A GENERAL NUMBER FIELD SIEVE IMPLEMENTATION

DANIEL J. BERNSTEIN, A. K. LENSTRA

ABSTRACT. The general number field sieve is the asymptotically fastest—and by far most complex—factoring algorithm known. We have implemented this algorithm, including five practical improvements: projective polynomials, the lattice sieve, the large prime variation, character columns, and the positive square root method. In this paper we describe our implementation and list some factorizations we obtained, including the record factorization of $2^{523} - 1$.

1. INTRODUCTION

The general number field sieve (GNFS) [3] is a modified version of the special number field sieve (SNFS) [8; 9]. GNFS factors arbitrary integers n in heuristic time

$$\exp((c_g + o(1)) \log^{1/3} n \log^{2/3} \log n)$$

as $n \to \infty$. Here $c_g = (64/9)^{1/3} \approx 1.9$. For the special integers n where SNFS is applicable, GNFS takes time

$$\exp((c_s + o(1)) \log^{1/3} n \log^{2/3} \log n),$$

with $c_s = (32/9)^{1/3} \approx 1.5$. These asymptotic estimates should be compared with the time

$$\exp((1 + o(1)) \log^{1/2} n \log^{1/2} \log n)$$

taken by the multiple polynomial quadratic sieve (QS) [14], generally regarded as the best general-purpose factoring method for large integers.

We implemented GNFS on Bellcore's MasPar[1], a SIMD (single instruction multiple data) computer with 16384 processors, and used it to obtain some

1991 *Mathematics Subject Classification.* Primary 11Y05, 11Y40.

Key words and phrases. Factoring algorithm, algebraic number fields.

Thanks to Joe Buhler, Hendrik Lenstra, John Pollard, and Carl Pomerance for their helpful suggestions, and to Andrew Odlyzko for his help with the factorization of $2^{523} - 1$. The first author was supported in part by a National Science Foundation Graduate Fellowship and by Bellcore.

[1] It is the policy of Bellcore to avoid any statements of comparative analysis or evaluation of products or vendors. Any mention of products or vendors in this presentation or accompanying printed materials is done where necessary for the sake of scientific accuracy and precision, or to provide an example of a technology for illustrative purposes, and should not be construed as either a positive or negative commentary on that product or vendor. Neither the inclusion of a product or a vendor in this presentation or accompanying printed materials, nor the omission of a product or a vendor, should be interpreted as indicating a position or opinion of that product or vendor on the part of the presenter or Bellcore.

factorizations, as described in Section 12. To speed the implementation we used five modifications to the basic algorithm: projective polynomials [3], Pollard's lattice sieve [13], the large prime variation [8; 9; 10], character columns [1], and Couveignes's positive square root method [4]. In this paper we describe our implementation. We also present some factorizations we obtained with GNFS.

We suspect that for general numbers GNFS is competitive with QS at the edge of what we can factor in a reasonable time today, between 120 and 130 digits. For larger numbers it is faster. For smaller numbers it is usable, though slower than QS in its current form—but see Section 13 for comments on this situation.

We warn the reader that the word "smooth" is defined anew in several sections of this paper. To avoid confusion we give each different use of "smooth" a unique prefix: basically smooth in Section 3, sieve-smooth and prec-smooth in Section 6, and rat-smooth and alg-smooth in Section 7. We write $\#S$ for the size of a set S. Most further notation for this paper is established in Section 3.

2. Outline of the implementation

We refer the reader to [3] for an explanation of how the number field sieve works; except for a few brief proofs in Section 11 we do not justify the algorithm here.

We begin with a composite number $n > 0$ to be factored. We assume as known that n is odd—in fact that it has no factors smaller than some reasonable bound, say the largest integer that fits into a computer word. We also assume that n is not a power of a prime. These facts will all be apparent after an application of trial division, standard compositeness and power-of-prime tests [8], Pollard's ρ method [7], and the elliptic curve method [11], all of which are tried before the number field sieve.

GNFS can be divided into four stages. In the first stage, described in Sections 3 through 5, we choose parameters and precompute various information about an algebraic number field associated with n. In the second and most time-consuming stage, described in Sections 6 and 7, we perform the heart of the number field sieve, looking for "relations." In the third stage, described in Sections 8 and 9, we build a matrix out of the relations and find random elements of the nullspace of the matrix. In the final stage, described in Sections 10 and 11, we effectively take a square root of a large element of our number field.

Figure 1 shows the flow of data in GNFS. Upper case names such as `POLY` are data files, i.e., sets stored on the computer; lower case names such as `pnfls` are programs. Each node is labeled with a section number. Data flows downwards through the graph.

3. Choosing the polynomial

The entire computation begins with a file `POLY`, which contains the number n to be factored together with a low-degree polynomial f and an integer m such that n divides $f(m)$. The degree d of f is most commonly 3 or 5; it is required to be odd because of a restriction in the square root stage, as described in Section 10.

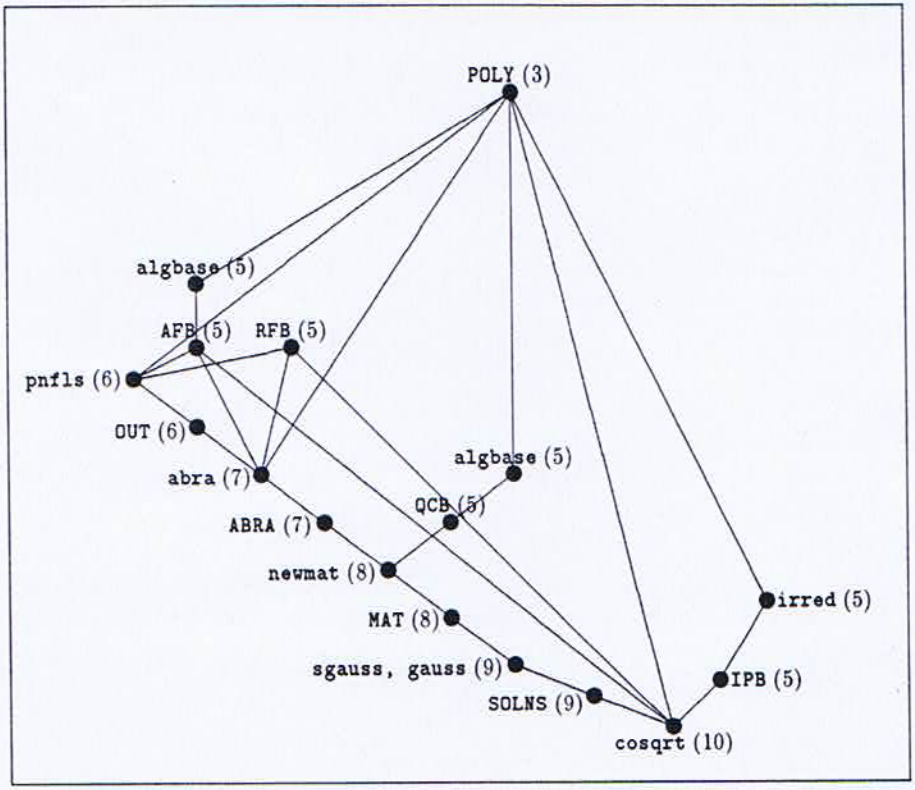

FIGURE 1. The flow of data in our GNFS implementation.

(See Remark 10.1. We can in principle use even degrees, as we have a single-processor implementation of the square root method described in [3], which can work with any degree. We have been using a MasPar implementation of another method, which demands odd degrees.)

Several mathematical objects are implicitly associated with the polynomial f we choose. Most important is the projective two-variable norm polynomial, which we define as $N(a,b) = f(a/b)b^d$; see Section 11 for a number-theoretic interpretation of this object. Another is the monic polynomial $g(x) = f(x/c_d)c_d^{d-1}$, where c_d is the leading coefficient of f. Another is the number field

$$\mathbf{Q}[\alpha]/f(\alpha) = \mathbf{Q}[\omega]/g(\omega).$$

Here α could be regarded as a root of f within the complex plane, and $\omega = \alpha c_d$ could be regarded as a root of g; but neither α nor ω is ever computed explicitly. We actually work within the number ring $\mathbf{Z}[\omega]/g(\omega)$ inside our number field. Elements of this number ring are represented as polynomials $\sum_{0 \le i < d} z_i \omega^i$ where each z_i is an integer. Finally we will need the discriminant $\operatorname{disc}(g)$ (but see Remark 10.4 for a way to avoid computing this quantity).

