The Journal of Supercomputing, 1,273-290 (1988)
© 1988 Kluwer Academic Publishers, Boston, Manufactured in the Netherlands

Parallel Implementation of the Quadratic Sieve

THOMAS R. CARON
ROBERT D. SILVERMAN
The MITRE Corporation,
Bedford, MA 01730

Abstract. A new version of the Quadratic Sieve algorithm, used for factoring large integers, has recently emerged. The new algorithm, called the Multiple Polynomial Quadratic Sieve, not only considerably improves the original Quadratic Sieve but also adds features that ideally suit a parallel implementation. The parallel implementation used for the new algorithm, a novel remote batching system, is also described.

1. Introduction

Although there has been much recent progress in the design and manufacture of new architectures for parallel processing, applications that can fully utilize all the available processing power are still rather rare. We present the design of a loosely coupled parallel processor constructed from a network of multiple SUN microcomputers using Ethernet protocols. This network has been used to implement a parallel version of a new algorithm for factoring large integers. Factoring, a problem that belongs to NP, has relevance to the RSA encryption system, so one would like to be able to carefully delineate current computing limits in factoring to determine how large an RSA key needs to be. The construction of our network and remote processing software enables us to use multiple SUN's as an asynchronous multiple-instruction, multiple data (MIMD) machine. The SUN operating system provides a centralized file system for all processors so that they can access data in a common file without explicitly passing data among processors. This feature is a great boon to application designers because they need not concern themselves with data passing mechanisms.

The version of the Quadratic Sieve used in this implementation was fully described in Silverman [1987]. It differs from the original algorithm by using multiple polynomials rather than just a single polynomial. This feature makes it well suited for parallel implementation. The idea of using multiple polynomials was independently developed by Davis and Holdridge [1983]; Davis, Holdridge, and Simmons [1985]; and Montgomery [1985], although they suggested somewhat different approaches. The forms of the polynomials used here, due to Montgomery, are somewhat better and easier to implement. A contrast of the Davis and Montgomery ideas was presented in Pomerance [1985] and Silverman [1987]. It is first necessary to give a review of the salient features of the multiple polynomial version.

The algorithm depends upon finding two squares in the same equivalence class mod N where N is the number one wishes to factor:

$$X^2 \equiv Y^2 \mod N. \tag{1.1}$$

If $X \not\equiv Y$ and $X \not\equiv -Y \mod N$, then $(X+Y, N)$ and $(X-Y, N)$ are proper factors of N.

Rather than construct (1.1) directly, one instead produces a large number of congruences of the form

$$X^2 \equiv Y \mod N \tag{1.2}$$

and then combines these congruences into (1.1). Specifically, using a quadratic polynomial, usually designated $Q(x)$, we first produce a very large number of these quadratic residues Y of N. Second, using a fixed set of primes, we attempt to factor Y, and then use the factorizations to find a set of congruences whose product is a square. Construction of $Q(x)$ is described below. Since $Q(x)$ is a polynomial, if a prime $p|Q(x)$, then $p|Q(x+kp)$ for all $k \in Z$. Thus, Q(x) can be factored with a sieve once $Q(x) \equiv 0 \mod p$ has been solved. The potential divisors p of $Q(x)$ are exactly those primes for which the Legendre symbol $(\frac{N}{p}) = 1$ and the unit -1 to hold the sign.

$Q(x)$ is chosen so that the produced residues are small relative to N. The smaller the residues, the faster the algorithm runs. So far the best that can be achieved is $Q(x) = O(\sqrt{N})$. This leads to an algorithm whose heuristic run time is [Pomerance 1983]:

$$L(N) = e^{(1+o(1))\sqrt{\ln(N)\ln\ln(N)}} . \tag{1.3}$$

In practice, for numbers of the size that can be factored today, adding three digits roughly doubles the run time. This factor of 2 drops very slowly to 1 as $N \to \infty$. To obtain even smaller residues, Coppersmith, Odlyzko, and Schroeppel [1986] recently suggested using a specialized cubic polynomial whose residues are $O(\sqrt[3]{N})$, which leads to an algorithm whose heuristic complexity is $O(L(N)^{\sqrt{2/3}})$; but it is not known whether this algorithm is practical. Constructing the cubic polynomical requires solving a very difficult cubic congruence.

The Quadratic Sieve algorithm continues as follows:

1. Select a factor base FB $= \{p_i \mid (\frac{N}{pi}) = 1, p_i$ prime, $i = 1, \dots, F\}$ for some appropriate value of F and $p_0 = -1$ for the sign.
2. Select a small integer k such that $kN \equiv 1 \mod 4$. A method for evaluating k, orginally due to Knuth and Schroeppel (unpublished), was discussed in Silverman [1987].
3. Compute $\sqrt{kN} \mod p_i$ and $[\log(p_i)]$ for all $p_i \in$ FB.
4. Compute $Q(x)$ and solve the quadratic congruence $Q(x) \equiv 0 \mod p_i$ for all $p_i \in$

FB. There will be two roots r_1 and r_2 for each p_i unless $p_i|k$, in which case there will only be one root at 0.

