SIAM J. COMPUT.
Vol. 17, No. 2, April 1988

012

A PIPELINE ARCHITECTURE FOR FACTORING LARGE INTEGERS WITH THE QUADRATIC SIEVE ALGORITHM*

CARL POMERANCE†, J. W. SMITH‡ AND RANDY TULER‡

Abstract. We describe the quadratic sieve factoring algorithm and a pipeline architecture on which it could be efficiently implemented. Such a device would be of moderate cost to build and would be able to factor 100-digit numbers in less than a month. This represents an order of magnitude speed-up over current implementations on supercomputers. Using a distributed network of many such devices, it is predicted much larger numbers could be practically factored.

Key words. pipeline architecture, factoring algorithms, quadratic sieve

AMS(MOS) subject classifications. 11Y05, 68M05

1. Introduction. The problem of efficiently factoring large composite numbers has been of interest for centuries. It shares with many other basic problems in the sciences the twin attributes of being easy to state, yet (so far) difficult to solve. In recent years, it has also become an applied science. In fact, several new public-key cryptosystems and signature schemes, including the RSA public-key cryptosystem [10], base their security on the supposed intractability of the factoring problem.

Although there is no known polynomial time algorithm for factoring, we do have subexponential algorithms. Over the last few years there has developed a remarkable six-way tie for the asymptotically fastest factoring algorithms. These methods all have the common running time

$$L(N) = \exp\{(1+o(1))\sqrt{\ln N \ln\ln N}\} \tag{1}$$

to factor N. It might be tempting to conjecture that $L(N)$ is in fact the true complexity of factoring, but no one seems to have any idea how to obtain even heuristic lower bounds for factoring. The six methods are as follows:

(i) The elliptic curve algorithm of Lenstra [6];
(ii) The class-group algorithm of Schnorr–Lenstra [11];
(iii) The linear sieve algorithm of Schroeppel (see [1] and [8]);
(iv) The quadratic sieve algorithm of Pomerance [8], [9];
(v) The residue list sieve algorithm of Coppersmith, Odlyzko and Schroeppel [1];
(vi) The continued fraction algorithm of Morrison–Brillhart [7].

It might be pointed out that none of these methods have actually been proved to have the running time $L(N)$, but rather there are heuristic arguments that give this function as the common running time. The heuristic analyses of the latter four algorithms use a new (rigorous) elimination algorithm of Wiedemann [13] for sparse matrices over finite fields. (This method may in fact be a practical tool; this is discussed further below.)

It should also be pointed out that to achieve the running time $L(N)$, method (vi) uses a weak form of the elliptic curve method as a subroutine.

As a last comment, the elliptic curve method has $L(N)$ as a worst-case running time, while the other five methods have $L(N)$ as a typical running time. The worst

* Received by the editors December 1, 1985; accepted for publication March 2, 1987. This research was partially supported by a National Science Foundation grant.
† Department of Mathematics, University of Georgia, Athens, Georgia 30602.
‡ Department of Computer Science, University of Georgia, Athens, Georgia 30602.

case for the elliptic curve method is when N is the product of two primes of about the same length. Of course, this is precisely the case of interest for cryptography.

A proof that $L(N)$ is the correct asymptotic complexity for factoring would of course be sensational. One corollary would be that $P \neq NP$. It seems to us, however, that of equal importance to practicing cryptographers is a complexity bound valid for finite values of N. Of course, such a bound necessarily enters into the amount of resources one is willing to invest in the problem.

To be specific, what is the largest number of decimal digits such that any number with this many or fewer digits can be factored in a year using equipment that would cost \$10,000,000 to replace? The dollar amount is not completely arbitrary; it is the order of magnitude of the cost of a new supercomputer. An answer to this question is not a fundamental and frozen constant, but rather a dynamic figure that reflects the state of the art at a given point in time. Nevertheless, it is an answer to precisely this kind of question that practicing cryptographers need.

Given an actual experiment where the running time is less than a year, performance for a full year can be extrapolated using the expression (1) (but ignoring the "$o(1)$"). The factoring algorithms listed above are all "divide and conquer," that is, with k identical computers assigned to the problem, the number will be factored about k times as fast. Thus if an actual experiment involves equipment costing less than \$10,000,000, again one can extrapolate.

