Lattice sieving and trial division

Roger A. Golliver[1], Arjen K. Lenstra[2], Kevin S. McCurley[3]

[1] Intel SSD, MS CO1-05, 15201 NW Greenbrier Parkway,
Beaverton, OR 97006, U.S.A
E-mail: roger@ssd.intel.com
[2] MRE-2Q334, Bellcore, 445 South Street,
Morristown, NJ 07960, U.S.A
E-mail: lenstra@bellcore.com
[3] Organization 1423, Sandia National Laboratories,
Albuquerque, NM 87185-1110, U.S.A
E-mail: mccurley@cs.sandia.gov

Abstract. This is a report on work in progress on our new implementation of the relation collection stage of the general number field sieve integer factoring algorithm. Our experiments indicate that we have achieved a substantial speed-up compared to other implementations that are reported in the literature. The main improvements are a new lattice sieving technique and a trial division method that is based on lattice sieving in a hash table. This also allows us to collect triple and quadruple large prime relations in an efficient manner. Furthermore we show how the computation can efficiently be shared among multiple processors in a high-band-width environment.

1 Introduction

Throughout this paper we assume that the reader is familiar with the number field sieve (NFS) integer factoring algorithm [8]. We restrict ourselves to the relation collection stage of NFS.

Let $n > 1$ be an odd integer which is not a prime power, and let m be an integer such that $f(m) \equiv 0 \bmod n$ for an irreducible polynomial $f \in \mathbf{Z}[X]$ of degree $d > 1$. We do not discuss how m and f might be found. In the relation collection stage of NFS we attempt to find sufficiently many *relations*, i.e., pairs (a, b), with $b > 0$, satisfying the following conditions (cf. [9: 2.8, 7.3]):

(1.1) $\gcd(a, b) = 1$,

(1.2) on the *rational side* the number $a - bm$ is B_1-smooth, except for at most two additional prime factors which should be $< B_3$, for some *rational smoothness bound* B_1 and *rational large prime bound* $B_3 < B_1^2$,

(1.3) on the *algebraic side* $N(a, b) = b^d f(a/b)$ is B_2-smooth, except for at most two additional prime factors which should be $< B_4$, for some *algebraic smoothness bound* B_2 and *algebraic large prime bound* $B_4 < B_2^2$,

where, as usual, an integer is called x-smooth is its prime factors are $< x$.

Notice that we allow two large primes on either side, unlike [9] or [2] where at most one large prime per side was used, and unlike [3] where at most one large

prime was used on the rational side. One of the advantages of our new implementation is that it finds these *double large prime relations* without the loss of efficiency that was reported in [9: 7.3] or that would follow from a straightforward extension of the method from [2]. Another advantage is that it is substantially faster than implementations previously reported in the literature (cf. [2, 3, 9]), even though it considers far more reports (see below) and produces far more relations; of course it is even faster without the double large primes, but then it produces just the same (and much smaller) amount of relations as were found in [9] and [2].

In [9] the traditional method was used in the relation collection stage: for $b = 1, 2, \ldots$ in succession a large interval of a-values is *sieved* to find pairs (a, b) for which both $a - bm$ and $N(a, b)$ are likely to be smooth, and these candidates are tested for actual smoothness using *trial division*, until sufficiently many relations have been found. In [12] Pollard suggested a faster sieving method: for a large enough collection of medium sized primes $q < B_1$ sieve in the sublattice L_q of the (a, b)-plane consisting of the pairs (a, b) for which q divides $a - bm$, where the sieve (and the trial division) on the rational side can be restricted to the primes $< q$. Pollard describes two ways to do the sieving in L_q: *sieving by rows*, which requires only a small amount of memory but which is inefficient for large primes, and *sieving by vectors*, which is much faster if a large amount of memory is available. In [2] the first method is used, because of the small processors used there. In [3] a more efficient variation of the first method was used, also because not enough memory was available to make an efficient implementation of the second method possible (cf. [14]).

Our implementation makes use of the second method, for the sieving *and* for the trial division, though our q's are medium sized primes on the algebraic rather than the rational side. So, our L_q is the sublattice of the (a, b)-plane consisting of the pairs (a, b) for which q divides $N(a, b)$. The speed-up that we achieve depends for the sieving on the amount of memory that is available, but our trial division is an order of magnitude faster than previous methods, almost irrespective of the available memory. Our sieving implementation is explained in Section 2, the trial division in Section 3.