For special numbers n, such as numbers of the form $b^c \pm 1$ where SNFS is applicable, we choose `POLY` by hand. Otherwise we let the computer search for a good polynomial by brute force in a given range of m's, as described in the next section.

All else being equal, one polynomial f is better than another with a similar value of m if the norm values $N(a,b)$ are more likely to be basically smooth for a and b small. Here x is basically smooth if its prime factors are small. (It is not necessary to define "small" precisely.) See next section for further details of how we find f in general.

We measure f (for a fixed d) in two ways. The first is coefficient size: if f has coefficients that are small in absolute value then $N(a,b)$ will be small and hence likely to be basically smooth. The second measurement is more algebraic: we compute an average logarithmic subtraction from $\log N(a,b)$ given the roots of f modulo tiny primes. Consider, for example, a polynomial f congruent to x^4-1 modulo 5, so that f has four roots modulo 5. Then $N(a,b)$ will be divisible by 5 for four-fifths of all choices of a, b, in some sense. So if we remove the first factor of 5 (if any) from $N(a,b)$ then $\log N(a,b)$ becomes an average of $(4/5)\log 5$ smaller. We compute these logarithmic subtractions for tiny primes through 19.

Once we have chosen f we choose several further parameters: the rational prime bound max`RFB` (on the order of 10^6), the algebraic prime bound max`AFB` (so far always chosen equal to max`RFB`), the large prime bound L (on the order of 10^8), a quadratic character base size #`QCB` (so far always chosen as 100), an inert prime base size #`IPB` (on the order of 10^5), and a range $\{e\}$ of integers (typically all values between 1 and 10^4—see Remark 3.1).

All these parameters except #`IPB` and #`QCB` are chosen with the constraint that the matrix constructed in Section 8 has a nontrivial nullspace. The size #`QCB` must be large enough that the quadratic characters are highly likely to span a certain vector space; see Section 8 and [3]. The size #`IPB` must be large enough that a certain inequality holds; see Remark 10.3 and Remark 11.1.

3.1. *Remark.* The range $\{e\}$ determines the length of one of the sides of the sieving parallelogram (see Section 6), and can be chosen freely. The other length is hard-wired in our sieving program and fixed at compile time; its choice depends on the design of the sieving program and is severely limited by the architecture of the MasPar.

3.2. *Remark.* In principle we should check that m and c_d do not have any factors in common with n, that the derivative $f'(m)$ does not have any factors in common with n, and that there are enough inert primes, but in practice none of these problems will occur. If f is "bad" in any way then the algorithm will fail in an obvious manner. We describe these failures at the points where they may occur. See Remark 5.2 and Remark 10.2.

3.3. *Remark.* The possibility of f not being monic characterizes the "projective" number field sieve. It permits much smaller coefficients at very little expense. Actually the projective number field can be chosen with two parameters m_1 and m_2 in place of m; the idea is that m is a fraction, m_1/m_2. The additional

possibility of $m_2 \neq 1$ characterizes the "homogeneous" polynomial variation. See [3] for details of this generalization. As m_2 intrudes (a bit) at every step of the computation, and does not show any obvious advantages, we have not used it. But see Section 13 for further comments.

4. Searching for a polynomial

The time taken by GNFS depends heavily on the size of the polynomial f. In this section we describe how we search for polynomials when we do not know a special form for n.

To fix ideas we set $d = 5$. Here is the polynomial-searching problem. We are given n. We want to find a positive integer m in a given range and six integer coefficients $c_0, \ldots, c_5$ such that $c_0 + c_1 m + \cdots + c_5 m^5$ is a positive multiple of n, with all the c_i small in absolute value.

The straightforward solution is to pick m around $n^{1/6}$ and to expand n in base m. But there are many other polynomials for n. For a detailed analysis of the polynomial existence problem we refer to [3]; here is a rough outline of the argument showing that we can expect that much better polynomials exist. Say $d = 5$ and n is around 10^{145}. We have seven parameters, m and c_0 through c_5, to change; let us choose m around 25 digits, c_5 around 20 digits, and the other c_i up to 25 digits. This gives approximately 10^{170} values of $f(m)$; it is reasonable to expect that at least 10^{25} of them will equal (or at least be divisible by) n. The five c_i other than c_5 vary wildly with the precise value of m. If they are independently and uniformly distributed then there is some polynomial for n where each of those five c_i is a factor of $10^{25/5} = 10^5$ better than the maximum. In other words there should be a polynomial for n with $m \approx 10^{25}$ and $|c_i| < 10^{20}$.

By the same argument, if we use $m = m_1/m_2$ as described in [3], choosing m_1 and m_2 around 25 digits, then there should be a polynomial for n with $m_j \approx 10^{25}$ and $|c_i| < 10^{16}$. But it is already extremely difficult to find optimal c_i without m_2, so we will ignore m_2 for the remainder of this section. In case of any breakthrough in polynomial-searching methods it will obviously be important to use m_2.

Unfortunately we do not know any way substantially faster than brute force to generate a good polynomial for general n. We start from a polynomial with a given m. We choose some integer k, replace m by $m - k$, and compute the polynomial for this new value of m. If any coefficient is larger than $m/2$ in absolute value, we subtract m from it and add 1 to the next coefficient. Finally we check whether all the coefficients are small. We repeat as long as necessary.

It is easy to choose k so that c_4 is bounded by a small multiple of c_5. But this still leaves four coefficients, c_0 through c_3, which can range up to about $m/2$. If we search through 10^{12} polynomials in this way then we expect to find some value of m such that each of c_0 through c_3 is at most $m/2000$ in absolute value. This is good, though still nowhere near optimal.

We emphasize that polynomial-searching is highly underdeveloped. There is much unexploited structure in the polynomial-searching problem. It *appears* far more tractable than factoring itself. Surely we can do better than brute force?

5. Computing bases

After settling on a polynomial we compute four bases: the rational factor base `RFB`, the algebraic factor base `AFB`, the quadratic character base `QCB`, and the inert prime base `IPB`. All these files are stored in binary format so that they can be loaded quickly. All the parameters listed in Section 3 except $\{\epsilon\}$ are used in constructing these factor bases.

`RFB` does not depend on `POLY`. It consists of all odd primes up to max`RFB`.

`AFB` is a list of all (p, r) pairs with p dividing $f(r)$ and $0 \le r < p$; p is restricted to the odd primes below max`AFB`. There may be as many as d roots r of f modulo p. Our `algbase` program computes them as in [7, Section 4.6.2]. We also include pairs (p, ∞) (represented inside the computer as (p, p)) with p dividing c_d, as per [3]. For all polynomials used in practice, the number of (p, r) pairs in the algebraic factor base with p in any given range will be very close to the number of primes p in the rational factor base in the same range.

`IPB` is a list of #`IPB` odd primes ℓ that remain inert in the number ring, i.e., for which f is irreducible modulo ℓ. We also require that no ℓ divide c_d. These conditions imply that ℓ does not divide the discriminant $\mathrm{disc}(g)$. See Remark 11.1 for details on the choice of #`IPB`. A program `irred`, similar to `algbase` but running on the MasPar, looks for inert primes among all primes $\ell > 10000$. (Actually it would be better to generate primes ℓ down from the computer's word size; see Remark 11.1.) It checks 16384 possibilities for ℓ at once using the tests in [7, Section 4.6.2].

Finally, `QCB` is a list of (q, s) pairs with the odd prime q dividing $f(s)$; here, unlike `AFB`, q is required to be larger than the large prime bound L. We choose the first #`QCB` primes q larger than L. As indicated in Section 3 we use only #`QCB` $= 100$ different pairs, so `QCB` is a small file.

5.1. *Remark.* In principle the q's in `QCB` should not divide c_d or the derivative $f'(s)$. These conditions rarely fail, but if they do fail then they will fail silently (i.e., without overt effects on the computation). On the other hand, we have not bothered checking for these failures, for reasons explained in Remark 8.1.

5.2. *Remark.* The files computed here could, in highly improbable circumstances, be much too small. In this case f is bad and a different f must be chosen.