5. Initialize a sieve array to zero over the interval $[-M, M]$ for some appropriate M.
6. For all $p_i \in$ FB add the value of $[\log(p_i)]$ to the sieve array at locations:

$$r_1, r_2 \pm p_i, r_1 \pm 2p_i, \ldots \text{ and } r_2, r_2 \pm p_i, r_2 \pm 2p_i, \ldots .$$

7. Since the value of $Q(x)$ is approximately $M\sqrt{N}$ over $[-M, M]$, compare each sieve location with $[\log(N)/2 + \log(M)]$. Fully factored residues have their corresponding sieve values close to this value. For these, construct the exact factorization via division. Since successes are rare, the time to do this division is negligible.

If $Q(x_j)$, for some x_j is factored as

$$Q(x_j) = \prod_{i=0}^{F} p_i^{\alpha_{ij}} \quad p_i \in \text{FB} \ , \tag{1.4}$$

then let v_j be the corresponding vector of exponents $\alpha_{j1}, \alpha_{j2}, \alpha_{j3}, \ldots, \alpha_{jF}$, and let H_j be the square root of $Q(x_j)$ mod N, so that $H_j^2 \equiv Q(x_j) \bmod N$.

8. Collect $F+1$ factorizations and form an $F+1$ by F matrix from the vectors v_j reduced mod 2. Then find a set of residues whose product is a square via Gaussian elimination over $GF(2)$. This creates a linear dependency mod 2 on the exponents and the product of the vectors in that dependency forms a square. It is then trivial to construct an instance of congruence (1.1). This final stage of the algorithm typically takes only 1 or 2% of the total run time. For more details see Morrison and Brillhart [1975].

Remark. Achieving the asymptotic performance of (1.3) requires a better algorithm than Gaussian elimination. Such a method has been proposed by Wiedemann [1986], but it is not currently known whether it is faster in practice for values of F discussed here. The reason for this is that Gaussian elimination over $GF(2)$ on a binary computer can be effected simply by packing 32 columns to a 32-bit word and using exclusive or to add 32 columns at once. Implemented in this fashion, Gaussian elimination runs very quickly.

The above approach has a serious limitation. We must collect approximately as many fully factored residues as there are primes in the factor base. To obtain enough factorizations, M must be very large, but the residues grow linearly in size with M. Thus, they become progressively more difficult to factor. The solution to this problem was presented in Silverman [1987]. Use many polynomials $Q(x)$, change them frequently, and keep M small. This approach has the further advantage that its parallel implementation is very efficient. L machines give an L-fold speedup, for the computation of $Q(x)$ is driven by a sequence of prime numbers D (described below). Each selected value of D results in a different polynomial, and we can give separate sequences to each of the L machines.

2. Computation of the quadratic polynomials

We select the following polynomial:

$$Q(x) = Ax^2 + Bx + C \qquad (2.1)$$
subject to $B^2 - 4AC = kN$ for some $k \in Z$.

The condition on the discriminant is necessary for $Q(x)$ to yield quadrátic residues. Because of the form of the discriminant, another requirement is that $kN \equiv 1$ or 0 mod 4. To satisfy the constraint on the discriminant one must have

$$B^2 \equiv kN \bmod 4A. \qquad (2.2)$$

Select integers D and A with the following properties:

$$D^2 = A, \quad \left(\frac{D}{kN}\right) = 1, \quad D \equiv 3 \bmod 4, \quad \text{and} \quad A \approx \frac{\sqrt{kN/2}}{M} \qquad (2.3)$$

It is desirable that D be prime because if a prime in the factor base divides A, then $Q(x) \equiv 0 \bmod p$ has only one root and the probability that $p|Q(x)$ drops from $\frac{2}{p}$ to $\frac{1}{p}$. For practical purposes, it is sufficient that D be only a probable prime.The values given in (2.3) result in a choice of $Q(x)$ whose average value is minimized over $[-M, M]$, the sieve interval. The computation of the polynomial coefficients, given below, is driven by the value of D. One utilizes multiple values of D by starting with the values given in (2.3) and successively adding 4 to D, checking that it is both a probable prime and a quadratic residue of kN. Many composite values of D can be avoided by constructing a "wheel" — a sequence of D's not divisible by a chosen set of small primes [Knuth 1981]. To find the coefficients of $Q(x)$ compute

$$h_0 \equiv (kN)^{(D-3)/4} \bmod D \qquad (2.4a)$$

and

$$h_1 \equiv kNh_0 \equiv (kN)^{(D+1)/4} \bmod D\ . \qquad (2.4b)$$

Then,

$$h_1^2 \equiv kN(kN)^{(D-1)/2} \bmod D \tag{2.5}$$
$$\equiv kN \bmod D \text{ since } \left(\frac{D}{kN}\right) = 1.$$

Let

$$h_2 \equiv (2h_1)^{-1} \left[\frac{kN - h_1^2}{D}\right] \bmod D. \tag{2.6}$$

We now have

$$B \equiv h_1 + h_2 D \bmod A \tag{2.7}$$

and

$$B^2 \equiv h_1^2 + 2h_1h_2D + h_2^2D^2 \equiv kN \bmod A. \tag{2.8}$$

Since B must be odd, subtract it from A if it is even. The value of $(2h_1)^{-1}$ is easily obtained since $h_0 \equiv h_1^{-1} \bmod D$ has already been computed. Finally, because of the way D was chosen we also have

$$Q(x) \equiv H^2 \equiv \left(\frac{2Ax + B}{2D}\right)^2 \bmod kN. \tag{2.9}$$

And the roots of $Q(x) \equiv 0 \bmod p_i$, $p_i \in \mathrm{FB}$ are

$$(-B \pm \sqrt{kN})(2A)^{-1} \bmod p_i \tag{2.10}$$

since $B^2 - 4AC$ is invariant by (2.1). Note that $\sqrt{kN} \bmod p_i$ was precomputed in step 3 above.