Here are two examples. In 1984, the Sandia National Laboratories team of Davis, Holdridge and Simmons [3] factored a 71-digit number in 9.5 hours on a Cray XMP computer. Moreover, the algorithm they used, the quadratic sieve, has running time $L(N)$. Since

$$\frac{1 \text{ year}}{9.5 \text{ hours}} \approx \frac{L(10^{101})}{L(10^{71})},$$

we thus predict that the Sandia group could factor any 101-digit number in a year-long run on a Cray XMP.

The other example comes from some recent work of Silverman who has implemented the quadratic sieve algorithm on a distributed network of 9 SUN 3 workstations. Although at retail a SUN 3 is fairly expensive, it may be fairer for these purposes to use a wholesale price for a stripped down version, say \$5000 each. With this system, which we value at \$45,000, Silverman was able to factor an 81-digit number in one week. Since

$$\frac{1 \text{ year}}{1 \text{ week}} \cdot \frac{10{,}000{,}000}{45{,}000} \approx \frac{L(10^{126})}{L(10^{81})},$$

we predict that he would be able to factor any 126-digit number in a year-long run with 2000 SUN 3's.

In this paper we shall describe a machine which should cost about \$50,000 to build and which should be able to factor 100-digit numbers in two weeks. This custom-designed processor will implement the quadratic sieve algorithm. Since

$$\frac{1 \text{ year}}{2 \text{ weeks}} \cdot \frac{10{,}000{,}000}{50{,}000} \approx \frac{L(10^{144})}{L(10^{100})},$$

we predict that with \$10,000,000 any 144-digit number could be factored in a year. The cost to factor a 200-digit number in a year with this strategy would be about 10^{11} dollars—or only 5 percent of the current U.S. national debt!

2. Combination of congruences factorization algorithms. To properly describe the factoring project, it is necessary to begin with a description of the quadratic sieve (qs)

as it will be implemented. This is in fact a fairly complex algorithm with many stages. Some of these stages are common to other factoring algorithms, other stages are specific to qs.

The qs algorithm belongs to a family of factoring algorithms known as combination of congruences. With these, the basic goal in factoring N is to find two squares X^2, Y^2 with $X^2 \equiv Y^2 \bmod N$. If these squares are found in a random or pseudorandom fashion and if N is composite, then with probability at least $\frac{1}{2}$, the greatest common factor $(X - Y, N)$ will give a nontrivial factorization of N. This greatest common factor can be computed very rapidly using Euclid's algorithm.

The two congruent squares X^2 and Y^2 are constructed from auxiliary congruences of the form

$$u_i^2 \equiv v_i^2 w_i \bmod N, \qquad u_i^2 \neq v_i^2 w_i. \tag{2}$$

If some nonempty set of indices I can be found such that $\prod_{i \in I} w_i$ is a square, we can let

$$X \equiv \prod_{i \in I} u_i \bmod N,$$

$$Y \equiv \left(\prod_{i \in I} v_i \right) \left(\prod_{i \in I} w_i \right)^{1/2} \bmod N.$$

This special set of indices I can be found using the techniques of linear algebra. In fact, if the number w_i appearing in (2) has the prime factorization

$$w_i = (-1)^{\alpha_{0,i}} \prod_{j=1}^{\infty} p_j^{\alpha_{j,i}},$$

where p_j denotes the jth prime and there are only finitely many j for which $\alpha_{j,i} > 0$, then let $\vec{\alpha}_i$ denote the vector $(\alpha_{0,i}, \alpha_{1,i}, \cdots)$. Then the following two statements are equivalent for finite sets of indices I:

(i) $\prod_{i \in I} w_i$ is a square;

(ii) $\sum_{i \in I} \vec{\alpha}_i \equiv \vec{0} \bmod 2$.

Thus the algorithmic problem of finding I (when given a collection of integers w_i and their prime factorizations) is reduced to the problem of finding a nontrivial linear dependency among the vectors $\vec{\alpha}_i$ over the finite field with two elements.

For this method to work, we need the prime factorizations of the numbers w_i appearing in (2). It is the difficulty in finding enough completely factored w_i so that a linear dependency may be found among the $\vec{\alpha}_i$ that is the major bottleneck in the combination of congruences family of algorithms.