6. Sieving

In this section we regard the rational primes p as pairs $(p, m \bmod p)$. We thus regard the rational and algebraic factor bases as a single collection of pairs (p, r), of size #`RFB` + #`AFB`. (For simplicity we ignore any pairs (p, ∞).) In this section we use the parameter $\{e\}$ as well as many sieving parameters chosen heuristically.

The object of the sieving stage is to produce an (a, b) pair, with a and b relatively prime integers that fit easily inside a computer word, such that $b > 0$ and $a \equiv br \pmod p$ for several pairs (p, r)—a sieve-smooth (a, b) pair, for short. Actually the sieving stage should produce as many sieve-smooth (a, b) pairs as possible in a reasonable amount of time.

6.1. *Sieving without special-q.* Here is the direct approach to sieving. We consider a region of the (a,b)-plane of area A. We divide the region into some number X of roughly equal-area pieces. We then do the following:

```
1.      For each piece
2.        For each (p, r)
3.          Figure out which (a, b) inside the piece have a ≡ br (mod p)
4.          "Hit" all those spots
5.        For each (a, b) in the piece
6.          Check if (a, b) has enough hits
```

For example, one might choose $X = 1$ and initialize an array of counters, one counter for each (a,b) pair in a rectangle near the origin, say $0 < b < v$ and $-u < a < u$. (With these choices we say we are using the "plane method.") For each pair (p,r) one "hits" the counters for all (a,b) pairs in the lattice defined by $a \equiv br \pmod{p}$. After all (p,r) have been considered one can "read off" the sieve-smooth (a,b) pairs from the counters.

As another example, in other NFS implementations [8; 9] one runs through the (a,b) plane in horizontal slices. More precisely, one still considers $0 < b < v$ and $-u < a < u$, but this area is split into $X = v - 1$ pieces, each piece with a fixed b. (This is the "line method.") For each $b = 1, 2, \ldots$ one initializes counters for the line of a values between $-u$ and u. For each pair (p,r) one runs through the a values with $a \equiv br \pmod{p}$, hitting the appropriate counters. After this one can read off the sieve-smooth (a,b) pairs and continue on to the next b. One feature of this method is that it is easy to keep track of $br \pmod{p}$ as b increases. More importantly, we end up using only a small amount of memory.

As yet another example, one can split the plane into horizontal slices, and then split each slice into several pieces. This reduces memory requirements still further.

How much time and memory does the direct approach take? Denote the set of (p,r) by P. We use time T_1X for step 1, time $T_2X\#P$ for steps 2 and 3, and time T_4A for steps 5 and 6; recall that A is the total area of the (a,b) plane under consideration. (Here each T_i is the amount of time taken by some basic step; see Remark 6.13.) We use time roughly $T_3 \sum_{(p,r)\in P} A/p$ for step 4, assuming (as is correct for current theory and practice) that the A pairs (a,b) fall almost equally into equivalence classes of $a - br$ modulo p.

The total time is

$$T_1X + T_2X\#P + T_4A + T_3A\sum 1/p. \tag{6.2}$$

The total memory is some constant plus A/X.

6.3. *Sieving with special-q.* In the direct approach described above, we must consider every pair (a,b). Pollard [13] pointed out that a variation of the "special-q" method for QS of Davis and Holdridge [5] could be applied to NFS. This variation, called the "lattice sieve" for NFS by Pollard, is a general improvement, which with the proper parameter selection lets us consider only a fraction of all (a,b) pairs.

Here is how it works. Choose an ordering of the set P of pairs (p, r)—see Remark 6.9. For each (a, b) define "the special (q, s)" as the largest (p, r) with $a \equiv br \pmod{p}$. Then partition all (a, b) pairs according to their special (q, s) values. Instead of searching through all (a, b) we search through the (a, b) for each special $(q, s) \in Q$ in turn, for some subset Q of P. Some (a, b) pairs don't have a special (q, s) in Q, and thus are ignored, but with the proper choice of Q we can find most sieve-smooth (a, b) pairs.

Fix $(q, s) \in Q$. The set of sieve-smooth (a, b) pairs with this special (q, s) is a subset of the lattice of (a, b) pairs with $a \equiv bs \pmod{q}$. Which subset is it? It is the subset that is sieve-smooth for those (p, r) pairs no larger than (q, s).

So we run through all $(q, s) \in Q$ in turn. For a fixed (q, s) we compute two short vectors $\mathbf{C} = (a_c, b_c)$ and $\mathbf{E} = (a_e, b_e)$ that generate the lattice $a \equiv bs \pmod{q}$. (If we cannot find short enough vectors we simply throw q away. See Remark 6.11.) Then we find pairs (c, e) such that $(a, b) = c\mathbf{C} + e\mathbf{E}$ is hit by several (p, r) smaller than (q, s). To do this we transform the condition "$a \equiv br \pmod{p}$" into a similar condition on (c, e), and we search through a small rectangle of (c, e) values near the origin.

More formally, here is the algorithm.

1. For each (q, s) in Q
2. Identify the lattice of (a, b) with $a \equiv bs \pmod{q}$
3. Form coordinates (c, e) for some area of size C of this lattice
4. For each (p, r) in P
5. Transform "$a \equiv br \pmod{p}$" to the (c, e) plane
6. For each piece of the (c, e) plane
7. For each (p, r) smaller than (q, s)
8. "Hit" the appropriate (c, e) inside the piece
9. For each (c, e) in the piece
10. Check if (c, e) has enough hits

We have two parameters here corresponding to the parameters A and X in the direct approach: we choose an area of size C in the (c, e)-plane, and split it into Y roughly equal pieces for steps 6 through 10.

For example we can choose $Y = 1$ and use the plane method ("sieving by vectors") in the (c, e) plane. Or we can split the (c, e) plane down into $Y > 1$ lines, each with a fixed c—hence using the line method ("sieving by rows") in the (c, e) plane. We can even split the (c, e) plane into pieces smaller than lines, so as to save on memory. We could choose a different region of the (c, e)-plane for each (q, s) (as suggested in [13]), but for simplicity we keep C and Y constant (also following [13]).

How much time and memory does the special-q approach take? To simplify the analysis let us assume (to the detriment of the lattice sieve) that all pairs (p, r), rather than just those pairs up through (q, s), are used in steps 7 through 10.

As in our analysis of the direct approach, for each (q, s) we do a sieve taking time $T_1Y + T_2Y\#P + T_4C + T_3C\sum_{(p,r)} 1/p$. The total over all (q, s) is $T_1Y\#Q + T_2Y\#P\#Q + T_4C\#Q + T_3C\#Q\sum 1/p$. We also spend time $T_5\#Q + T_6\#P\#Q$

on the preparatory steps of the lattice sieve. The total time is

$$\#Q(T_1Y + T_2Y\#P + T_4C + T_3C\sum 1/p + T_5 + T_6\#P). \tag{6.4}$$

The total memory is some constant plus C/Y.

6.5. *Comparison of sieving methods.* We have presented two different approaches: the direct approach and the special-q approach. Each approach uses parameters that determine the time, memory, and yield of the sieving step: the direct approach uses X pieces of total area A in the (a, b) plane, and the special-q approach uses Y pieces of total area C in the (c, e) plane.

To compare the approaches we must first assume that they give the same yield. An area of A pairs (a, b) corresponds roughly to an area of $A/\max P$ of the (c, e) plane; as illustrated by Pollard [13] we will miss some, though not too many, sieve-smooth (a, b) pairs if we set C as small as $A/\max P$. To equalize the yield we might choose an area as large as $10A/\max P$ of the (c, e) plane. This number 10 is surely large enough if we choose Q following Pollard. In the following discussion we will take $C = VA/\max P$ where V is an unknown "fuzz factor" between 1 and 10.

Now for fixed A and C our time analyses (6.2), (6.4) show that sieving is faster when X and Y are smaller—more pieces means more overhead. On the other hand we use A/X or C/Y memory, so if memory is limited then we cannot choose X or Y too small.

Therefore, no matter which approach we use, we split the plane into as few pieces as possible, subject to the sole constraint that our sieve fit into memory. This is important both for theory and practice.

