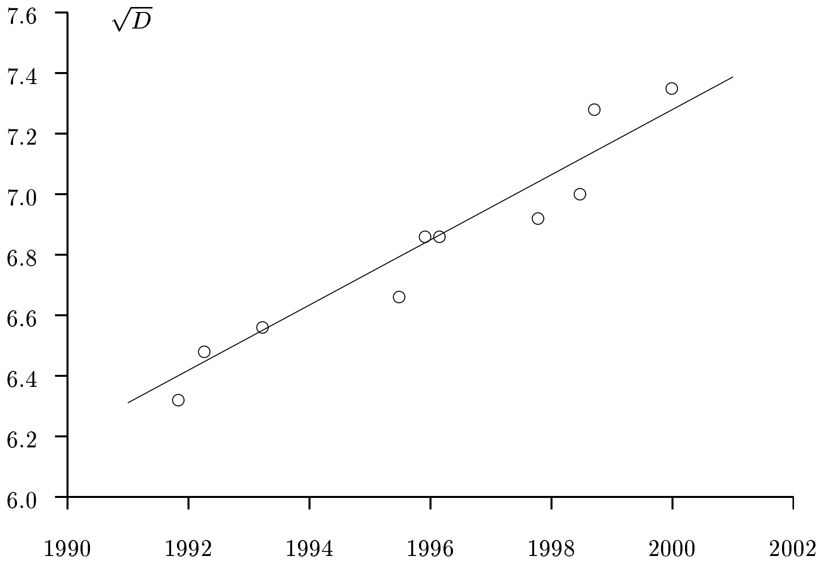

Figure 2: $\sqrt{D}$ versus year Y for ECM

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 gives a significant performance improvement [2]. 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 [33].

4.1 Parallel/distributed implementation of MPQS

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 [32], 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[2]. The final stage of MPQS – Gaussian elimination to combine the relations – is not so easily distributed. We discuss this in §7.

4.2 MPQS examples

MPQS has been used to obtain many impressive factorisations [10, 32, 52, 59]. At the time of writing (April 2000), the largest number factored by MPQS is the 129-digit "RSA Challenge" [55] number RSA129. It was factored in 1994 by Atkins *et al* [1]. The relations formed a sparse matrix with 569466 columns, which was reduced to a dense matrix with 188614 columns; a dependency was then found on a MasPar MP-1. It is certainly feasible to factor larger numbers by MPQS, but for numbers of more than about 110 decimal digits GNFS is faster [22]. For example, it is estimated in [16] 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 [18].

5 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 §6.

Most of our numerical examples involve numbers of the form

$$a^e \pm b \,, \tag{4}$$

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 (4). 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* [29, 30] for details, and merely give an example to show the power of the algorithm.

5.1 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

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

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

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 (3).

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 in §6.)

Using these and related ideas, Lenstra *et al* [31] 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 columns 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, for example, the 211-digit number $10^{211} - 1$ was factored early in 1999 by a collaboration called "The Cabal" [13].

6 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 (4) then it is usually easy to write down suitable polynomials with small coefficients. 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[3]. 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 $m = \lfloor N^{1/(d+1)} \rfloor$ and write

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

where the a_j are "base m digits". 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.

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 [36, 37, 41, 47, 48, 62]. 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 §2.

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. Peter Montgomery and Brian Murphy [40–42] have shown how a very considerable improvement (by a factor of more than ten for number of 140–155 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 columns. It is not practical to do this by structured Gaussian elimination [25, §5] because the "fill in" is too large. Odlyzko [43, 17] and Montgomery [37] showed that the Lanczos method [26] 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[4]. Once again, Montgomery [36] found a way to do this.

[3] This is not necessary. For example, Montgomery found a clever way of choosing two quadratic polynomials.

[4] An idea of Adleman, using quadratic characters, is essential to ensure that the desired square root exists with high probability.