This bottleneck is ameliorated by discarding those w_i that do not completely factor with small primes. This is a good plan for two reasons. First, those w_i with a large prime factor probably will not be involved in a dependency. Second, those w_i that can be factored completely over the small primes can be seen as having this property with a not unreasonable amount of work. The set of small primes used is fixed beforehand and is called the "factor base."

3. The quadratic sieve algorithm. To make the above factoring scheme into an algorithm, we need a systematic method of generating congruences of the shape (2) and a systematic method of recognizing which w_i can be factored completely over the factor base. In the qs algorithm, we use parametric solutions of (2). That is, we exhibit three polynomials $u(x)$, $v(x)$, $w(x)$ such that for each integral x we obtain a solution

of (2) with $u(x) = u_i$, $v(x) = v_i$, $w(x) = w_i$. Moreover, those x for which $w(x)$ completely factors over the factor base can be quickly found by a sieve procedure described below.

In the original description of the qs algorithm [8], only one triple of polynomials $u(x)$, $v(x)$, $w(x)$ was used, namely

$$u(x) = [\sqrt{N}] + x, \qquad v(x) = 1, \qquad w(x) = ([\sqrt{N}] + x)^2 - N.$$

It was later found more advantageous to work with a family of triples of polynomials. The Sandia implementation of qs used a scheme of Davis and Holdridge [2]: If p is a large prime and $w(x_0) \equiv 0 \bmod p$, $0 \leqq x_0 < p$, then let

$$u_p(x) = u(x_0 + xp), \qquad v_p(x) = 1, \qquad w_p(x) = w(x_0 + xp).$$

A further refinement of this idea in [9] that has never been implemented is to choose x_1 with $w(x_1) \equiv 0 \bmod p^2$, $0 \leqq x_1 < p^2$, and

$$u_{p^2}(x) = u(x_1 + xp^2), \qquad v_{p^2}(x) = p, \qquad w_{p^2}(x) = \frac{1}{p^2} w(x_1 + xp^2).$$

A different and somewhat better scheme for choosing multiple polynomials was suggested by Peter Montgomery (see [9] and [12]). This method has been implemented by Silverman and is the method that we too shall use. Suppose we know beforehand that we will only be dealing with a polynomial $w(x)$ for values of x in $[-M, M)$, where M is some large, but fixed integer. Then we choose quadratic polynomials $w(x)$ that "fit" this interval well. That is, $w(M) \approx w(-M) \approx -w(0)$ and these approximately common values are as small as possible.

This task is done as follows. First choose an integer a with

$$a^2 \approx \sqrt{2N}/M \tag{3}$$

and such that $b^2 \equiv N \bmod a^2$ is solvable. Let b, c be integers with

$$b^2 - N = a^2 c, \qquad |b| < a^2/2. \tag{4}$$

Then we let

$$u(x) = a^2 x + b, \qquad v(x) = a, \qquad w(x) = a^2 x^2 + 2bx + c. \tag{5}$$

How do we determine which values of x in $[-M, M)$ give a number $w(x)$ that completely factors over the factor base? Fix the factor base as the primes $p \leqq B$ for which

$$t^2 \equiv N \bmod p \tag{6}$$

is solvable. The parameter B is fixed at the beginning of the program. We shall only recognize those values of $w(x)$ not divisible by any prime power greater than B. Presumably, if B is large enough compared with typical values of $|w(x)|$ (typical values are about $\sqrt{N}\, M$), most values that factor completely with the primes up to B will, in fact, not be divisible by any prime power greater than B.

For each prime power $q = p^\alpha \leqq B$ where p is in the factor base, solve the congruence

$$w(x) \equiv 0 \bmod q \tag{7}$$

and list the solutions $A_q^1, A_q^2, \cdots, A_q^{k(q)}$. The number of solutions $k(q)$ is either 1, 2 or 4. (Almost always we have $k(q) = 2$.)

Next we compute integral approximations to the numbers $\log |w(x)|$ for $x \in [-M, M)$. Because of the relationships of a, b, c, M, N to each other given by (3) and (4), the values of $[\log |w(x)|]$ tend to stay constant on long subintervals of $[-M, M)$. For example, it is about $[\log (M\sqrt{N/2})]$ for $|x| < \sqrt{1/4}\, M$ and $\sqrt{3/4}\, M < |x| < M$. It is about $[\log (M\sqrt{N/2})] - 1$ for $\sqrt{1/4}\, M < |x| < \sqrt{3/8}\, M$ and $\sqrt{5/8}\, M < |x| < \sqrt{3/4}\, M$, etc. Thus not only is this an easy computation for one choice of polynomial, but the results are virtually the same for each polynomial.