To continue our comparison of the two approaches we must consider two cases. Either we have an incredibly large amount of memory available (at least $A/\max P$), or we do not. In the first case we can choose $Y = 1$ (and $X \le \max P$). Again to the detriment of the lattice sieve we will ignore the X terms in (6.2) altogether: now the time (6.2) for the direct approach is over $A(T_4 + T_3\sum 1/p)$, and the time (6.4) for the special-q approach is

$$A\frac{\#Q}{\max P}\left((T_1 + T_5)\frac{\max P}{A} + (T_2 + T_6)\frac{\#P \max P}{A} + T_4V + T_3V\sum\frac{1}{p}\right).$$

Both in theory [3, Section 10] and in practice we may assume that A is at least roughly as large as $\#P \max P$. Now it is easy to see that the special-q approach is asymptotically some constant times $\max P/\#Q > \log \max P$ faster than the direct approach.

In the second case we do not have $A/\max P$ memory available. Then both approaches are constrained by the available memory, so as per our previous comments we assume that they both use as much memory as possible—in particular, they both use the same amount of memory. In other words A/X and $C/Y = VA/(Y \max P)$ should be equal. Hence $Y = XV/\max P$.

Now the time (6.2) for the direct approach is

$$X(T_1 + T_2\#P) + A(T_4 + T_3\sum\frac{1}{p}),$$

and the time (6.4) for the special-q approach is

$$X\frac{V\#Q}{\max P}(T_1+T_2\#P)+A\frac{V\#Q}{\max P}\left(T_4+T_3\sum\frac{1}{p}+T_5\frac{\max P}{VA}+T_6\frac{\#P\max P}{VA}\right).$$

The first terms may dominate the second terms, or they might not. In either case, the special-q approach is once again asymptotically $\log\max P$ faster than the direct approach.

Of course, the special-q approach is more complex than the direct approach, so in small examples [13] the direct approach may be faster, depending on the constants T_i and V. But as $\max P$ grows the special-q approach has a $\log\max P$ advantage, both in theory and in practice. For us $\max P$ is large enough that we suspect the special-q method is more than twice as fast as any implementation of the direct approach.

6.6. *Sieving on a MasPar.* The MasPar consists of a mesh of 128×128 SIMD processors with 64 KByte memory per processor. As shown in [6] the multiple polynomial variant of the quadratic sieve can be implemented quite efficiently on the MasPar. The MasPar is split into 128 rows, each handling the sieve line for one polynomial. Each sieve line is split into 128 pieces, one piece per processor. We use just 32 KByte on each processor to avoid conflicts with other users; of this we allocate 27000 bytes for the sieve.

To implement the sieving stage of NFS on the MasPar we could follow the same approach—i.e., the direct approach, with $A/X = 27000$ values of (a,b) per piece, and one piece per processor. However, we never attempted to implement this. Instead we tried to get the lattice sieve—special-q, with the plane method in the (c,e) plane—to work on the MasPar. Unfortunately we have not come up with a way to lay out the (c,e) plane over more than one processor without running into serious communications overhead.

So we chop the (c,e) plane into pieces that fit into one processor. In Pollard's terminology [13] we "sieve by rows": each processor considers one e for one (q,s) and uses its memory to sieve a line of c's with all the pairs (p,r) smaller than (q,s). Our MasPar sieving program, `pnfls`, handles 128 special (q,s) pairs at a time, one pair per row of processors. For each (q,s) batch it iterates through 128 e's from the range $\{e\}$ at a time, one e per column of processors.

6.7. *Sieving results.* The sieving program `pnfls` produces some "reports" of good (c,e) pairs, which (together with the corresponding (a,b) pairs and q's) are saved in a binary file `OUT`.

Our first version of `pnfls` used 10^4 values of c, namely $-5000 \le c < 5000$, and used 24 KByte per processor. We used counters of the form (x_r, x_a), initialized to $(0,0)$ and stored in two bytes. Denote the approximate logarithm base 2 of p by $l(p)$. To hit a counter with a prime p from `RFB` we added $(l(p),0)$ to the counter. To hit a counter with a prime pair (p,r) from `AFB` we added $(0,l(p))$ to the counter. We choose, by hand, minimum values for x_r and x_a; we throw out any (c,e) whose counter does not meet or exceed those minima. Pairs (c,e) that pass this test are subjected to more stringent tests, involving approximations to

$a - bm$ and $N(a,b)$. When all is said and done perhaps one out of every 7 or 8 reported sieve-smooth (a,b) pairs is actually prec-smooth (precisely smooth). (See Remark 6.8.) Hopefully we do not throw out more than a fraction of prec-smooth pairs.

In the current version of `pnfls` we use the same byte to represent x_r during the first sieve and x_a during the second one: we first sieve with the primes from `RFB`, remember the locations of x_r's where the counter is large enough before resetting the x_r's to zero, and next use the same sieve locations to sieve with the pairs from `AFB`. In this way we could double the length of the c interval and increase the yield per unit of time, without affecting the memory requirements of `pnfls`. With a c interval of $-10000 \le c < 10000$, `pnfls` fits in 24 KByte per processor; currently we are using $-14000 \le c < 14000$, which fits in 32 KByte.

6.8. *Remark.* In the above description we neglected to define various terms: "several pairs," "hitting," "read off," "most," "small," etc. The real object of the sieve stage is to limit the number of (a,b) pairs that have to be checked by trial division (next section). For trial division there is a quite precise definition of what "smoothness" (prec-smoothness) means: namely, $a - bm$ factors almost completely into primes smaller than max`RFB` with at most one "large prime" between max`RFB` and L, and similarly for $N(a,b)$ and max`AFB`. In sieving we can at best approximate this definition.

6.9. *Remark.* We choose the ordering of (p,r) values for the special-q method as suggested by Pollard [13]: all (p,r) from `AFB` are considered smaller than any $(p, m \bmod p)$ from `RFB`; the p's from `RFB` are placed into their natural order. Then the special q's are chosen from among the rational primes p. This means that the entire algebraic factor base is used at every step. This may not be optimal. It might be better to interleave the rational and algebraic factor bases, with all pairs (p,r) ordered simply by the value of p. We have not explored the practical import of this approach.

6.10. *Remark.* By definition, all special values of (q,s) come from `RFB` or `AFB`. It may happen that $a - bm$ has a large prime factor from outside `RFB`; this factor is not the special q! If we were to include large primes in our ordering of (p,r) values, then any pair with a large p would have to be considered smaller than any pair from `RFB` or `AFB` with respect to this ordering.

6.11. *Remark.* We generate short vectors in a given lattice as in [7, exercise 3.3.4–5]. Except where certain intermediate values are 0 modulo q we accept the resulting vectors as "short enough."

6.12. *Remark.* Initially we suspected that the special-q method would require larger factor bases to break even on the number of reports, even though the total time spent should be smaller. This would make the matrix reduction step slightly more difficult. This was, however, only a consequence of our initial choice of c interval, which was fairly short. The newer version of the program does not require substantially larger factor bases.

6.13. *Remark.* In this section we have taken $T_1, \ldots, T_6$ as positive constant multiples of some unspecified unit of time. In practice this is a reasonable assumption, but in some theoretical machine models these times are not constant. For example, in a bit-oriented model of computation, each T_i is between $\log \max P$ and $\log \max P \log^4 \log \max P$, if we use fast arithmetic for the basic steps. Our conclusion that special-q is asymptotically beneficial holds for any model where all the ratios T_i/T_j grow more slowly than $\log \max P$.

7. Trial dividing

We retain the notation of the previous section. The goal of the trial division stage is to produce (a, b) pairs, with a and b relatively prime integers that fit into a computer word, such that b is positive, $a - bm$ is rat-smooth, and $N(a, b) = f(a/b)b^d$ is alg-smooth—prec-smooth pairs, for short. Here $a - bm$ is rat-smooth if all its prime factors are smaller than $\max$**RFB**, except for at most one large prime factor between $\max$**RFB** and L. Similarly $N(a, b)$ is alg-smooth if all its prime factors are smaller than $\max$**AFB**, except for at most one large prime factor between $\max$**AFB** and L.

Our trial division program **abra** takes as input the reports **OUT** from the sieving stage. One hopes that many of the pairs (a, b) produced by the sieving stage have $a \equiv b(m \bmod p) \pmod p$ for several primes p, so that $a - bm$ is divisible by several primes p and thus has a good chance of being rat-smooth. (Note that $a - bm$ is always divisible by the special q, which is saved in **OUT**.) Similarly one hopes that $a \equiv br \pmod p$ for several algebraic pairs (p, r). Then

$$N(a, b) = f(a/b)b^d \equiv f(r)b^d \equiv 0 \pmod p$$

so that $N(a, b)$ is divisible by several primes p and thus has a good chance of being alg-smooth.