Another approach to changing polynomials was also suggested by Montgomery and appears in Pomerance [1985]. In (2.3), rather than selecting D to be prime, instead select a large set of primes P_i, $i = 1, 2, \ldots, R$ near $\sqrt{D}$. Then select primes, in pairs, from this set and let D be the product of this prime pair. $\sqrt{kN} \bmod P_iP_j$ can be computed by computing $\sqrt{kN} \bmod P_i$ and $\sqrt{kN} \bmod P_j$ separately, and then applying the Chinese Remainder theorem. Also, computing $p_i^{-1} \bmod p_j$ for each $p_j \in \mathrm{FB}$ can then replace the modular inverse operation in (2.10) by two modular multiplications. The advantage of this approach is that once the precomputations are performed we essentially get $\binom{R}{2}$ sets of different D's and inverses at the cost of R sets of computations. Additionally, the arithmetic in (2.3) through (2.10) is easier, since P_i is much smaller than D. This approach works well on a single-machine implementation, but is ineffective for multiple machines, for reasons discussed below.

3. Some practical coding considerations

This algorithm was originally implemented on a single machine, a VAX 11/780. The program was written in C except for several very small assembler routines which did 64-bit arithmetic. Although this arithmetic was rather small, a few comments are in order, for it does influence the choice of computer used.

1. The sieving can be done using scaled integer approximations of the logs of primes in the factor base. Our experience showed that $[\log_3(p_i)]$ was sufficiently accurate. This also allowed the sieve array to be composed of cells that were each only a single byte. On some machines, however, the addition of two bytes is actually slower than adding together full words. On the other hand, using one byte for each cell cuts down on memory requirements. These considerations must be balanced against one another. Furthermore, it may be advantageous to partition the sieve array so that individual pieces fit within the machine's data cache, cutting down on memory fetches while sieving.
2. A factor base of 10,000 primes typically has its largest prime between 170,000 and 200,000. A factor base of 3000 primes, which is the size recommended for 60-digit numbers, typically has its largest prime between 50,000 and 60,000. Thus, doing numbers in the 60+ digit range requires more than 16 bits to hold the primes in the factor base. Much multiprecise arithmetic can therefore be avoided only on machines with a word size large enough to hold the factor base in single precision. Most of the cost of changing polynomials occurs in the computation of (2.10). In particular, (2.10) must be computed F times each time $Q(x)$ changes. Factorizations require anywhere from 10^3 polynomials for 40-digit numbers to 10^6 polynomials for 80-digit numbers. Keeping p_i single precision improves speed significantly. Typically, one uses the extended Euclidean algorithm to find $(1/2A) \bmod p_i$, which is expensive when p_i is greater than single precision.
3. Virtually all multiprecise arithmetic in the algorithm arises from the computation of equations (2.4)–(2.10). The work needed to compute the coefficients A and B is actually quite small and in practice it is never necessary to compute C, except perhaps as a check on the other computations. All of this arithmetic runs in polynomial time. We give estimates here of the total work needed for equations (2.4)–(2.9), assuming that a binary powering algorithm is used:

$$\begin{array}{ll}
(2.4) & O\ ({}^3/_2\, lg^3(D) + lg^2(D)) \\
(2.5) & O\ (lg^2(D)) \\
(2.6) & O\ (2lg^2(D) + lg(D)) \\
(2.7) & O\ (lg^2(A) + lg(A)) \\
(2.8) & O\ (lg^2(A)) \\
(2.9) & O\ (2lg^2(kN) + lg(kN))\,.
\end{array} \tag{3.1}$$

Most of the arithmetic occurs in (2.4) and in the computations that determine whether D is a probable prime. One can eliminate many of the probable prime tests by eliminating most composite values of D via the use of a sieve or wheel. All of the other computations involve only simple arithmetic operations mod N.

4. A sieve is a very efficient computational device; however, there are tricks we can use to speed it up as well. The first of these combines sieve initialization with the sieving of the smallest primes. For example, if $kN \equiv 1 \bmod 8$, then $2|Q(x)$ always. Therefore, rather than initialize the sieve array to 0, we can set each cell to [log (2)] and skip sieving with respect to 2. Similarly, if $3 \in \mathrm{FB}$, then we can predetermine the roots and include [log (3)] as well. Larger primes tend to have root patterns that are too complex to make this worthwhile. Furthermore, since the sieve is usually a byte array, time can be saved by equivalencing the array to a full word array and initializing the latter instead. Finally, since the smallest primes take the longest to sieve and contribute the least toward the accumulated log sum, one might skip them entirely, or sieve only with respect to their small powers. This variation, known as the small prime variation, is essentially due to Pomerance [1985]. It can improve speed 15 to 20% while sieving.
5. Since $D \approx \sqrt[4]{N}/\sqrt{M}$, on a 32-bit machine D is single precision for N up to about 45 digits and double precision for N up to about 75 digits. Clearly, it is advantageous to use machines that perform 32×32-bit multiplies and 64×32-bit divides in hardware or firmware. The algorithm slowed down on a machine restricted to 16×16-bit arithmetic by more than an order of magnitude. On a machine with 32×32-bit multiplies, sieving takes about 80 to 85% of the run time. On machines with only 32-bit arithmetic, changing polynomials dominates the computations.
6. Recommendations for the selection of M, F, and other parameters not discussed here may be found in Silverman [1987].