If $q = p^{\alpha} \leqq B$ where p is a prime in the factor base, let $\Lambda(q) = \log p$. We compute single precision values for the $\Lambda(q)$. This computation can be done once; it is independent of the polynomial used. For other integers m let $\Lambda(m) = 0$. Then every prime power factor of $w(x)$ is at most B if and only if $\log |w(x)| = \sum_{m|w(x)} \Lambda(m)$.

We are now ready to sieve. A memory addressed by integers x in $[-M, M)$ is initialized to zero. For each power $q \leqq B$ of a prime in the factor base and each $j \leqq k(q)$, we retrieve the numbers in those memory locations addressed by integers $x \equiv A_q^j \bmod q$, add $\Lambda(q)$ to the number there, and put this result back in the same place. This is what we call sieving and it is of central importance to the qs algorithm. In pseudocode it may be described as follows:

For each power $q \leqq B$ of a prime in the factor base and
$j \in \{1, \cdots, k(q)\}$ do:
  Let $A = A_q^j + \lceil (-M - A_q^j)/q \rceil q$ (thus A is the first number in $[-M, M)$ which is $\equiv A_q^j \bmod q$)
  While $A < M$ do:
    $D \leftarrow S(A)$
    $S(A) \leftarrow D + \Lambda(q)$; $A \leftarrow A + q$.

After sieving, we scan the 2M memory locations for values near $\log |w(x)|$. Such a location x most likely corresponds to a value of $w(x)$, all of whose prime power factors are at most B. It is possible that there could be some false reports or some locations that should have been reported that are missed. This error, which is introduced from our approximate logarithms, should be negligible in practice.

4. Fine points. In §§ 2 and 3 we described the basic qs algorithm using Montgomery's polynomials. In this section we shall give some adornments to the basic algorithm which should speed things up.

Use of a multiplier. We may wish to replace N in (3), (4) and (6) with kN where k is some small, fixed, positive, square-free integer. This trick, which goes back to Kraitchik [5, p. 208], can sometimes speed up implementation by a factor of 2 or 3. The idea is to skew the factor base towards smaller primes. However, there is a penalty in that the values $|w(x)|$ are larger by a factor of $\sqrt{k}$. We balance these opposing forces with the function

$$f(k, N) = -\frac{1}{2}\log k + \sum_{p \leqq B} E_p^{(k)},$$

where the sum is over primes $p \leqq B$, $E_p^{(k)} = 0$ if $t^2 \equiv kN \bmod p$ is not solvable, and

$$E_p^{(k)} = \begin{cases} \dfrac{2\log p}{p-1}, & p \text{ odd and } p \nmid k, \\ \dfrac{\log p}{p}, & p \text{ odd and } p \mid k, \\ \dfrac{1}{2}\log 2, & p = 2 \text{ and } kN \equiv 2 \text{ or } 3 \bmod 4, \\ \log 2, & p = 2 \text{ and } kN \equiv 5 \bmod 8, \\ 2\log 2, & p = 2 \text{ and } kN \equiv 1 \bmod 8, \end{cases}$$

if $t^2 \equiv kN \bmod p$ is solvable (see [9]).

When presented with a number N to factor, we first find the value of k which maximizes $f(k, N)$. In practice one can assume $k < 100$. Also in practice, we may replace B in the definition of $f(k, N)$ with a smaller number, say 1000.

Small prime variation. If $q \leqq B$ is a power of a prime in the factor base, then the time it takes to sieve with one A_q^j is proportional to M/q. Thus we spend more time with a smaller q, less with a larger q. However, small q's do not contribute very much. For example,

$$\sum_{q<30} \Lambda(q) < 42,$$

which is usually small compared to $\log|w(x)|$. In addition, it is usually not the case that every prime less than 30 is in the factor base and it usually is the case that a number which factors over the factor base is not divisible by all the factor base primes below 30. Thus only a small error is introduced by forgetting to sieve with the prime powers below 30. By lowering the threshold which causes a value $w(x)$ to be reported, no fully factored values need be lost. The only penalty is possibly a few more false reports. In fact, even this should not occur (see [9]). The small prime variation might save 20 percent of the running time.