6.1 RSA155

At the time of writing, the largest number factored by GNFS is the 155-digit RSA Challenge number RSA155. It was split into the product of two 78-digit primes on 22 August, 1999, by a team coordinated from CWI, Amsterdam. For details see [51]. To summarise: the amount of computer time required to find the factors was about 8000 Mips-years. The two polynomials used were

$$f(x) = x - 39123079721168000771313449081$$

and

$$\begin{aligned} g(x) = &+119377138320x^5 \\ &-80168937284997582x^4 \\ &-66269852234118574445x^3 \\ &+11816848430079521880356852x^2 \\ &+7459661580071786443919743056x \\ &-40679843542362159361913708405064\,. \end{aligned}$$

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 137 decimal digits (so in effect we have removed at least 18 digits from RSA155 by judicious polynomial selection). The polynomial selection took approximately 100 Mips-years or about 1.25% of the total factorisation time. For details of the polynomial selection, see Brian Murphy's thesis [41].

The total amount of CPU time spent on sieving was 8000 Mips-years on assorted machines (calendar time 3.7 months). The resulting matrix had about 6.7×10^6 rows and weight (number of nonzeros) about 4.2×10^8 (about 62 nonzeros per row). Using Montgomery's block Lanczos program, it took almost 224 CPU-hours and 2 GB of memory on a Cray C916 to find 64 dependencies. Calendar time for this was 9.5 days.

Table 1 gives some statistics on the RSA130, RSA140 and RSA155 factorisations. The Mips-year times given are only rough estimates. Extrapolation of these figures is risky, since various algorithmic improvements were incorporated in the later factorisations and the scope for further improvements is unclear.

7 Parallel Linear Algebra

At present, the main obstacle to a fully parallel and scalable implementation of GNFS (and, to a lesser extent, MPQS) is the linear algebra. Montgomery's block Lanczos program runs on a single processor and requires enough memory

Table 1. RSA130, RSA140 and RSA155 factorisations

	RSA130	RSA140	RSA155
Total CPU time in Mips-years	1000	2000	8000
Matrix columns	3.5×10^6	4.7×10^6	6.7×10^6
Total nonzeros	1.4×10^8	1.5×10^8	4.2×10^8
Nonzeros per column	39	32	62
Matrix solution time (on Cray C90/C916)	68 hours	100 hours	224 hours

to store the sparse matrix (about five bytes per nonzero element). It is possible to distribute the block Lanczos solution over several processors of a parallel machine, but the communication/computation ratio is high [38].

Similar remarks apply to some of the best algorithms for the integer discrete logarithm problem. The main difference is that the linear algebra has to be performed over a (possibly large) finite field rather than over $\mathbf{GF}(2)$.

In this section we present some preliminary ideas on how to implement the linear algebra phase of GNFS (and MPQS) in parallel on a network of relatively small machines. This work is still in progress and it is too early to give definitive results.

7.1 Assumptions

Suppose that a collection of P machines is available. It will be convenient to assume that $P = q^2$ is a perfect square. For example, the machines might be 400 Mhz PCs with 256 MByte of memory each, and $P = 16$ or $P = 64$.

We assume that the machines are connected on a reasonably high-speed network. For example, this might be multiple 100Mb/sec Ethernets or some higher-speed proprietary network. Although not essential, it is desirable for the physical network topology to be a $q \times q$ grid or torus. Thus, we assume that a processor $P_{i,j}$ in row i can communicate with another processor $P_{i,j'}$ in the same row or broadcast to all processors $P_{i,*}$ in the same row without interference from communication being performed in other rows. (Similarly for columns.) The reason why this topology is desirable is that it matches the communication patterns necessary in the linear algebra: see for example [4–6].

To simplify the description, we assume that the standard Lanczos algorithm is used. In practice, a block version of Lanczos [37] would be used, both to take advantage of word-length Boolean operations and to overcome the technical problem that $u^T u$ can vanish for $u \neq 0$ when we are working over a finite field. The overall communication costs are nearly the same for blocked and unblocked Lanczos, although blocking reduces startup costs.

If R is the matrix of relations, with one relation per column, then R is a large, sparse matrix over $\mathbf{GF}(2)$ with slightly more columns than rows. (Warning: in the literature, the transposed matrix is often considered, i.e. rows and columns are often interchanged.) We aim to find a nontrivial *dependency*, i.e. a nonzero vector x such that $Rx = 0$. If we choose $b = Ry$ for some random y, and solve $Rx = b$, then $x - y$ is a dependency (possibly trivial, if $x = y$). Thus, in the description below we consider solving a linear system $Rx = b$.