4. Network batching system

We have constructed a software system that allows the execution of many programs at once, each running on an independent workstation, from a central processor. Our network batching system runs on a collection of UNIX 4.2BSD systems connected by an Ethernet. The machines are a mixture of VAX's and SUN's of varying sizes. We summarize here those features that were relevant to our Quadratic Sieve implementation.

The network batching system enables users to access wasted CPU time in two ways: through a set of utility programs, accessible at the operating system level, and through a C language library, accessible by user programs. The library, which was used in our parallel implementation of the Quadratic Sieve, can start programs running in parallel on remote workstations, or satellites, and can communicate with the satellite programs from a central controlling program on a single host machine.

The batching daemon, which is within the operating system and therefore completely transparent to users, controls the network batching system. For example, the daemon maintains a job queue on each machine, and determines whether to awaken or put to sleep a batched job in a queue as its machine becomes idle or busy. It also supports an extensive set of utilities that help users monitor the status of batch jobs across the entire network.

Although the criteria for determining a busy satellite are configured into the system on a per workstation basis, they are usually defined to be when no users are logged in and the average number of runnable jobs over the past minute is less than one. The suspension of execution is unfortunate, but unavoidable, to prevent interference with interactive users of a satellite. The daemon will automatically restart a suspended process should its corresponding satellite no longer be busy.

Operations provided to the user can be classified as follows:

1. start programs on satellites;
2. communicate with programs on satellites;
3. synchronize programs on satellites;
4. monitor programs on satellites;
5. terminate programs on satellites.

For example, one function starts a program on a satellite. Another function obtains the name of a free satellite with specific characteristics—the user's requirements—such as the type of machine, the minimum amount of memory, and the specific operating system. Users can thus tailor their programs to machines that best suit their applications.

A secondary goal while designing the system was to simplify communication between host and satellite as much as possible. Each satellite program communicates with the host through the satellite's normal input and output channels. The host and satellites can therefore communicate using standard C input/output functions.

Because host program and satellite programs run independently, some form of synchronization is necessary. This is indirectly achieved through normal communication channels, but there is also a synchronous selecting function. Basically, this allows the host to wait until one or more members of a set of satellite programs has pending output. The function will block execution in the host program until at least one satellite program does have pending output. At that moment, the data associated with the first satellite with pending output are returned. An optional time limit can be specified as a parameter to this function to prevent the possibility of indefinite blocking of execution. Because input data are buffered, it is not necessary to select satellites that may be waiting for input. If a host program sends data to a satellite program before it is ready to receive them, the data will simply wait in a buffer until the buffer is exhausted by future input requests from the satellite program.

Since a satellite program can be suspended by the batching daemon or terminated abnormally by a satellite crash, there is a library function that enables the host program to monitor satellite programs. The host program periodically uses this function to take appropriate actions on satellite programs.

The number of satellite programs a host program can activate at any given time is limited by the resource limits imposed on the host program by the local operating system. This mainly depends on the amount of memory available for storing the required data structures. Because of these and other constraints, the host has functions to terminate a satellite program and to deallocate its associated data structures in the host program. It is also limited by the available I/O bandwidth of the network. One does not wish to overload the network with too many messages passed between host and satellites.

5. Description of parallel implementation

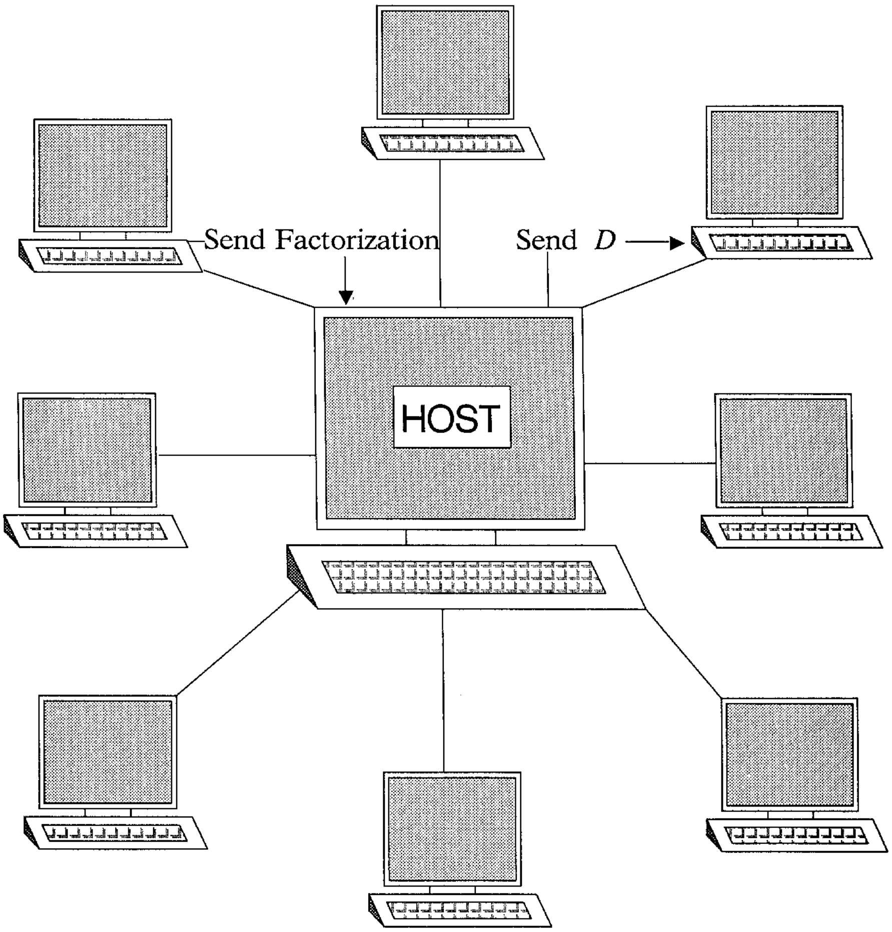
The simplest way to parallelize is to implement stand-alone versions of the program on many machines and give each machine its own initial value of D. Initial multiple values of D should be spaced far enough apart so that duplicate polynomials do not occur. After a little experience, we found it easy to estimate how far apart they should be. The value of D is near $\sqrt[4]{N}$; the average gap between primes near D is about $\log D$, approximately half of which will be quadratic residues. Thus, we spaced the initial values of D by about