Our relation collection program can be executed on any multiple-instruction multiple data machine that has enough memory per node, by assigning different q's to different nodes. Our program can also be executed on two smaller nodes simultaneously if a high-bandwidth communication between the two nodes exists. Various parallelization issues are discussed in Section 4.

Relations with one or two large primes can be combined in the usual way to turn them into useful relations (cf. [9: 2.10]). The algorithm to find these combinations can be generalized to find most combinations among relations with at most three large primes, as shown in [4]. This approach gets stuck, however, at four large primes. Satisfactory methods to find all combinations among relations with any (small) number of large primes are described in [5]. As shown in [5] the relations with four large primes give rise to an unusually large number of combinations, when combined with relations with fewer large primes. This could lead

to considerable savings in the relation collection stage compared to approaches that use fewer than four large primes. It could also lead to substantially more work in the final stages of NFS, the matrix reduction and the square root computation. It remains to investigate how these effects can best be balanced. We refer to [5] for a further discussion of this point.

In Section 5 we give the run times and yields of our relation collection program for various q's for a 129-digit n, and we derive some estimates how long the complete relation collection might take.

2 Sieving

Let the notation be as above, and let P be the set of all pairs (p,r) with p a prime $< B_1$ and $r \equiv m \bmod p$ with $0 \le r < p$. Similarly, let Q be the set of all pairs (p,r) with p a prime $< B_2$ and $f(r) \equiv 0 \bmod p$ with $0 \le r < p$. Notice that a particular prime can occur up to d times in Q because f may have d distinct roots modulo p. Assume that the elements of Q are ordered in some well-defined way such that $(p_1,r_1) < (p_2,r_2)$ if $p_1 < p_2$. In practice $\#Q$ will be close to $\pi(B_2)$, the number of primes $\le B_2$. The sets P and Q are called the *rational* and the *algebraic factor base*, respectively.

For any $u = (p,r) \in P \cup Q$ let L_u be the lattice generated by the vectors $(p,0)$, $(r,1)$ in the (a,b)-plane. Notice that L_u corresponds to the pairs (a,b) for which p divides $a - bm$ for $u \in P$, and for which p divides $N(a,b)$ and $r = a/b \bmod p$ for $u \in Q$.

2.1. *Preparing q.* Throughout this section and the next we fix a pair $(q,s) \in Q$ with $q > B_0$, for some $B_0 < B_2$. Abbreviate $L_{(q,s)}$ to L_q. We apply a straightforward lattice reduction method to the basis $(q,0)$, $(s,1)$ of L_q in the (a,b)-plane, and find two short vectors $V_1 = (a_1,b_1)$, $V_2 = (a_2,b_2)$ generating L_q, such that the Euclidean length $||V_1||$ of V_1 is $\ge ||V_2||$. Each point of L_q can be written as $c \cdot V_1 + e \cdot V_2 \in \mathbf{Z}^2$, for $c, e \in \mathbf{Z}$, and each point in the (c,e)-plane corresponds to a pair (a,b) for which q divides $N(a,b)$ and $s = a/b \bmod q$:

$$(a,b) = (c \cdot a_1 + e \cdot a_2, c \cdot b_1 + e \cdot b_2). \tag{2.2}$$

Sieving will take place in the (c,e)-plane. Let $Q_q = \{u : u \in Q, u < (q,s)\}$.

2.3. *Preparing the primes.* Let D be a crossover value between 'small' and 'large' primes. Its value depends on n, q and the implementation (cf. Section 5). Define for all $u = (p,r) \in P \cup Q_q$ the sublattice L_{qu} of L_q as $L_q \cap L_u$. A basis for L_{qu} in the (c,e)-plane is given by

$$\bar{U}_{u1} = (p,0), \qquad \bar{U}_{u2} = \left(\frac{a_2 - r \cdot b_2}{r \cdot b_1 - a_1} \bmod p, 1\right).$$

For all u's for which this basis is well-defined, $\bar{U}_{u2} \not\equiv (0,1) \bmod p$, and $p \ge D$, we compute a reduced basis U_{u1}, U_{u2} for L_{qu} in the (c,e)-plane, with $||U_{u1}|| \ge ||U_{u2}||$; for all other u we call u *exceptional* and set both U_{u1} and U_{u2} equal to $\bar{U}_{u2}$ (but see Remarks 2.4 and 2.5).