The Lanczos algorithm over the real field works for a positive-definite symmetric matrix. In our case, R is not symmetric (or even square). Hence we apply Lanczos to the *normal equations*

$$R^T Rx = R^T b$$

rather than directly to $Rx = b$. We do not need to compute the matrix $A = R^T R$, because we can compute Ax as $R^T(Rx)$. To compute $R^T z$, it is best to think of computing $(z^T R)^T$ because the elements of R will be scattered across the processors and we do not want to double the storage requirements by storing R^T as well as R.

The details of the Lanczos iteration are not important for the description below. It is sufficient to know that the Lanczos recurrence has the form

$$w_{i+1} = Aw_i - c_{i,i}w_i - c_{i,i-1}w_{i-1}$$

where the multipliers $c_{i,i}$ and $c_{i,i-1}$ can be computed using inner products of known vectors. Thus, the main work involved in one iteration is the computation of $Aw_i = R^T(Rw_i)$. In general, if R has n columns, then w_i is a *dense* n-vector. The Lanczos process terminates with a solution after $n + O(1)$ iterations [37].

7.2 Sparse matrix-vector multiplication

The matrix of relations R is sparse and almost square. Suppose that R has n columns, $m \approx n$ rows, and ρn nonzeros, i.e. ρ nonzeros per column. Consider the computation of Rw, where w is a dense n-vector. The number of multiplications involved in the computation of Rw is ρn. (Thus, the number of operations involved in the complete Lanczos process is of order $\rho n^2 \ll n^3$.)

For simplicity assume that $q = \sqrt{P}$ divides n, and the matrix R is distributed across the $q \times q$ array of processors using a "scattered" data distribution to balance the load and storage requirements [5]. Within each processor $P_{i,j}$ a sparse matrix representation will be used to store the relevant part $R_{i,j}$ of R. Thus the local storage requirement is $O(\rho n/q^2 + n/q)$.

After each processor computes its local matrix-vector product with about $\rho n/q^2$ multiplications, it is necessary to sum the partial results (n/q-vectors) in each row to obtain n/q components of the product Rw. This can be done in several ways, e.g. by a systolic computation, by using a binary tree, or by each processor broadcasting its n/q-vector to other processors in its row and summing the vectors received from them to obtain n/q components of the product Rw. If broadcasting is used then the results are in place to start the computation of $R^T(Rw)$. If an inner product is required then communication along columns of the processor array is necessary to transpose the dense n-vector (each diagonal processor $P_{i,i}$ broadcasts its n/q elements along its column).

We see that, overall, each Lanczos iteration requires computation $O(\rho n/q^2 + n)$ on each processor, and communication of $O(n)$ bits along each row/column of the processor array. Depending on details of the communication hardware, the time for communication might be $O((n/q)\log q)$ (if the binary tree method is used) or even $O(n/q+q)$ (if a systolic method is used with pipelining). In the following discussion we assume that the communication time is proportional to the number of bits communicated along each row or column, i.e. $O(n)$ per iteration (not the total number of bits communicated per iteration, which is $O(nq)$).

The overall time for n iterations is

$$T_P \approx \alpha\rho n^2/P + \beta n^2 \ , \tag{5}$$

where the constants α and β depend on the computation and communication speeds respectively. (We have omitted the communication "startup" cost since this is typically much less than βn^2.)

Since the time on a single processor is $T_1 \approx \alpha\rho n^2$, the *efficiency* E_P is

$$E_P = \frac{T_1}{PT_P} \approx \frac{\alpha\rho}{\alpha\rho + \beta P}$$

and the condition for $E_P \geq 1/2$ is

$$\alpha\rho \geq \beta P \ .$$

Typically β/α is large (50 to several hundred) because communication between processors is much slower than computation on a processor. The number of nonzeros per column, ρ, is typically in the range 30 to 100. Thus, we can not expect high efficiency.

Fortunately, high efficiency is not crucial. The essential requirements are that the (distributed) matrix R fits in the local memories of the processors, and that the time for the linear algebra does not dominate the overall factorisation time. Because sieving typically takes *much* longer than linear algebra on a single processor, we have considerable leeway.