$$2T\frac{\log N}{L}, \tag{5.1}$$

where T is the estimated total number of needed polynomials and L is the number of machines, leaving, of course, a margin for error. Estimates for the total number of required polynomials for numbers of differing sizes are given in Table 1. With this approach, each machine maintains its own separate data files for storing $Q(x)$ factorizations, values of H, and necessary restart data. The only restart data needed are the most recent value of D and a count of the factorizations. We strongly recommend that the value of D be frequently written out to disk because frequent stops and starts are common, especially with computations that can run in background for a week or more. Once enough factorizations are produced, the data from all machines are collected and step 7 of the algorithm begins. A way of determining when enough factorizations have been collected is presented in Silverman [1987]. It is clear that using this approach results in L machines giving an L-fold speedup. There is no comunication overhead to consider since each machine runs independently of others.

However, this implementation is difficult to control. Machines go down for a variety of reasons, and constantly monitoring them and restarting them when necessary is a nuisance. For this reason a *star configuration* is preferred. Figure 1 shows the actual network configuration. The satellites communicate only with the host, not with each other. This implementation of the Multiple Polynomial Quadratic Sieve is, with our network batching system, quite simple: The host computes and sends out values of D when requested by the satellites and receives back factorizations. These are the only required communications.

Figure 1. Star configuration.



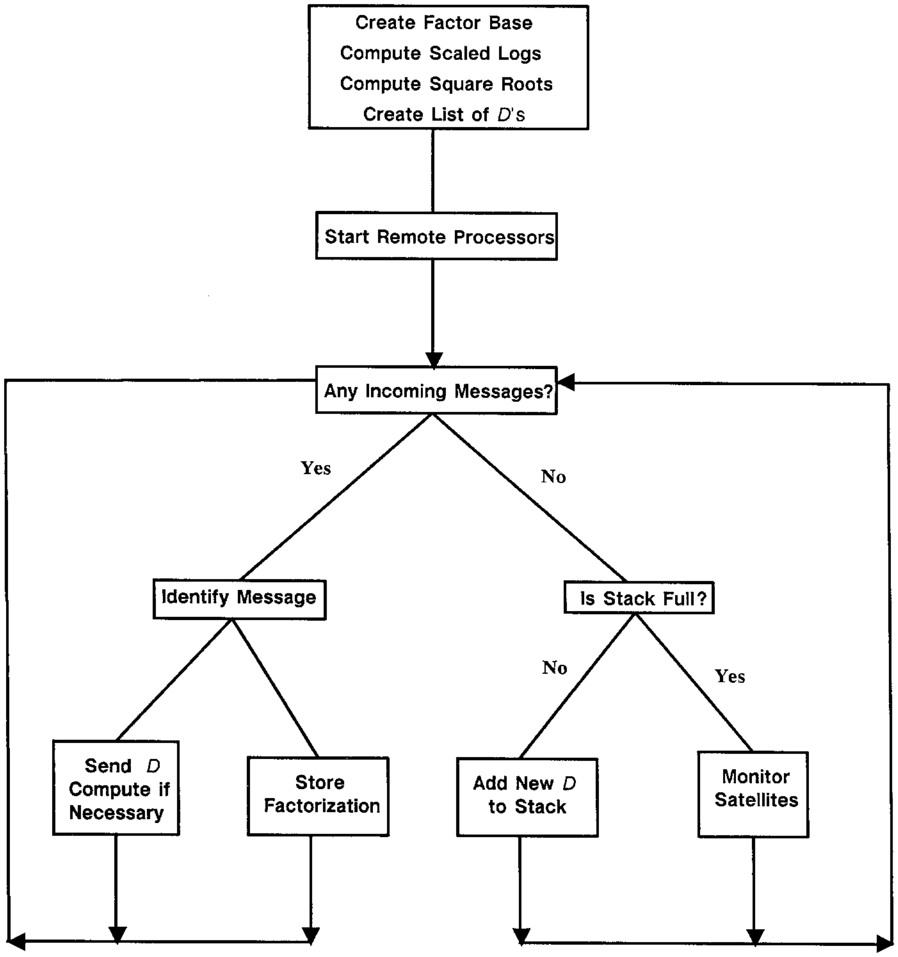
5.1. Host functions

The host machine has three responsibilities: first, to compute the sequence of D's; second, to store incoming factorizations from the satellites; third, to monitor the satellites and provide for satellite failures. This sequence of operations is detailed in Figure 2.