2.4. *Remark.* In our implementation we have a total of 65 bits available to store U_{u1} and U_{u2}: one parity bit, two bits for signs, and 2×17 and 2×14 bits for the absolute values of the entries of U_{u1} and U_{u2}, respectively (cf. Remark 2.12). If the basis U_{u1}, U_{u2} for a certain u does not fit in this format we declare u to be exceptional and treat it as such. This happens only occasionally for the B_1 and B_2 that we have been using; see Section 5 for examples. For exceptional u's the vectors U_{u1} and U_{u2} can easily be stored in 65 bits.

2.5. *Remark.* In the above initialization of U_{u1} and U_{u2} for the exceptional u's we assume that the smallest e-value to be sieved equals 1. If the first e-value to be sieved equals e_1, then U_{u1} should be initialized as $(e_1 \cdot \tilde{U}_{u2}) \bmod p$.

2.6. *Partitioning the sieve.* Suppose that we want to sieve the (c,e)-plane for $|c| \le C$ and $0 < e \le E$, where C and E depend on n; see Section 5 for examples. Because in general the required $(2C+1)\cdot E$ sieve-locations do not fit in memory, we partition the sieve into t pieces $S_1, S_2, \ldots, S_t$, with $S_i = \{(c,e) : |c| \le C,\ E_{i-1} < e \le E_i\}$ for appropriately chosen $0 = E_0 < E_1 < \ldots < E_t = E$, and sieve over S_i for $i = 1,2,\ldots,t$ in succession as described in 2.7, 2.8, and 2.9 (cf. Remark 2.13). See Section 5 for examples of $\#S_i$ used in practice.

2.7. *The algebraic sieve.* To find the pairs $(c,e) \in S_i$ for which the corresponding $N(a,b)/q$ is likely to be q-smooth we first set all $\mathtt{S}[c,e]$ to zero, where $\mathtt{S}$ is an array with indices $(c,e) \in S_i$ (but see Remark 2.10 and 2.13). Next we do the following for all $u = (p,r) \in Q_q$.

For the exceptional u's we sieve 'by rows': for $e = E_{i-1}+1, E_{i-1}+2, \ldots, E_i$ in succession first replace $\mathtt{S}[c,e]$ by $\mathtt{S}[c,e] + [0.5 + \log p]$ for all c with $|c| \le C$ that are modulo p equal to the first coordinate of U_{u1}, and next replace U_{u1} by $(U_{u1}+U_{u2}) \bmod p$. Notice that the u with $p \le 2C+1$ should be treated differently from the u with larger p For exceptional u's with *very* small p sieving by rows is quite slow. In practice the small p are therefore replaced by appropriately chosen powers (and the roots are changed accordingly).

For the non-exceptional u's we sieve 'by vectors': for all $\mathbf{Z}$-linear combinations (c,e) of U_{u1} and U_{u2} with $(c,e) \in S_i$ replace $\mathtt{S}[c,e]$ by $\mathtt{S}[c,e] + [0.5 + \log p]$. The appropriate linear combinations can be found efficiently by considering the relevant inequalities involving U_{u1}, U_{u2}, E_{i-1}, E_i, and C; we do not elaborate.

The vectors U_{u1} and U_{u2} remain unchanged in this step, unlike the vectors for the exceptional u's which get updated from one e-row to the next. So, for the non-exceptional u's the S_i can be processed in any order, but for the exceptional u's they have to be processed in order, unless U_{u1} is given the correct initial value for each S_i, using Remark 2.5. Obviously this step takes at least a constant amount of time per u; it follows that $\#\mathtt{S}$ should not be too small compared to p to make 'sieving by vectors' efficient.

2.8. *Checking sieve locations and rational sieve.* After sieving with all $u \in Q_q$, we inspect all values stored in $\mathtt{S}$ (cf. Remark 2.10), remember the coprime pairs (c,e) and $\mathtt{S}[c,e]$'s for which the latter is larger than some fixed *algebraic report bound* and replace those $\mathtt{S}[c,e]$'s by zero. Next we sieve with all $u \in P$, as described in 2.7 (but with the obvious changes), to find the $(c,e) \in S_i$ for which the corresponding $a - bm$'s are likely to be B_1-smooth.