For example, consider the factorisation of RSA155 (see Table 1 above). We have $n \approx 6.7 \times 10^6$, $\rho \approx 62$. Thus $n^2 \approx 4.5 \times 10^{13}$. If the communications speed along each row/column is a few hundred Mb/sec then we expect β to be a small multiple of 10^{-8} seconds. Thus βn^2 might be (very roughly) 10^6 seconds or say 12 days. The sieving takes 8000 Mips-years or 20 processor-years if each processor runs at 400 Mips. With P processors this is $20/P$ years, to be compared with $12/365$ years for communication. Thus,

$$\frac{\text{sieving time}}{\text{communication time}} \approx \frac{600}{P}$$

and the sieving time dominates unless $P > 600$. For larger problems we expect the sieving time to grow at least as fast as n^2.

For more typical factorisations performed by MPQS rather than GNFS, the situation is better. For example, using Arjen Lenstra's implementation of PP-MPQS in the Magma package, we factored a 102-decimal digit composite c_{102} (a factor of $14^{213} - 1$) on a 4×250 Mhz Sun multiprocessor. The sieving time was 11.5 days or about 32 Mips-years. After combining partial relations the matrix R had about $n = 73000$ columns. Thus $n^2 \approx 5.3 \times 10^9$. At 100 Mb/sec it takes less than one minute to communicate n^2 bits. Thus, although the current Magma implementation performs the block Lanczos method on only one processor, there is no reason why it could not be performed on a network of small machines (e.g. a "Beowulf" cluster) and the communication time on such a network would only be a few minutes.

7.3 Reducing communication time

If our assumption of a grid/torus topology is not satisfied, then the communication times estimated above may have to be multiplied by a factor of order $q = \sqrt{P}$. In order to reduce the communication time (without buying better hardware) it may be worthwhile to perform some steps of Gaussian elimination before switching to Lanczos.

If there are k relations whose largest prime is p, we can perform Gaussian elimination to eliminate this large prime and one of the k relations. (The work can be done on a single processor if the relations are hashed and stored on processors according to the largest prime occurring in them.) Before the elimination we expect each of the k relations to contain about ρ nonzeros; after the elimination each of the remaining $k - 1$ relations will have about $\rho - 1$ additional nonzeros (assuming no "cancellation").

From (5), the time for Lanczos is about

$$T = n(\alpha z/P + \beta n) ,$$

where $z = \rho n$ is the weight (number of nonzeros) of the matrix R. The effect of eliminating one relation is $n \to n' = n - 1$, $z \to z' = z + (\rho - 1)(k - 1) - \rho$, $T \to T' = n'(\alpha z'/P + \beta n')$. After some algebra we see that $T' < T$ if

$$k < \frac{2\beta P}{\alpha\rho} + 3 .$$

For typical values of $\beta/\alpha \approx 100$ and $\rho \approx 50$ this gives the condition $k < 4P + 3$.

The disadvantage of this strategy is that the total weight z of the matrix (and hence storage requirements) increases. In fact, if we go too far, the matrix becomes full (see the RSA129 example in §4.2).

As an example, consider the c_{102} factorisation mentioned at the end of §7.2. The initial matrix R had $n \approx 73000$; we were able to reduce this to $n \approx 32000$, and thus reduce the communication requirements by 81%, at the expense of increasing the weight by 76%. Our (limited) experience indicates that this behaviour is typical – we can usually reduce the communication requirements by a factor of five at the expense of at most doubling the weight. However, if we try to go much further, there is an "explosion", the matrix becomes dense, and the advantage of the Lanczos method over Gaussian elimination is lost. (It would be nice to have a theoretical explanation of this behaviour.)