1. *Computation of D.* We wanted to keep all satellite processors as busy as possible, so when a request arrived from a satellite for a new value of D, the

host had already computed it. The simplest way to do this was to maintain a stack of D's. The length of this stack will be implementation and machine dependent. We used a stack of D's equal to twice the number of machines. When a request arrived for a D, it was already available on the stack. When the host had no incoming messages, it spent its time computing D's and augmenting the stack if necessary.

Figure 2. Host functions.



2. *Storage of incoming factorizations.* The second type of message received by the host was a factorization. Our implementation sent the factorization to the host as a formatted string already encoded for disk storage. All the host needed to do was read the message, determine that it was a factorization rather than a

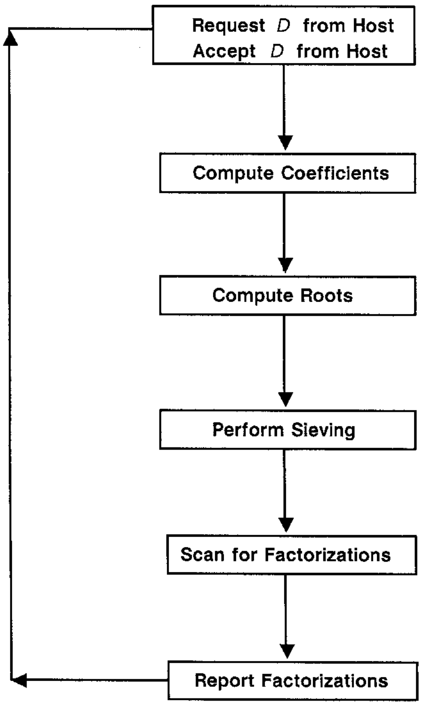
request for a D, and then store the string in a file. This operation takes virtually no time.

3. *Satellite monitoring*. There are two types of processor failure. The first, considered a critical failure because it halts all computation, occurs when the host or the network crashes. The only cure for this is a complete restart. To safeguard against this possibility the host frequently checkpointed the current value of D so that a restart would not duplicate polynomials. The second type of failure occurs when one of the satellites goes down. This is a noncritical failure because the satellite simply stops sending messages to the host—all other satellites continue running. To prevent gradual degradation of the system from satellite shutdowns, our host periodically checked whether all satellites were still running. When a satellite went down, the host attempted to restart the program on that machine or, if another satellite was available, on a previously unused machine. The automation of this process saved a great deal of trouble.

5.2 *Satellite functions*

Satellites have functional duties. That is, any computation on any satellite must have no side effects on the host or another satellite. A satellite's chores are to request and

Figure 3. Satellite functions.



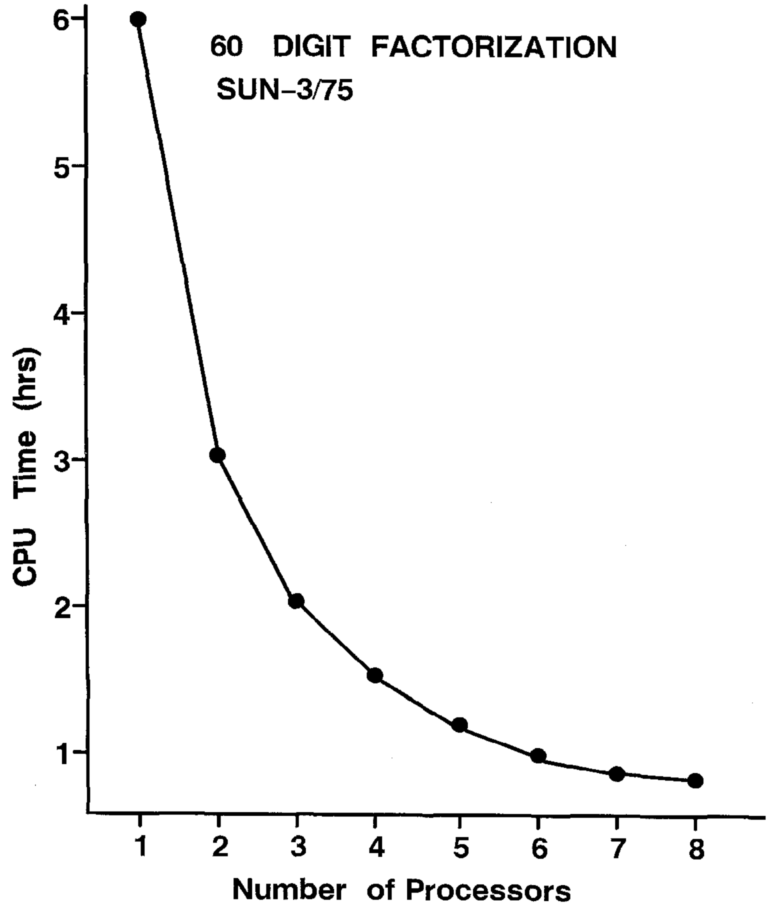
receive a value of D, construct $Q(x)$, compute its roots, perform the sieving, and report back any factorizations. Figure 3 illustrates this sequence of operations. The advantage of treating each satellite as a functional unit is that should one fail, the others can keep running without adverse effects.

6. Performance of the algorithm

6.1. Run time

We will first show the influence of using multiple processors for factoring a typical 60-digit number (Fig. 4). The number chosen is a cofactor of $3^{280}+1$, taken from the Cunningham Project [Brillhart and others 1983].