Our MasPar trial division program **abra** processes one pair (a, b) at a time, together with the special q value saved from sieving. If a and b are not coprime then (a, b) is thrown out immediately. Otherwise **abra** computes $(a - bm)/q$ and $N(a, b)$. It then invokes a MasPar routine to trial-divide $(a - bm)/q$ by all primes p in the rational factor base. (The prime 2 is not in the factor base for various reasons; it is trial-divided separately.) Unless $a - bm$ is rat-smooth **abra** goes on to the next (a, b) pair. If $a - bm$ factors properly, **abra** trial-divides $N(a, b)$. If $N(a, b)$ is alg-smooth then **abra** saves the prec-smooth pair (a, b) and its prime factorization in a readable file **ABRA**, together with some useful statistics from the sieving stage. Actually, to make disk space easier to manage, we split **ABRA** into a directory of individual files, each with at most 1000 prec-smooth (a, b) pairs.

Once **abra** is done processing (a, b) pairs we remove the output file **OUT**.

7.1. *Remark.* A prec-smooth pair is usually called an *ff*, *pf*, *fp*, or *pp* relation [9]; it is *pf* or *pp* iff $a - bm$ has a large prime factor, and it is *fp* or *pp* iff $N(a, b)$ has a large prime factor.

Our matrix construction program **newmat** reads the results **ABRA** of the trial division stage, removes duplicates, and produces a single large file **MAT** representing a matrix modulo 2 in binary format. It uses no other data except **QCB**.

The rows of the matrix are indexed by prec-smooth (a, b) pairs. A row contains the following bits:

(1) A bit always equal to 1.
(2) The log base -1 of the sign of $a - bm$. This bit is 1 in all our runs as $a - bm$ is always negative.
(3) For each prime p, $\operatorname{ord}_p(a - bm) \bmod 2$.
(4) For each prime p dividing c_d, $\operatorname{ord}_p N(a, b) \bmod 2$ if p divides b, 0 otherwise.
(5) For each pair (p, r) with $0 \le r < p$ and p prime, $\operatorname{ord}_p N(a, b) \bmod 2$ if p divides $a - br$, 0 otherwise.
(6) Finally, for each (q, s) in the quadratic character base, the log base -1 of the Legendre symbol $\left(\frac{a-bs}{q}\right)$.

Note that these rows are defined in terms of all primes p, not just those appearing in **RFB** and **AFB**. Of course we do not want to store infinitely long rows inside the computer. So instead of one bit for each prime p, we simply store a list of the primes that divide $a - bm$ to odd powers. We allow up to 18 primes in this list. Similarly we store a list of the primes p that divide $N(a, b)$ to odd powers, together with the corresponding values of r. We allow up to 19 pairs (p, r) in this list. We store the $\#\mathbf{QCB} = 100$ Legendre symbols and two extra bits packed into 14 bytes. Together with control information a row takes exactly 256 bytes to store in this format.

Most of the information in a row is easy to compute from **ABRA**. To compute the Legendre symbols we use a fast Jacobi symbol routine due to Peter Montgomery.

For the reasons behind these rows we refer to [3]. In the next steps (next section) of the algorithm we construct a set S of (a, b) such that the corresponding rows add up to 0 (mod 2). In this case we call S a dependency. If S is any set of (a, b) (with $\#S$ even) such that $\prod_{(a,b)\in S}(a - bm)$ and the element $\prod_{(a,b)\in S}(a - b\alpha)$ of our number field are both squares then S will, in fact, be a dependency. Conversely, we hope that if S is constructed as a dependency then $\prod_{(a,b)\in S}(a - bm)$ and $\prod_{(a,b)\in S}(a - b\alpha)$ will both be squares. If the quadratic character base is ridiculously large then this will necessarily be true. In practice a quadratic character base of size $\#\mathbf{QCB} = 50$ would probably suffice for all factorizations ever accomplished with the number field sieve in its present form; our choice of $\#\mathbf{QCB} = 100$ is almost certainly excessive.

8.1. *Remark.* Recall from Remark 5.1 that we might occasionally have a bad (q, s) column. For all we know hardware failures might corrupt several columns. It will no longer be true that if $\prod(a - bm)$ and $\prod(a - b\alpha)$ are squares then the corresponding rows of the computed matrix always add up to 0. But if the number of bad quadratic character columns does not exceed the number of excess

quadratic character columns, then any computed dependency S will necessarily produce squares. The only bad effect of such errors is that some fraction of the correct dependencies will be ignored during the matrix reduction stage. This justifies the first author's somewhat careless attitude towards the possibility of errors.

9. Reducing the matrix

The goal of this step is to produce several independent elements of the nullspace of the matrix constructed in the previous section. Note that the matrix is always taken modulo 2. We could have checked the validity of the rows at this point if we were worried about hardware errors.

We find cycles among the partial relations as described in [10]. This reduces the matrix to the set of rows that will be useful in producing dependencies. The useful matrix can be reduced in various ways, such as structured Gaussian elimination [8]. A problem for the future is that structured Gaussian elimination appears to take time cubic in the number of columns of the matrix, although the constant factor is very small. We tried the quadratic method of Wiedemann [15], but were not able to produce a sufficiently fast implementation for our purposes.

Our matrix reduction consists of two programs: **sgauss**, which runs on a workstation, to reduce the large but sparse matrix of *ff*'s and cycles to a smaller dense matrix, and **gauss**, which runs on the MasPar, to find dependencies in the dense matrix. The output of the matrix reduction stage is a file **SOLNS** containing some sets of pairs (a, b) such that the corresponding rows for each set add up to 0 modulo 2.

10. Computing the square root

We now have a file **SOLNS** containing one or more sets S of pairs (a, b) such that $\prod_{(a,b)}(a - bm)$ is an integer square and $\prod_{(a,b)}(a - b\alpha)$ is (we hope) a square in the number field. In the final stage of GNFS we convert each set S into a relation $x^2 \equiv y^2 \pmod{n}$. Heuristically at least one half of these pairs (x, y) will give rise to a nontrivial factor $\gcd(x - y, n)$ of n.

Our **cosqrt** program uses the ideas of Couveignes [4]. We prove in the next section that this procedure works. **cosqrt** begins by reading the inert primes ℓ in **IPB**. As we will see in the next section there is a lower bound on the size of **IPB**; we assume that this lower bound is met. Write $P = \prod \ell$. We next compute $(P/\ell) \bmod \ell$ directly for each ℓ, 16384 at a time, on the MasPar. (In this section, when we say "compute X mod M directly," when X is any sort of product, we mean "multiply the factors of the product X, one at a time, modulo M." We do not actually compute X and then reduce it modulo M.) Then we invert P/ℓ modulo ℓ and keep the result, k_ℓ. We also compute $P \bmod n$ directly for later use.

We read a set of pairs (a, b) that together form a dependency. For each of the inert primes ℓ we compute $\gamma_\ell = g'(\omega)^2 \prod_{(a,b)}(c_d a - b\omega) \mod \ell$ directly inside the number ring $\mathbf{Z}[\omega]/g(\omega)$. We also trial-divide each $a - bm$ and each $N(a, b)$ as in **abra** (see Section 7); we tally up the powers of each prime p dividing $a - bm$

and dividing $N(a,b)$. The tallies are forced to be even, so we end up with

$$\prod_{(a,b)}(a-bm)=\prod_p p^{2r_p} \qquad \text{and} \qquad \prod_{(a,b)}N(a,b)=\prod_p p^{2a_p}.$$

Next we compute $w=\prod_p p^{r_p} \bmod n$ directly, as well as

$$N_\ell=\operatorname{disc}(g)c_d^{(d-1)\#S/2}\prod_p p^{a_p} \bmod \ell$$

for each inert prime ℓ. We will show in the next section that each N_ℓ is nonzero. Next we set

$$\beta_\ell=\gamma_\ell^{1+\ell(1+\cdots+\ell^{d-2})/2}/N_\ell$$

inside the number ring modulo ℓ; here the division $/N_\ell$ means multiplication by the reciprocal of N_ℓ modulo ℓ.