8 Historical Factoring Records

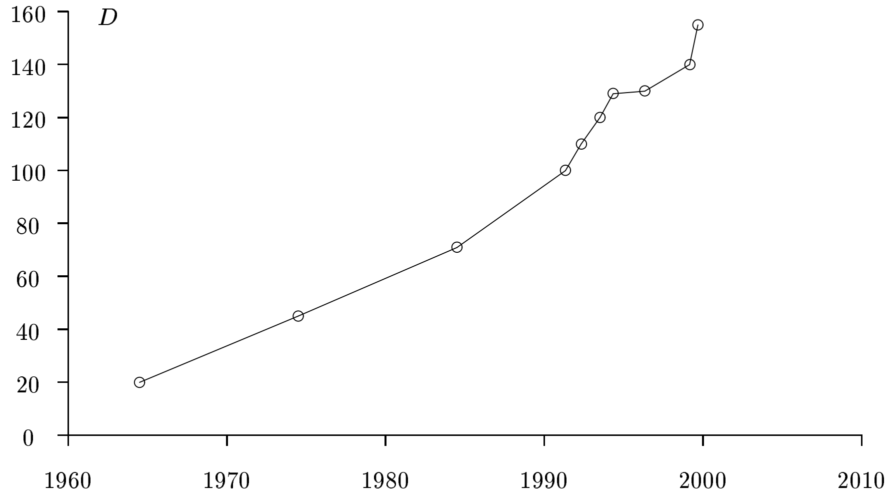

Figure 3: Size of "general" number factored versus year

Figure 3 shows the size D (in decimal digits) of the largest "general" number factored against the year it was done, from 1964 (20D) to April 2000 (historical data from [41, 44, 55]).

8.1 Curve fitting and extrapolation

Let D be the number of decimal digits in the largest "general" number factored by a given date. From the theoretical time bound for GNFS, assuming Moore's law, we expect $D^{1/3}$ to be roughly a linear function of calendar year (in fact $D^{1/3}(\ln D)^{2/3}$ should be linear, but given the other uncertainties we have assumed for simplicity that $(\ln D)^{2/3}$ is roughly a constant). Figure 4 shows $D^{1/3}$ versus year Y.

The straight line shown in the Figure 4 is

$$D^{1/3} = \frac{Y - 1928.6}{13.24} \quad \text{or equivalently} \quad Y = 13.24 D^{1/3} + 1928.6 \, ,$$

and extrapolation, for what it is worth, gives $D = 309$ (i.e. 1024 bits) in the year $Y = 2018$.

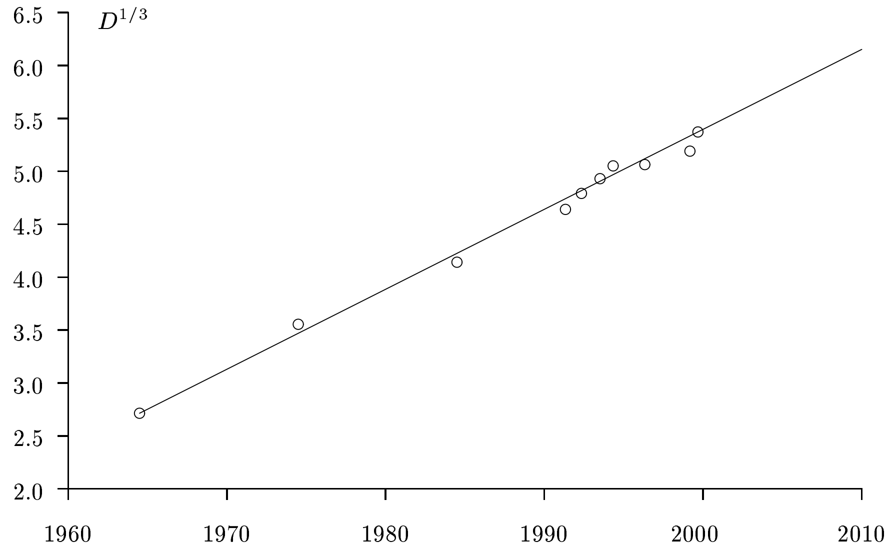

Figure 4: $D^{1/3}$ versus year Y

9 Summary and Conclusions

We have sketched some algorithms for integer factorisation. The most important are ECM, MPQS and NFS. As well as their inherent interest and applicability to other areas of mathematics, the advent of public key cryptography has given them practical significance.

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 is built, large factorisations will remain an interesting challenge.

A survey similar to this one was written a decade ago [3]. Comparing the examples there with those given here 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 [3] had "only" 111 decimal digits. In 1994, RSA129 was factored by MPQS. In 1996, GNFS overtook MPQS by factoring RSA130. In 1999, RSA155 also fell to GNFS.

From the predicted run time for GNFS, we would expect RSA155 to take 6.5 times as long as RSA140[5]. Moore's law predicts that circuit densities will double every 18 months or so. Thus, as long as Moore's law continues to apply

[5] It actually took only about 4.0 times as long, thanks to algorithmic improvements such as better polynomial selection.

and results in correspondingly more powerful parallel computers, we expect to get 3–4 decimal digits per year improvement in the capabilities of GNFS, even without any algorithmic improvements. The extrapolation from historical figures is more optimistic: it predicts 6–7 decimal digits per year in the near future. The discrepancy is because the progress over the past decade is due in large part to algorithmic improvements. Althouh these seem unlikely to be repeated, one can not rule out the possibility..