Figure 4. Algorithm performance.



Utilizing satellite processors is virtually 100% efficient. Of course, one could hook up enough satellites to overwhelm the host, but that can be obviated by implementing multiple stars using (5.1) and hooking the various hosts together. One could also lengthen the sieve interval, at the cost of some minor performance degradation. Increasing M would cause satellites to spend more time sieving; hence, the time interval between messages sent to the host would also increase.

6.2. *Communication requirements*

The total data storage requirement for an 80-digit number is about 20 Mbytes. This includes all of the necessary values of H_j and the vectors v_j. A practical device, known as the large prime variation, has been described by several authors and was originally suggested by Morrison and Brillhart [1975]. Rather than require that $Q(x)$ be factored completely over the factor base, we allow a partial factorization with one large prime factor lying outside the factor base. By collecting many of these large prime factorizations we can search for duplicates among the large prime factors and combine the respective factorizations to yield a power of two for that large prime. In practice, most of the data storage requirement arises from storing the large prime factorizations, since we typically find 30 large prime factorizations for each full factorization. The storage requirement can be modified by limiting the size of large prime factors we are willing to accept.

The data are transmitted as a string from satellite to host. For an 80-digit number each string is approximately 160 bytes in the following form:

$$\text{LP COUNT SIGN } I_1 \alpha_1 I_2 \alpha_2 \cdots I_{\text{COUNT}} \alpha_{\text{COUNT}} H.$$

LP is the large prime and is arbitrarily set to 0 to indicate when the factorization was complete over the factor base. COUNT is the number of nonzero exponents appearing in (1.4). SIGN is the sign of $Q(x)$. I_k α_k is the value of i and corresponding value of α_i in (1.4), and H is the corresponding square root of $Q(x)$. Typical values of COUNT range from 9 to 15. For an 80-digit number, using a factor base of 10,000 primes, one would typically find about 4500 complete factorizations and about 120,000 large prime factorizations. This latter figure depends, of course, on the size of the large primes one is willing to accept. We set a cutoff of 200 million. One can cut storage requirements, and hence host–satellite messages, by reducing the maximal size of the large primes. A reduction to 20 million would cut storage requirements approximately in half at the cost of a few percent in run time needed to generate more factorizations.

An 80-digit number takes about one week to factor using ten satellite processors. If the total data size sent back to the host is 20 Mbytes, this averages about 33 bytes/sec of data transmission on the network. In Table 1 the optimal number of polynomials is about 1/2 million; hence, requests for D arrive at the host about once every 1.2 sec. Each request for D requires 2 bytes and each value of D requires 2 bytes, so the total average data transmission rate is 37 bytes/sec. The bandwidth of our Ethernet is 128K bytes/sec so it is apparent that this type of implementation need not be concerned with data transmission capacity.

Implementing Montgomery's idea of letting D be the product of two nearly equal primes would require that each time the host sent a value of D to a satellite it would also have to send the accompanying inverses for each prime in the factor base. Since there are two prime factors of D and we need the inverse of each, a factor base of 10,000 primes would require the transmission of 20,000 inverses each time we change polynomials. The host could premultiply the inverses of the two primes, but even then we would have to send 10,000 inverses, and the host would have to do much more work. This would overwhelm either the host or the network bandwidth. It might be possible, however, to let each satellite generate its own supply of D's independently from its own set of P_i's by being very careful about selecting P_i on different machines so as not to create duplicate polynomials. There is only one foreseeable obstacle to this. Since P_i is half the size of D, there are fewer available values, and as P_i grows their product in pairs could wander from the optimal value of D quite quickly. We do anticipate, however, trying this approach.

6.3. Algorithm parameters

Table 1 gives optimal estimates for the total number of polynomials required for numbers of varying sizes. It is not uncommon to see variations in run time by a factor of two to three for numbers of a given size, primarily because some numbers are inherently richer in small quadratic residues. The number of polynomials given therefore represents an average value. Furthermore, the number of polynomials depends on M, since as M increases we need fewer polynomials. This table gives guidelines should one choose to implement independent stand-alone versions.

Table 1. Optimal algorithm parameters.

Digits	Polynomials	M	F
60	8K	150K	3K
63	15K	200K	3.5K
66	25K	250K	4K
69	45K	300K	5K
72	85K	350K	6K
75	150K	425K	7K
78	280K	525K	8K
81	450K	750K	10K
84	750K	1M	12K
87	1.2M	1.3M	15K

We used SUN-3's for both host and satellites. These turned out to have greater individual performances than we expected. Although they only yield about 70% greater performance than a VAX/780 for most applications, they ran the Quadratic Sieve three times faster, for the SUN-3 CPU has a built-in instruction cache that is large enough to hold all the code for the sieving loops of the program. While the program was sieving, a machine did not have to go to memory to fetch instructions. Thus, sieving speeded up about 3.5 times and overall performance about 3 times.

6.4. *Comparison with other implementations*

A single SUN-3/75 takes approximately 100 hr to factor a single 71-digit number. Davis, Holdridge, and Simmons [1985] reported the factorization of the 71-digit number $(10^{71}-1)/9$ in 9.5 hr using a CRAY X-MP. A private communication with Davis indicates that they have since been able to cut that time in half. Thus, we would need approximately 20 SUN-3/75's, running in parallel, to achieve a similar result. The exact price of a CRAY X-MP varies widely but is at least $10,000,000. A single SUN-3/75 costs about $15,000. Twenty SUN-3/75's would therefore show a minimum 33:1 price/performance improvement over the CRAY X-MP in executing this algorithm.