2.9. *Collecting reports.* Finally we re-inspect the same pairs and S-values that were remembered after the algebraic sieve, and keep the *reports*: the coprime (a, b) for which the corresponding $\mathtt{S}[c, e]$ (cf. (2.2)) is larger than some fixed *rational report bound*, for which $\mathtt{S}[c, e]$ is sufficiently close to a floating point approximation of $\log(a - bm)$, and for which the value of the first sieve is sufficiently close to a floating point approximation of $\log(N(a, b)/q)$. Notice that these tests get successively more discriminating and more expensive. The resulting pairs (a, b) have a good chance to satisfy (1.2) and (1.3) if the report bounds are set correctly. See Section 5 for examples of report bounds, of numbers of pairs to be remembered, and of numbers of reports.

After $S_1, S_2, \ldots, S_t$ have been sieved all reports (a, b) have to be inspected, i.e., the corresponding $a - bm$ and $N(a, b)/q$ have to be trial divided (but see Remark 3.4). Our implementation of the trial division is described in Section 3.

2.10. *Remark.* As usual we use single bytes to represent the $\mathtt{S}[c, e]$. Instead of initializing the $\mathtt{S}[c, e]$ as zero for the first sieve and testing $\mathtt{S}[c, e] \geq x$ for some bound x, we initialize the $\mathtt{S}[c, e]$ as $-x$ and test $\mathtt{S}[c, e]$ for non-negativity. On most architectures this can be done several bytes at a time, which is often much faster. The initialization of S can be speeded up similarly.

2.11. *Remark.* Because for general n the number $N(a, b)/q$ can be expected to be substantially larger than $a - bm$ we sieve over the algebraic side before we sieve over the rational side. This minimizes the number of pairs and values to be remembered after the first sieve. For special n (where $a - bm$ is often larger than $N(a, b)/q$) we change the order of sieving.

2.12. *Remark.* Per $u = (p, r) \in P \cup Q_q$ we use 16 consecutive bytes, i.e., 128 bits, of storage: one byte for the difference (divided by 2) with the previous p, three bytes for r, a floating point approximation of $1/p$ in 31 bits, and 65 bits for the vectors U_{u1} and U_{u2} (cf. Remark 2.4). The approximation of $1/p$ is used to speed up the computations modulo p in the sieving and in the trial division. Evidently, this limits the general applicability of our implementation, but for n well beyond our current range of interest this format suffices.

This dense data structure was chosen for two reasons. First, by having the factor base's C data structure size equal to a small power of two, we can be sure that the compiler will efficiently map the structure in memory, avoiding misaligned data access penalties and wasted memory. Secondly, since the factor base is accessed sequentially, having a single large array for the factor base, requires only one active TLB entry for accessing the structure, thus freeing more TLB entries to map the large sieve array.

2.13. *Remark.* Because only the $u \in P \cup Q_q$ have to be stored during the sieving and trial division for a certain q, the value of t increases with q, without affecting the total memory used, i.e., the smaller q, the more memory can be devoted to S. During the sieving the (c, e) for which both c and e are even can be avoided at no extra cost. Because these points are not needed (cf. (1.1)) we do not include them in S. This slightly affects the description given above.

3 Trial division

The straightforward approach to process the reports (a, b) is to apply trial division successively to each $a - bm$ with the elements from the rational factor base and, if necessary, to each $N(a, b)/q$ with the elements $\leq (q, s)$ from the algebraic factor base. Because up to two additional primes are allowed per side, these trial divisions might be followed, per report, by at most two applications of for instance, Shanks's 'squfof' to factor the remaining composite cofactors.

In [9] and [2] at most one additional prime factor per side is allowed in the relations. Consequently there are far fewer reports, and there are no remaining composites to be factored after the trial divisions. At most two trial divisions per report works thus reasonably well, although the trial division time becomes a substantial fraction of the time spent on the relation collection stage (varying from a quarter to a third, according to [2: Table 1]).

There are several ways to make the trial divisions go faster. Obviously, it is always better to check that $a \equiv br \bmod p$ for $(p, r) \in P \cup Q$, before the actual trial division on $a - bm$ or $N(a, b)/q$ is carried out. We can also keep track of the sum of the logarithms of the divisors found, compare this to the sieve values, and try to use this (cheap) information to jump ahead in the factor bases; this saves some time. We could use *early aborts*: give up on reports for which not enough divisors have been found after processing certain fixed fractions of the factor bases. This also saves some time, but some relations get lost which is something we try to avoid in NFS (unlike Quadratic Sieve).