Write $\beta_\ell=\sum_i b_{\ell,i}\omega^i$. We process each i from 0 through $d-1$ in turn. For each i we compute an approximation to

$$K_i=\sum_\ell b_{\ell,i}k_\ell/\ell$$

in floating-point arithmetic. Next we compute $Z_i=\operatorname{round}(K_i)$, the integer nearest K_i.

All further computations in **cosqrt** are modulo n. For each i we compute

$$S_i=\sum_\ell b_{\ell,i}k_\ell\frac{P}{\ell} \bmod n;$$

here $(P/\ell) \bmod n$ equals the product of $P \bmod n$ and the multiplicative inverse of $\ell \bmod n$. Finally we compute

$$x=c_d^{d-2+\#S/2}f'(m)w \bmod n$$

and

$$y=\sum_i (S_i-Z_iP)(c_dm)^i \bmod n.$$

We have $x^2\equiv y^2 \bmod n$, and we finish by checking whether $x-y$ has a factor in common with n. If n is not completely factored, **cosqrt** continues on to the next dependency from **SOLNS**.

10.1. *Remark.* It is the computation of β_ℓ that forces the degree d to be odd. See Section 13 for further comments.

10.2. *Remark.* Some of the ways in which f could conceivably be bad will show up at this stage, though they will rarely affect the factorization. It could turn out that $x\equiv y\equiv 0 \pmod n$. In this case, either $f'(m)$, c_d, or one of the primes p in **RFB** will have a factor in common with n.

10.3. *Remark.* If the parameters #QCB and #IPB are not chosen large enough then it may happen that $x^2 \not\equiv y^2 \pmod{n}$. In general (see next section) all values K_i should be extremely close to integers; if they are not then #IPB is too small.

10.4. *Remark.* We can compute N_ℓ without computing $\operatorname{disc}(g)$. For

$$\operatorname{disc}(g) \equiv (-1)^{d(d-1)/2} g'(\omega)^{(\ell^d-1)/(\ell-1)} \pmod{\ell}$$

if ℓ is any inert prime. This computation must be performed in the number ring, though the result will be an integer. Note that the factor $(-1)^{d(d-1)/2}$ can be omitted, if it is omitted for all ℓ consistently; for reasons explained in the next section, $\operatorname{disc}(g)$ need be computed only up to sign.

11. Proof that the square root works

In this section we verify the assertions of the previous section, along the lines of [4]. We retain all notation of the previous section.

We write $N(\delta)$ for the norm of an element δ of our number ring. We will need three properties of the norm: $N(\delta\delta') = N(\delta)N(\delta')$; $N(g'(\omega)) = \pm\operatorname{disc}(g)$; and $N(c_d a - b\omega)$ equals c_d^{d-1} times the norm polynomial $N(a,b)$ defined in Section 3.

Set $\gamma = g'(\omega)^2 \prod_{(a,b)}(c_d a - b\omega)$, so that $\gamma_\ell \equiv \gamma \pmod{\ell}$. If all has gone well γ is in fact a square inside the number ring. Define $\beta = \sum_i b_i \omega^i$ as the unique square root of γ such that $N(\beta)$ has the same sign as $\operatorname{disc}(g) c_d^{(d-1)\#S/2}$; here to ensure uniqueness we need the fact that d is odd. (This is what we call the positive square root method.) Now

$$\begin{aligned} N(\gamma) &= N(g'(\omega))^2 N\Big(\prod_{(a,b)}(c_d a - b\omega)\Big) \\ &= \operatorname{disc}(g)^2 c_d^{(d-1)\#S} \prod_{(a,b)} N(a,b) \\ &= \operatorname{disc}(g)^2 c_d^{(d-1)\#S} \prod_p p^{2a_p}. \end{aligned}$$

By choice of β we must have

$$N(\beta) = \operatorname{disc}(g) c_d^{(d-1)\#S/2} \prod_p p^{a_p}.$$

Thus $N_\ell \equiv N(\beta) \pmod{\ell}$. It is a pleasant fact of life that

$$N(\beta) \equiv \beta^{(\ell^d-1)/(\ell-1)} \pmod{\ell}.$$

We now work modulo ℓ:

$$\beta N_\ell \equiv \beta N(\beta) \equiv \beta\beta^{(\ell^d-1)/(\ell-1)} = \gamma^{1+\ell(1+\cdots+\ell^{d-2})/2} \equiv \gamma_\ell^{1+\ell(1+\cdots+\ell^{d-2})/2} \equiv \beta_\ell N_\ell.$$

We will show later that N_ℓ cannot be divisible by ℓ. Therefore $\beta \equiv \beta_\ell \pmod{\ell}$. Of course this means that $b_i \equiv b_{\ell,i} \pmod{\ell}$ for each i.

The remaining computations described in the previous section are an application of the Chinese remainder theorem. By construction $k_\ell P/\ell$ is congruent to 1 modulo ℓ and congruent to 0 modulo any $\ell' \neq \ell$. Define

$$R_i = \sum_\ell b_{\ell,i} k_\ell \frac{P}{\ell},$$

so that $S_i = R_i \bmod n$. Since $R_i \equiv b_{\ell,i} \pmod{\ell}$ for each ℓ, we must have $R_i \equiv b_i \pmod{P}$. Define $Z_i' = (R_i - b_i)/P$.

We now show that $Z_i' = Z_i$. Assume that the product $P = \prod \ell$ is large enough that $-P/2 < b_i < P/2$ for each i. We must choose #IPB large enough that this is satisfied. Now $\operatorname{round}(b_i/P) = 0$, so $\operatorname{round}(R_i/P) = Z_i'$. But by definition of R_i

$$\frac{R_i}{P} = \sum_\ell \frac{b_{\ell,i} k_\ell}{\ell} = K_i.$$

Therefore

$$Z_i' = \operatorname{round}(R_i/P) = \operatorname{round}(K_i) = Z_i$$

as desired.

So $b_i = R_i - Z_i' P = R_i - Z_i P \equiv S_i - Z_i P \pmod{n}$. We substitute this into the definition of y:

$$y \equiv \sum_i b_i (c_d m)^i \pmod{n}.$$

Define a homomorphism φ from $\mathbf{Z}[\omega]$ to $\mathbf{Z}/n$ by $\varphi(\omega) = c_d m$. Then

$$\varphi(g(\omega)) = g(c_d m) = f(m) c_d^{d-1} \equiv 0 \pmod{n}$$

so φ induces a homomorphism, which we also label φ, from our number ring $\mathbf{Z}[\omega]/g(\omega)$ to $\mathbf{Z}/n$. We compute modulo n:

$$\begin{aligned}
y^2 &\equiv \Big(\sum_i b_i (c_d m)^i\Big)^2 \equiv \Big(\varphi\Big(\sum_i b_i \omega^i\Big)\Big)^2 \\
&= \varphi(\beta)^2 = \varphi(\gamma) = \varphi\Big(g'(\omega)^2 \prod_{(a,b)} (c_d a - b\omega)\Big) \\
&= g'(c_d m)^2 \prod_{(a,b)} (c_d a - b c_d m) = (c_d^{d-2} f'(m))^2 c_d^{\#S} \prod_{(a,b)} (a - bm) \\
&= f'(m)^2 c_d^{\#S + 2(d-2)} \prod_p p^{2r_p} \\
&\equiv x^2.
\end{aligned}$$

Let us now prove the lemma that N_ℓ is nonzero modulo ℓ. By construction ℓ does not divide $\operatorname{disc}(g)$ or c_d. Furthermore, any prime p appearing in $\prod p^{a_p}$ must come from a pair (p, r) in AFB, so that f has a root modulo p. But f is irreducible modulo ℓ. So N_ℓ has no factors divisible by ℓ.

11.1. *Remark.* As we noted above `IPB` must be large enough that $P = \prod \ell$ is at least twice b_i in absolute value. Certainly the coefficients of γ are comparable to $(ca)^{\#S}$ where c reflects the coefficients of g and a reflects the values of a and b in our relations. Thus the coefficients of β are roughly comparable to $(ca)^{\#S/2}$. We want P to be larger than this. We choose `#IPB` loosely based on rough estimates of ca and the average value of ℓ. At worst x^2 and y^2 will not be congruent modulo n and we will try a larger inert prime base. Notice that `IPB` can be chosen smaller if the average ℓ is increased.