The Quadratic Sieve completely dominates the older, continued fraction algorithm even when the latter is run on special purpose hardware. For example, Wagstaff [1986] reports the factorization of a 62-digit number that took several weeks on the EPOC at the University of Georgia, a machine specially constructed to run the continued fraction algorithm. Wunderlich [1986] reports the factorization of a 64-digit number in about 9 hr using the massively parallel processor at NASA. A comparison of a single machine VAX implementation of both algorithms shows that even for very small numbers the quadratic sieve is faster [Silverman 1987].

The implementation described here shows that special hardware is not necessary to achieve large factorizations. Networks of microcomputers are becoming quite common. Our results show that using them instead of supercomputers or special hardware is much more cost effective.

7. Results

Table 2 shows some of our achieved factorizations. All were computed using 8 to 10 SUN-3's running in parallel. The CPU time, given in hours, is the total time summed over all satellites plus the time used by the host. Typically, the host was busy approximately 7% of the time in its computations and monitoring of the various satellites. Naive extrapolation therefore says one could hook about 150 satellite processors onto a single host before exceeding its capacity. However, as numbers grow larger, the time between host – satellite communications grows because the individual satellites take longer in their computations and report successful factorizations back less frequently. It is therefore difficult to predict how many satellites could be controlled by a single host because one would not, in fact, want to use all the available satellites in factoring smaller numbers.

The factorizations in Table 2 are all of composite cofactors taken from the Cunningham Project. The designation base, exp+ or base, exp− indicates a composite cofactor of $\text{base}^{\text{exp}} \pm 1$, respectively, while the designation base, exp M indicates a composite cofactor of a special algebraic factorizion of $\text{base}^{\text{exp}} + 1$, known as an Aurefeuillian factorization [Brillhart and others 1983]. The designation Pxx indicates a prime number of xx digits.

Table 2. Quadratic sieve factorizations in parallel with multiple SUN/3's.

N	Digits	Factors	CPU Time (hr)
6, 166+	69	437801891817492814657.P48	75
10,270M	70	13029637224192121671301.P48	88
3, 197+	71	49676157359100536013871955394289.P39	135
3, 193−	72	218246094772601642186973037.P44	195
3, 158+	72	10393819190815276250780568850989337.P38	210
2, 562M	74	58381440973934522510444290213069.P43	345
11, 95+	76	125391374277955216479795494641.P46	425
7, 109−	76	635057076990048880101785428103.P47	480
6, 106+	77	175436926004647658810244613736479118917.P39	590
11, 83+	77	47433880332031195178437273.P52	560
2, 277−	78	31133636305610209482201109050392404721.P40	650
2, 538M	78	122575221550682354302309961053.P48	663
11, 79−	79	185277551100523662054683911.P53	1720
2, 269+	81	424255915796187428893811.P57	1260
6, 121+	83	15186641018595718629290023681.P55	2150
3, 178+	84	1192464167514295068582330293.P57	1450
5, 128+	87	23653200983830003298459393.P62	4950

Acknowledgments

The authors thank the MITRE Corportation for providing the CPU time used in this project, the referees for several insightful comments, and John McCoubrey for his careful editing of this manuscript.

References

Brillhart, J., Lehmer, D.H., Selfridge, J.L., Tuckerman, B., and Wagstaff, S.S., Jr. 1983. *Factorizations of $b^n \pm 1$ for $b = 2, 3, 5, 6, 7, 10, 11, 12$, up to High Powers.* American Mathematical Society, Providence, Rhode Island.

Coppersmith, D., Odlyzko, A.M., and Schroeppel, R. 1986. Discrete logarithms in *GF(p)*. *Algorithmica*, 1: 1−15.

Davis, J.A., and Holdridge, D.B. 1983. Factorization using the quadratic sieve algorithm. Sandia National Laboratories Tech. Rept. SAND 83-1346.

Davis, J.A., Holdridge, D.B., and Simmons, G.J. 1985. Status report on factoring. *Advances in Cryptology: Lecture Notes in Computer Science*, pp. 183 – 215.

Knuth, D.E. 1981. *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, 2nd ed. Addison-Wesley, Reading, Massachusetts, p. 365.

Montgomery, P. 1985. Personal communication.

Morrison, M., and Brillhart, J. 1975. A method of factoring and the factorization of F7. *Math. Comput.*, 29:183 – 205.

Pomerance, C. 1983. Analysis and comparison of some integer factoring algorithms. In *Computational Methods in Number Theory* (H.W. Lenstra, Jr., and R. Tijdeman, Eds.), Mathematisch Centrum, Amsterdam, pp. 89 – 140.

Pomerance, C. 1985. A pipe-line architecture for factoring large integers with the quadratic sieve algorithm. *SIAM J. Comput.* Special Issue on Cryptography. To appear.

Silverman, R.D., 1987. The multiple polynomial quadratic sieve. *Math. Comput.* 48: 329 – 339.

Wagstaff, S.S., Jr. 1986. Personal communication.

Wiedemann, D. 1986. Solving sparse linear equations over finite fields. *IEEE Trans. Information Theory*, IT-32, 54 – 61.

Wunderlich, M. 1986. Personal communication.