Because we have substantially more reports than [9] and [2], but also than [3], none of these approaches works satisfactorily for us. Therefore we decided to do the trial division simultaneously for all reports per (q, s) by sieving in a hash table. The pairs (c, e) corresponding to the reports (a, b) are the hash keys, and for each key the hash table H has storage for the key, a counter $\mathtt{C}_{(c,e)}$, and an array $\mathtt{A}_{(c,e)}$ of, say, 50 integers (cf. Remark 3.4). We also use flags $\mathtt{F}_c$ and $\mathtt{F}_e$ for $|c| \leq C$ and $0 < e \leq E$ (cf. 2.6). We use the space allocated for S to store H, the $\mathtt{F}_c$'s, and the $\mathtt{F}_e$'s.

3.1. *Rational trial sieving.* Initially we set all flags to 0. For all reports (a, b) we store the corresponding (c, e) in H, we set $\mathtt{C}_{(c,e)}$ to -1 and set $\mathtt{F}_c$ and $\mathtt{F}_e$ to 1. For all non-exceptional $u = (p, r) \in P$ we do the following: for all $\mathbf{Z}$-linear combinations (c, e) of U_{u1} and U_{u2} for which $\mathtt{F}_c$ and $\mathtt{F}_e$ are both equal to 1 and for which (c, e) is in H, we increase $\mathtt{C}_{(c,e)}$ by 1, and we store p in $\mathtt{A}_{(c,e)}[\mathtt{C}_{(c,e)}]$. This is similar to what we did in 2.7, except that we now remember the primes instead of simply accumulating the logarithms of the primes, and that we process all S_i for $1 \leq i \leq t$ simultaneously (but see Remark 3.4).

3.2. *Rational factorization.* For all reports (a, b) we remove all factors p from $a - bm$, first for the primes p stored in $\mathtt{A}_{(c,e)}[0]$ through $\mathtt{A}_{(c,e)}[\mathtt{C}_{(c,e)}]$, and next for all exceptional $u = (p, r) \in P$. If the remaining cofactor of $a - bm$ is either $\geq B_3^2$, or both $\geq B_3$ and $< B_1^2$, or not easily proved to be composite (cf. [7]), then (a, b) is removed from the list of reports. We keep the reports (a, b) for

which we have found the complete factorization of $a - bm$, except for at most one factor which is either prime and $< B_3$, or composite and $< B_3^2$.

3.3. *Algebraic trial sieving and factorization.* For the remaining reports we repeat 3.1 with the $u \in Q_q$ instead of the $u \in P$. Next we remove all factors p from $N(a,b)/q$, by retrieving the p's stored in $\mathtt{A}_{(c,e)}$ followed by trial division with the p's for which $u = (p,r) \in Q_q$ is exceptional. If the remaining cofactor is 1, prime and $< B_4$, or composite and $< B_4^2$, then we use Shanks's 'squfof' (followed by the elliptic curve method (cf. [7]) if 'squfof' did not work) to factor both this cofactor and the cofactor of $a - bm$, if needed, and report the resulting relation if it satisfies (1.2) and (1.3). Notice that $N(a,b)$ might have one or two prime factors p with $q < p < B_2$ (cf. [2: Section 12]); these (a,b) are reported as well, but since they might also be found for a larger q (i.e., one of those p's) we have to remove duplicate relations before processing them.

3.4. *Remark.* Because efficient look-up in H is essential, we assume that the number of reports is small compared to the number of locations #H in H. After processing a piece of the (c,e)-plane (i.e., after each completion of 2.9), we therefore check if the total number of reports found so far exceeds $\#\mathtt{H}/x$, for some small x (2 or 3, say), and if that is the case we perform 3.1 to process them. Of course, we also perform 3.1 after the sieving for a particular (q,s) is completed if new reports have been found after the last execution of 3.1.

3.5. *Remark.* Because sieving with the exceptional u's is relatively slow, and because there are typically only a few thousand of them, we do not resieve with them, but find the corresponding primes using root checking and trial division.