12. Examples

In this section we list some results obtained with our GNFS implementation, and we describe the resources used for these results.

We used GNFS to factor the 145-digit number $(2^{488}+1)/257$. Its prime factors, found at 14:50 EDT on 23 July 1992, are p_{49} and p_{97}, where

$$p_{49} = 1035\,81787\,79260\,14488\,58713\,38184\,91976\,75938\,90347\,64353$$

and

$$p_{97} = 30\,02073\,75742\,87773\,82273\,85792\,23855\,12797\,76379\,27232\,66417$$
$$65602\,50215\,27116\,98977\,99529\,50182\,55653\,75418\,50817.$$

As of August 1992 this was the record non-networked factorization of any difficult number, i.e., any number with no factors under 40 digits.

We proved p_{49} and p_{97} prime both with the Jacobi sum primality test implementation of Bosma and Van der Hulst [2] and with Morain's ECPP implementation of the elliptic-curve primality test [12]. To see if the $p \pm 1$ methods would have worked, we factored $p_{49} \pm 1$ and $p_{97} \pm 1$. It turns out that $p_{49} + 1$ has the large prime factor $(p_{49} + 1)/6$, $p_{49} - 1$ has the large prime factor

$$40\,69106\,49418\,15554\,84740\,76559,$$

$p_{97} + 1$ has the large prime factor $(p_{97} + 1)/474$, and $p_{97} - 1$ has a few large prime factors, including

$$p_{25} = 64969\,31962\,87009\,71159\,63241$$

and

$$p'_{25} = 10512\,71418\,58017\,46720\,49443.$$

Actually it was not entirely trivial to factor $p_{97} - 1$: after a few minutes of the elliptic curve method we were stuck with the 66-digit number

$$346838\,1582437308\,8881132906\,1576183794\,4797477331\,0360327973\,7735819099.$$

To complete this factorization with the elliptic curve method took about three

(without any tuning) GNFS, which was the first truly general GNFS factorization obtained with our implementation; QS succeeded in a few minutes, and GNFS succeeded in a few hours. The prime factors of this 66-digit number are 50781286063727873, p_{25}, and p'_{25}.

At 21:47 EDT on 15 September 1992 we found the prime factors of the 151-digit number $(2^{503}+1)/3$. They are p_{55} and a p_{97} different from the one above:

$$p_{55} = 20497\,44746\,26356\,86465\,84566\,17590\,83859\,07415\,01232\,90052\,98331$$

and

$$p_{97} = 42\,58599\,33875\,58828\,53705\,02226\,73947\,72292\,11842\,80651\,25200$$
$$53270\,05475\,15369\,56453\,88200\,68351\,21461\,75553\,45713.$$

It is a debatable point whether or not this factorization beats the record set by the factorization of the ninth Fermat number [8]: the composite factor $(2^{512}+1)/2424833$ of F_9 has only 148 digits, but was just as hard to factor as the 155-digit number $2^{512}+1$, whereas here we factored a 151-digit number for the price of the 152-digit number $2^{503}+1$.

We set an unequivocal new factoring record by factoring the 158-digit Mersenne number $2^{523}-1$. The prime factors, found at 19:30 EDT on 24 October 1992, are p_{69} and p_{90}, where

$$p_{69} = 1601\,88778\,31320\,21186\,10543\,68536\,88786\,88932$$
$$82870\,11365\,01444\,93221\,74680\,39063$$

and

$$p_{90} = 17141\,76918\,61249\,19812\,83170\,96534\,32211\,64761\,65056$$
$$71863\,03450\,94896\,62036\,78600\,06486\,97710\,18595\,04089.$$

We gratefully acknowledge Andrew Odlyzko's help with the sieving and trial division for this record factorization. He did about one third of the work on the MasPar at AT&T Bell Laboratories; the MasPar at Bellcore did the rest.

We proved p_{55}, p_{97}, p_{69}, and p_{90} prime with the Jacobi sum primality test implementation from [2].

In Table 1 our parameter choices and other details concerning these factorizations are reported. We did not include the 66-digit general factorization in the table, as we spent no time tuning it. (We ended up not needing any of the partial relations we generated!)

Our choices for `#QCB` and `#IPB` were not tuned at all. Both `RFB` and `IPB` use four bytes per prime, and `AFB` uses eight bytes per (p, r) pair. So for all three numbers these files take at most a few MByte of disk space.

For the first number our MasPar sieving program, `pnfls`, processed over $2 \cdot 10^8$ values of $(q, e, (p, r))$ per second, or over 12,000 values per processing element (PE) per second (where $2 \cdot 10^8 \approx 199947 \cdot 5888 \cdot 66161/(4.5 \cdot 24 \cdot 3600)$). This figure should be taken as somewhat variable because we sieve only up to q on the rational side, and because smaller values of p required more than one hit on the

TABLE 1. *Data on the factorizations*

n	$(2^{488}+1)/257$	$(2^{503}+1)/3$	$2^{523}-1$
# digits of n	148	151	158
f	X^5+4	$8X^5+1$	$8X^5-1$
m	2^{98}	2^{100}	2^{104}
max **RFB**	1300000	1300000	1600000
#**RFB**	100020	100020	121126
max **AFB**	1300000	1300000	1600000
#**AFB**	99827	99827	120909
#**QCB**	100	100	100
#**IPB**	163909	163909	196586
L	10^8	10^8	10^8
c interval	$[-5000, 5000)$	$[-5000, 5000)$	$[-14000, 14000)$
$\{e\}$	$[1, 5888]$	$[1, 7936]$	$[1, 12800]$
$\#Q$	66161	82037	≈ 65000
$\min Q$	≈ 400000	≈ 200000	≈ 700000
run time **pnfls**	4.5 days	6.5 days	14 days
space **pnfls**	24 KByte per PE	24 KByte per PE	32 KByte per PE
#**OUT**	$18 \cdot 10^6$	$55 \cdot 10^6$	$13 \cdot 10^7$
run time **abra**	1.7 days	3 days	7 days
space **abra**	4 KByte per PE	4 KByte per PE	4 KByte per PE
# *ff*'s	60253	51905	not kept
# *ffq*'s	32060	40999	not kept
# *pf*'s	367402	348074	not kept
# *pfq*'s	20573	59192	not kept
# *fp*'s	324296	289665	not kept
# *fpq*'s	141171	213419	not kept
# *pp*'s	1681923	1920259	not kept
# *ppq*'s	2057	239124	not kept
size **ABRA**	721 MByte	867 MByte	not kept
# distinct *ff*'s	65626	57728	> 70000
# distinct partials	2409258	2713637	$\approx 3.5 \cdot 10^6$
size **MAT**	624 MByte	709 MByte	895 MByte
total # cycles	142657	149813	> 210000
sparse matrix	208283×199947	207541×199947	280000×242135
(run time **sgauss**)	(16 hours)	(16 hours)	(2 days)
dense matrix	not kept	74100×73900	94000×93900
size dense matrix	not kept	685 MByte	1.1 GByte
run time **gauss**	not kept	2.1 hours	5 hours
$\#S$	not counted	≈ 257000	≈ 283000
run time **cosqrt**	7.5 hours	7.5 hours	9 hours
# **cosqrt** trials	3	5	3
total run time	7.5 days	12 days	22 days

c interval from (0.7), we used primes as low as 23. In any case, as each processor performs about $2 \cdot 10^5$ additions per second, it appears that we did the work of about 16 additions per $(q, e, (p, r))$. For the second number the sieving speed was comparable. The version of **pnfls** that we used for the first two numbers took 2 bytes per c. At some expense in run time we reduced this to 1 byte in the version of **pnfls** that we used for the third number. With slightly more space per PE, this allowed us to almost triple the c interval. As a result the sieving speed for the third number was somewhat slower, but the yield was much higher.

OUT, the binary output from sieving, uses 24 bytes per report. With well-tuned cutoff parameters there are not too many more reports than actual relations. For the first number we chose the cutoffs so as to eliminate many, perhaps most, of the *pp* relations. For the other two numbers the cutoffs were chosen more conservatively. This led to far fewer relations per report, but to more partials and cycles. For the last number this could have led to storage problems for **OUT**. We avoided this by sieving in batches of special q values, removing **OUT** after running **abra** for each batch, so that **OUT** never used more than eighty MByte of disk space at a time.