Regarding the size of factors found by ECM, we expect an increase of about 1.5 decimal digits per year from extrapolation of historical data (this is slightly greater than Moore's law predicts).

The best algorithms for the integer discrete logarithm problem are analogous to NFS, requiring the solution of large, sparse linear systems over finite fields, and similar remarks apply to such algorithms.

Concerning the cryptographic consequences, we can say that 512-bit RSA keys are already insecure. 1024-bit RSA keys should remain secure for at least fifteen 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. In the long term, public key algorithms based on the discrete logarithm problem for elliptic curves appear to be a promising alternative to RSA or to algorithms based on the integer discrete logarithm problem.

Acknowledgements

Thanks are due to Peter Montgomery, Brian Murphy, Andrew Odlyzko, John Pollard, 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. H. Boender and H. J. J. te Riele, *Factoring integers with large prime variations of the quadratic sieve,* Experimental Mathematics, **5** (1996), 257–273.
3. R. P. Brent, Vector and parallel algorithms for integer factorisation, *Proceedings Third Australian Supercomputer Conference* University of Melbourne, December 1990, 12 *pp.* `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub122.html` .
4. R. P. Brent, The LINPACK benchmark on the AP 1000, *Proceedings of Frontiers '92* (McLean, Virginia, October 1992), IEEE Press, 1992, 128-135. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub130.html`
5. R. P. Brent, Parallel algorithms in linear algebra, *Algorithms and Architectures: Proc. Second NEC Research Symposium* held at Tsukuba, Japan, August 1991 (edited by T. Ishiguro), SIAM, Philadelphia, 1993, 54-72. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub128.html`

6. R. P. Brent and P. E. Strazdins, Implementation of the BLAS level 3 and Linpack benchmark on the AP 1000, *Fujitsu Scientific and Technical Journal* 29, 1 (March 1993), 61-70. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub136.html`
7. 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.* `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub161.html` .
8. R. P. Brent, Some parallel algorithms for integer factorisation *Proc. Europar'99*, Toulouse, Sept. 1999. *LNCS* **1685**, Springer-Verlag, Berlin, 1–22.
9. R. P. Brent, *Large factors found by ECM,* Oxford University Computing Laboratory, March 2000. `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/champs.txt` .
10. 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` .
11. T. R. Caron and R. D. Silverman, Parallel implementation of the quadratic sieve, *J. Supercomputing* **1** (1988), 273–290.
12. 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` .
13. 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` .
14. 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.
15. H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, Berlin, 1993.
16. S. Contini, The factorization of RSA-140, RSA Laboratories Bulletin **10**, 8 (March 1999). Available from `http://www.rsa.com/rsalabs/html/bulletins.html` .
17. D. Coppersmith, A. Odlyzko and R. Schroeppel, Discrete logarithms in $\mathbf{GF}(p)$, *Algorithmica* **1** (1986), 1–15.
18. 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.
19. R. Crandall and B. Fagin, Discrete weighted transforms and large-integer arithmetic, *Math. Comp.* **62** (1994), 305–324.
20. D. Deutsch, Quantum theory, the Church-Turing principle and the universal quantum computer, *Proc. Roy. Soc. London, Ser. A* **400** (1985), 97–117.
21. D. Deutsch, Quantum computational networks, *Proc. Roy. Soc. London, Ser. A* **425** (1989), 73–90.
22. M. Elkenbracht-Huizing, A multiple polynomial general number field sieve *Algorithmic Number Theory – ANTS III*, *LNCS* **1443**, Springer-Verlag, Berlin, 1998, 99–114.
23. K. F. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag, Berlin, 1982.

24. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison Wesley, third edition, 1997.
25. B. A. LaMacchia and A. M. Odlyzko, Solving large sparse systems over finite fields, *Advances in Cryptology, CRYPTO '90* (A. J. Menezes and S. A. Vanstone, eds.), *LNCS* **537**, Springer-Verlag, Berlin, 109–133.
26. C. Lanczos, Solution of systems of linear equations by minimized iterations, *J. Res. Nat. Bureau of Standards* **49** (1952), 33–53.
27. S. Lang, *Elliptic Curves – Diophantine Analysis*, Springer-Verlag, Berlin, 1978.
28. R. S. Lehman, Factoring large integers, *Math. Comp.* **28** (1974), 637–646.
29. 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.
30. 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.
31. 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.
32. A. K. Lenstra and M. S. Manasse, Factoring by electronic mail, *Proc. Eurocrypt '89*, *LNCS* **434**, Springer-Verlag, Berlin, 1990, 355–371.
33. A. K. Lenstra and M. S. Manasse, Factoring with two large primes, *Math. Comp.* **63** (1994), 785–798.
34. H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Annals of Mathematics* (2) **126** (1987), 649–673.
35. 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` .
36. P. L. Montgomery, Square roots of products of algebraic numbers, *Mathematics of Computation 1943 – 1993, Proc. Symp. Appl. Math.* **48** (1994), 567–571.
37. P. L. Montgomery, A block Lanczos algorithm for finding dependencies over $\mathbf{GF}(2)$, *Advances in Cryptology: Proc. Eurocrypt'95*, *LNCS* **921**, Springer-Verlag, Berlin, 1995, 106–120. `ftp://ftp.cwi.nl/pub/pmontgom/BlockLanczos.psa4.gz` .
38. P. L. Montgomery, *Parallel block Lanczos*, Microsoft Research, Redmond, USA, 17 January 2000 (transparencies of a talk presented at RSA 2000).
39. M. A. Morrison and J. Brillhart, A method of factorisation and the factorisation of F_7, *Math. Comp.* **29** (1975), 183–205.
40. B. A. Murphy, Modelling the yield of number field sieve polynomials, *Algorithmic Number Theory – ANTS III*, *LNCS* **1443**, Springer-Verlag, Berlin, 1998, 137–150.
41. B. A. Murphy, *Polynomial selection for the number field sieve integer factorisation algorithm*, Ph. D. thesis, Australian National University, July 1999.
42. B. A. Murphy and R. P. Brent, On quadratic polynomials for the number field sieve, *Australian Computer Science Communications* **20** (1998), 199–213. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub178.html` .
43. 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.
44. A. M. Odlyzko, The future of integer factorization, *CryptoBytes* **1**, 2 (1995), 5–12. Available from `http://www.rsa.com/rsalabs/pubs/cryptobytes` .
45. J. M. Pollard, A Monte Carlo method for factorisation, *BIT* **15** (1975), 331–334.
46. C. Pomerance, The quadratic sieve factoring algorithm, *Advances in Cryptology, Proc. Eurocrypt '84, LNCS* **209**, Springer-Verlag, Berlin, 1985, 169–182.
47. C. Pomerance, The number field sieve, *Proceedings of Symposia in Applied Mathematics* **48**, Amer. Math. Soc., Providence, Rhode Island, 1994, 465–480.

48. C. Pomerance, A tale of two sieves, *Notices Amer. Math. Soc.* **43** (1996), 1473–1485.
49. 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.
50. 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/` .
51. H. te Riele *et al*, *Factorization of a 512-bits RSA key using the number field sieve*, announcement of 26 August 1999, `http://www.loria.fr/~zimmerma/records/RSA155` .
52. 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.
53. H. Riesel, *Prime numbers and computer methods for factorization,* 2nd edition, Birkhäuser, Boston, 1994.
54. R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM* **21** (1978), 120–126.
55. RSA Laboratories, Information on the RSA challenge, `http://www.rsa.com/rsalabs/html/challenges.html` .
56. R. S. Schaller, Moore's law: past, present and future, *IEEE Spectrum* **34**, 6 (June 1997), 52–59.
57. 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
58. P. W. Shor, Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Computing* **26** (1997), 1484–1509.
59. R. D. Silverman, The multiple polynomial quadratic sieve, *Math. Comp.* **48** (1987), 329–339.
60. U. Vazirani, Introduction to special section on quantum computation, *SIAM J. Computing* **26** (1997), 1409–1410.
61. D. H. Wiedemann, Solving sparse linear equations over finite fields, *IEEE Trans. Inform. Theory* **32** (1986), 54–62.
62. J. Zayer, *Faktorisieren mit dem Number Field Sieve*, Ph. D. thesis, Universität des Saarlandes, 1995.
63. P. Zimmermann, *The ECMNET Project*, `http://www.loria.fr/~zimmerma/records/ecmnet.html`.