4 Parallelization issues

The code was originally developed to run as a single UNIX®* process. That version can be executed in an "embarrassingly parallel" fashion by assigning disjoint sets of q's to different processes and collecting the relations. We wanted to run this code on the Intel Paragon [6] at Sandia National Laboratories. This machine currently has 528 nodes with 32MB of memory and 1312 nodes with 16MB of memory, for a total of 1840 computational nodes and 37GB of memory. Other nodes in the machine are responsible for providing interactive user and I/O services. The nodes are all connected by a high speed/low latency communication network as a 16×120 mesh. Each node has two i860XP processors, although the present software dedicates one of the processors to handling communication (in the future we plan to utilize the second processor for computation). We chose to use the SUNMOS operating system** [11] on the compute nodes rather than the standard OSF operating system. The reasons for this are:

- SUNMOS consumes only 240K of physical memory on a node, whereas OSF consumes approximately 5 megabytes,

* UNIX is a registered trademark of UNIX Software Laboratories, Inc.

** SUNMOS stands for Sandia UNM Operating System, named after the organizations that collaborated on its development.

- SUNMOS offers message passing bandwidth of 170 megabytes per second, whereas OSF currently peaks at about 35 megabytes per second,
- SUNMOS user 4 megabyte physical memory pages for it's memory addressing, while OSF uses 4 kilobyte pages. The larger memory page size allows yields approximately 35% faster speed in the sieving code due to more efficient use of TLB entries.

Experiments are currently underway with SUNMOS to allow the second node processor to be used for computation.

4.1. *Large nodes.* Our first step at parallelization for the Paragon followed the "embarrassingly parallel" UNIX model. Each 32MB Paragon node ran one process which was given a distinct set of q's. To optimize the performance the only change required was to reduce the startup I/O by having a single root node read the factor base data from disk and broadcast the data to the other nodes over the communication network. The problem of load balancing was resolved by assigning each node a number of q's per job. Each node appended the relations it found to a file which is named to help the management of sieving over a large number of q's and possibly a number of (c, e) ranges.

4.2. *Small nodes.* We wanted to be able to use the entire Sandia Paragon, but the majority of nodes in the machine have only 16MB of memory, and this is too small to hold the factor base and the sieve array in physical memory. Paging would extract an extremely heavy performance penalty, so we overcame this problem with a second level of parallelization. The 16MB nodes were treated as even and odd node pairs. The factor base was distributed between the node pairs. The even node got all the even indexed factor base elements, and the odd processor got all the odd ones, with the obvious implications for the 'differences' from (2.12). This distribution of the factor bases was chosen over others, because it had better load balancing properties for the division of work between the node pairs. All other data structures including the sieve array were duplicated between the node pair.

The lattice sieving and trial division process is done as described in Sections 2 and 3. The sieving is done in parallel with the partitioned factor base, and the node pairs use the high speed communication network to combine and exchange the sieving results. Both nodes in the pair duplicate the initialization steps. The algebraic sieving is done in parallel by each node, using their own portion of the factor base, after which the node pair synchronizes and combine their sieve results. This communication step is the most intensive of the parallelization, since the entire sieve array is exchanged between the pair of nodes.

Each node examines the sieve array for the interesting locations as in 2.8. These are re-initialized and each node sieves over their own half of the rational factor base. The node pair then synchronizes again and exchanges the values of the interesting sieve locations remembered after the algebraic sieve. The nodes in the pair each examine disjoint parts of the possible reports and save those which pass the screening described in 2.9. The nodes then synchronize and exchange the reports which will be saved for the trial division stage.

The trial division sieving parallelized as follows. First each node sieves with their half of the rational factor base as in 3.1. For each report the node pairs exchange and remove the prime factors thus found, after which they both remove their exceptional primes from the remaining cofactors and exchange these factors too. The even node then checks to see if the report might satisfy condition (1.2) and should be checked for condition (1.3) (i.e., no attempt is made yet to factor a remaining composite cofactor $\geq B_1^2$ and $< B_3^2$). Algebraic trial sieving is parallelized in a similar way to process the remaining reports.

The node pair parallelization required the modification of about 100 lines of the 4000 line sieve program, and the inclusion of about 800 lines for the data partitioning, synchronization, and communications operations. Our experiments using the same size sieve array, show that the time to sieve over an equivalent $q \times (c, e)$ range on a 32MB node vs. a 16MB node pair is about 1.8 to 1. Thus this second level of parallelization allows us to use the 16MB nodes with only a small loss of efficiency and greatly increasing the number of nodes which can be used. It should be noted that this level of parallelization to use small memory nodes heavily depends on the existence of a high-bandwidth, reliable network between the nodes.