For the first number **abra** processed 7500 reports per minute. A newer version of **abra**, which was used for the other two numbers, achieved almost twice that speed. Of course, many of the reports were cut out after trial division by the primes of **RFB** and did not require division by the primes of **AFB**.

ABRA, the readable relations file resulting from **abra**, takes a lot of disk space. The "q" suffix of the *ff*'s, *pf*'s, *fp*'s, and *pp*'s (see Remark 7.1) means that the factorization of $a - bm$ ended up with a larger prime than q from **RFB**. In principle such relations have the "wrong" q and are not useful, as they should also be found with the "right" q. However, our sieving bounds were so high that many relations are not actually hit during sieving by their special q's; and our sieving bounds were so low that many *ff*'s and *fp*'s were caught by a lower q than their special q's. The fact that our sieving bounds were simultaneously so high and so low stems from the inherent inaccuracy of the logarithmic approximations we used; see Section 6. Another explanation for the occurrence of relations with the "wrong" q is that we searched somewhat different areas of the (a, b) plane for different q.

Because of the occurrence of *ffq*'s, *pfq*'s, *fpq*'s, and *ppq*'s, duplicate relations were found, which were removed by **newmat** before it constructed the quadratic characters. The run time of **newmat** was reasonably short, and is not given in Table 1. By far the most costly activity in this stage was I/O. The run time for **sgauss** is parenthesized because this was the only substantial step that was not carried out on a MasPar but on an ordinary workstation. We note that the dense matrix for the second number is larger than the dense matrix involved in factorization of the ninth Fermat number [8]. Its reduction produced considerably more than $74100 - 73900$ dependencies. For the last number the dense matrix did not fit in the MasPar memory (which is only 1 GByte), so we had to upgrade our in-core matrix eliminator to a more flexible but slower program that stores intermediate results on disk. Again we found many more dependencies than the

approximately 94000 – 93900 that we expected. This might be due to our very conservative choice of #QCB.

Our cosqrt is not inherently faster than the simple method stated in [3] but it is much more amenable to parallelization. The cosqrt run time in the Table refers to the processing time per dependency. Most of this time was used to compute γ_ℓ.

We also factored a 123-digit composite factor of $2^{491}+1$, as well as $(11^{131}-1)/2630$. Of course all these numbers could have been handled by SNFS, even though $\mathbf{Q}(\sqrt[5]{11})$, the field of choice for $(11^{131}-1)/2630$, has class number 5, as reported by R. D. Silverman.

13. Future directions

The crucial problem in GNFS is finding a good polynomial. We cannot overestimate the practical importance of searching for a polynomial a few digits better than random. According to theoretical estimates [3] our polynomials are nowhere near optimal. As polynomials improve, GNFS will become more and more competitive with QS; a much better polynomial-finding method could make GNFS the leader for general 100-digit numbers.

As indicated in Section 4, without m_2 from [3] we can in theory reduce the coefficients by several digits. With m_2 we can in theory reduce the coefficients by twice as many digits. So we should incorporate m_2 into our GNFS implementation as polynomial-searching methods improve.

The restriction that d must be odd is not at all helpful, as $d=4$ appears to be a very good choice for numbers around 100 digits. If d is even, is there a way to uniquely specify β, analogous to the requirement that $N(\beta)$ have a certain sign, that can be tested easily modulo any ℓ? Or is there a fast way to determine the two values of $\beta \bmod \ell\ell'$? If either of these problems is solved then we can use even degrees. Of course, the method in [3] is reasonably fast, given a fast multidigit multiplication routine; but it does not seem to parallelize easily.

Don Coppersmith has suggested a method that will work for even degrees at the expense of some arithmetic on huge numbers (much less than in [3] for $d>1$). We illustrate it for $d=4$ under the assumption that (say) b_1 is not divisible by n. Compute γ in the number ring modulo each ℓ. By standard methods compute some square root β_ℓ of γ in each of the finite fields $(\mathbf{Z}/\ell\mathbf{Z})[\omega]/g(\omega)$. The square roots need not be consistent over different ℓ, but the products $b_{\ell,j}b_{\ell,1}$ for $j=0,1,2,3$ are consistent. So we combine these products for $j\neq 1$ into values b_jb_1 modulo n as in Section 11. Using the standard Chinese remainder theorem, with arithmetic on huge numbers, we combine the squares $b_{\ell,1}^2$ into a value b_1^2. We then choose one of the two square roots b_1 and have enough information to put together a consistent $b_j \bmod n$ for $j=0,1,2,3$ as desired. It remains to be seen how much work this method will require in practice.

We have not yet incorporated certain practical improvements. There are a significant number of "free relations" [9]; we should use them. We judge a special q as productive if it is not in the bottom fraction of RFB, as per [13]; we should pay more attention to the reduced lattice vectors. We should experiment with

interleaving the factor bases as suggested in Remark 6.9. We should compute the actual number of quadratic characters needed in various real examples, then use only a few more than the maximum, rather than 100. We should use (p, ∞) pairs in the sieving when possible.

What can be said about the sizes of the coefficients b_i? It would be nice to have accurate estimates—taking the distribution of (a, b) into account—so that we could select `#IPB` sensibly. We have certainly wasted time on the MasPar processing excessively many values of ℓ.

Our parameters are not chosen optimally. In particular the algebraic factor base should probably be much larger than the rational factor base for best results. It would help greatly if we could quickly and reliably estimate, to within a few percent, the number of relations that will be produced by the sieving, given `POLY` and all relevant parameters.

It is tempting to try to construct pairs (a, b) with a high likelihood of being prec-smooth. For instance, if a has a large factor in common with m, then $a - bm$ will also have that factor (this is equivalent to dividing b by the factor, as suggested by R. D. Silverman). Unfortunately it appears that $N(a, b)$ becomes too big. Similarly if a/b is extremely close to a root of f then $N(a, b)$ will be small, but then $a - bm$ will not remain small enough. It is not clear whether considerations like these are the path to an improved number field sieve or merely minor curiosities.

References

1. L. M. Adleman, *Factoring numbers using singular integers*, Proc. 23rd Annual ACM Symp. on Theory of Computing (STOC), New Orleans, May 6–8, 1991, 64–71.
2. W. Bosma, M.-P. van der Hulst, *Primality proving with cyclotomy*, Universiteit van Amsterdam, 1990.
3. J. P. Buhler, H. W. Lenstra, Jr., C. Pomerance, *Factoring integers with the number field sieve*, this volume, pp. 50–94.
4. J.-M. Couveignes, *Computing a square root for the number field sieve*, this volume, pp. 95–102.
5. J. A. Davis, D. B. Holdridge, *Factorization using the quadratic sieve algorithm*, Tech. Report SAND 83-1346, Sandia National Laboratories, Albuquerque, New Mexico, 1983.
6. B. Dixon, A. K. Lenstra, *Factoring integers using SIMD sieves*, Advances in Cryptology, Eurocrypt '93, to appear.
7. D. E. Knuth, *The art of computer programming*, volume 2, *Seminumerical algorithms*, second edition, Addison-Wesley, Reading, Massachusetts, 1981.
8. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard, *The factorization of the ninth Fermat number*, Math. Comp. **61** (1993), to appear.
9. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard, *The number field sieve*, this volume, pp. 11–42.
10. A. K. Lenstra, M. S. Manasse, *Factoring with two large primes*, Math. Comp., to appear.
11. H. W. Lenstra, Jr., *Factoring integers with elliptic curves*, Ann. of Math. **126** (1987), 649–673.
12. F. Morain, *Implementation of the Goldwasser-Kilian-Atkin primality testing algorithm*, INRIA report 911, INRIA-Rocquencourt, 1988.
13. J. M. Pollard, *The lattice sieve*, this volume, pp. 43–49.
14. C. Pomerance, *The quadratic sieve factoring algorithm*, Lecture Notes in Comput. Sci. **209** (1985), 169–182.

15. D. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory **32** (1986), 54–62.

5 Brewster Lane, Bellport, NY 11713, U.S.A.
E-mail address: brnstnd@nyu.edu

Room MRE-2Q334, Bellcore, 445 South Street, Morristown, NJ 07960, U.S.A.
E-mail address: lenstra@bellcore.com