5 Run times and parameter choices

At the time of writing the relation collection stage of a factoring attempt of n=RSA-129 (cf. [13]) has just been completed on the Internet, using the quadratic sieve method (QS) and the familiar electronic mail approach [10]. This computation took approximately 5000 to 6000 MIPS years [1]. We list some timings and parameter choices for our program applied to the same n and running on a Sparc-10 workstation, estimated as 33 MIPS. Our numbers suggest that relation collection for NFS for this n could be completed in a fraction of the time needed by QS.

We use $m = 8000000099160814875591$ from which f follows by writing n in the symmetric base m; these m and f were found by James B. Shearer. Based on results from [2], we chose $B_1 = 3 \cdot 10^6$, $\#P = 216815$, $B_3 = 11932088$, $\#Q = 783185 = 10^6 - \#P$, $B_2 = B_4 = 2^{30}$. We used $D = 50000$ (cf. 2.3). Logarithms on the algebraic side have base 2, with report bound 83; on the rational side these numbers are 1.6 and 42.

We tried low range ($5 \cdot 10^6$), medium range ($7 \cdot 10^6$), and high range ($11 \cdot 10^6$) q's, all of them with $C = 2^{13}$ and $E = 30000$ (cf. 2.6), which are realistic choices. The number of exceptional primes $\geq D$ on the rational side usually varied between 10 and 20; on the algebraic side the number grows with q, but never became more than 250. The total number of exceptional primes on either side was therefore never substantially more than $\pi(D) + 250 \approx 5300$.

With a bound of 23.5MB on the total memory use by our program, we could devote 14MB to S for the low range q's, 12MB for mid range q's, and 8MB for the high range. The 23.5 was chosen to make it easier to compare performance with both the 32MB and the pairs of 16MB Paragon nodes. If more than 23.5MB

can efficiently be used it is in general a good idea to do that. Our choice led to average sieving times of 1810, 1900, and 2300 seconds per q, and average trial division times of 360, 420, and 500 seconds per q, for the low, medium, and high range, respectively. Sieving and trial division for 500000 q's and our C and E can therefore be done in 1400 MIPS years.

After the algebraic sieve we typically have to remember <= 120000 pairs per S_i. The yield after the rational sieve decreases with i, usually from $\approx$ 800 to $\approx$ 80. On average slightly less than 1/4 of the accumulated reports survives 3.2, and a much smaller fraction of the survivors leads to a relation. The yields per q are roughly 109, 120, and 143 relations for the three ranges. The q's combined should therefore lead to more than $5 \cdot 10^7$ relations. According to data collected for another number in [5] it is very likely that this will lead to far more than 10^6 combinations. We refer to [5] for details, also concerning the distribution of the various types of relations.

References

1. D. Atkins, M. Graff, A. K. Lenstra, P. C. Leyland, Title to be announced, in preparation
2. D. J. Bernstein, A. K. Lenstra, A general number field sieve implementation, 103–126 in: [8]
3. J. Buchmann, J. Loho, J. Zayer, An implementation of the general number field sieve, Advances in Cryptology, Proceedings Crypto'93, Lecture Notes in Comput. Sci. **773** (1994) 159–165
4. J. Buchmann, J. Loho, J. Zayer, Triple-large-prime variation, manuscript, 1993
5. B. Dodson, A. K. Lenstra, NFS with four large primes: an explosive experiment, in preparation
6. Intel Corporation, Paragon(tm) XP/S Product Overview, 1991
7. A. K. Lenstra, H. W. Lenstra, Jr., Algorithms in number theory, Chapter 12 in: J. van Leeuwen (ed.), Handbook of theoretical computer science, Volume A, Algorithms and complexity, Elsevier, Amsterdam, 1990
8. A. K. Lenstra, H. W. Lenstra, Jr. (eds), The development of the number field sieve, Lecture Notes in Math. **1554**, Springer-Verlag, Berlin, 1993
9. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard, The number field sieve, 11–42 in: [8]
10. A. K. Lenstra, M. S. Manasse, Factoring by electronic mail, Advances in Cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci. **434** (1990) 355–371
11. B. Maccabe, K. S. McCurley, R. Riesen, SUNMOS for the Intel Paragon: A Brief User's Guide, Sandia National Laboratories Technical Report # SAND 93-1024
12. J. M. Pollard, The lattice sieve, 43–49 in: [8]
13. RSA Data Security Corporation Inc., sci.crypt, May 18, 1991; public information available by sending electronic mail to `challenge-rsa-list@rsa.com`
14. J. Zayer, personal communication, September 1993