Large prime variation. If x is such that

$$\log|w(x)| - \sum_{m|w(x)} \Lambda(m) < 2 \log B,$$

then either $w(x)$ completely factors with the primes in the factor base, or there is some large prime p,

$$B < p < B^2,$$

such that $w(x)/p$ completely factors with the primes in the factor base. Thus, by again lowering the threshold for reports, we can catch these values of $w(x)$ as well. For such a value to be eventually part of a linear dependency (see § 2), there must be at least one other report $w_1(x_1)$ using the same large prime p. The birthday paradox suggests that duplication of large primes p should not be so uncommon. In practice we shall probably only try to use those $w(x)$ which factor with a large prime $p < 100\,B$. The large prime variation (also used by Silverman [12]) speeds up the algorithm by about a factor of 2 or 3. However, the larger we take B, for a fixed N, the less useful will be this variation. It should also be noted that the large prime variation has been implemented in other combinations of congruences algorithms as well.

Generation of polynomials and sieve initialization data. Both experience [3], [12] and theory [9] suggest that it is advantageous to change polynomials fairly often. The reason for this is as follows. On $[-M, M)$, the largest values of $|w(x)|$ are about $M\sqrt{N/2}$. Moreover, more than half of the values are at least half this big. However, the larger is $w|(x)|$, the less likely it will factor completely over the factor base. Thus it would be advantageous to choose a rather small M. But M is directly proportional to the time spent sieving $w(x)$, so a smaller M translates to less time spent per polynomial.

Since we will want to change polynomials often, we should learn to do this efficiently. For each polynomial $w(x) = a^2x^2 + 2bx + c$ given by (5) we shall need to find a, b, c satisfying (3) and (4) and we shall need to solve all of the congruences (7). One last criterion is that a should be divisible either by a prime greater than the large prime bound or by two primes greater than B so that we do not get duplicate solutions of (2) from different polynomials.

One possibility, suggested in [9] and implemented in [12], is to choose a as a prime $\approx(\sqrt{2N}/M)^{1/2}$ with $t^2 \equiv N \bmod a^2$ solvable. Then we may let the solution for t least in absolute value be b and choose $c = (b^2 - N)/a^2$. Since a is a prime, the

quadratic congruence is easily solved. This then gives a legal polynomial $w(x)$ and it remains to solve the congruences (7) for all the powers $q \leqq B$ of a prime in the factor base. By the quadratic formula, these roots (for q coprime to $2a$) are given by the expression

$$(-b \pm \sqrt{N})a^{-2} \bmod q, \tag{8}$$

where $\sqrt{N}$ is interpreted as a residue $t_q \bmod q$ with $t_q^2 \equiv N \bmod q$ and where a^{-2} is the multiplicative inverse of $a^2 \bmod q$. The numbers t_q may be stored of course, since they are used in each polynomial. However, since each polynomial uses a new value of a in this scheme, it appears that an inversion mod q (using the extended g.c.d. algorithm) is necessary for each q and for each polynomial.

Thanks to a suggestion from Peter Montgomery, sieve initialization can in fact be accomplished with much less computation. The idea is to choose a as a product of l primes $g \approx (\sqrt{2N}/M)^{1/(2l)}$ with $t^2 \equiv N \bmod g^2$ solvable. (The value of l here is quite small, we plan to use $l = 3$ or 4.) Say $a = g_1 \cdots g_l$. Using known solutions $\pm b_i$ of the congruences $t^2 \equiv N \bmod g_i^2$ for $i = 1, \cdots, l$, we may assemble via the Chinese Remainder Theorem 2^l different values of $b \bmod a^2$ which satisfy $b^2 \equiv N \bmod a^2$. Since b and $-b$ will give essentially the same polynomial $w(x)$, we will get 2^{l-1} different polynomials for the one value of a.

Suppose now we use r primes $g_1, \cdots, g_r$ and we form different values of a by choosing l of these primes. Thus there are $\binom{r}{l}$ values of a and thus $2^{l-1}\binom{r}{l}$ different polynomials. If $a = g_{i_1} \cdots g_{i_l}$, then

$$a^{-2} \bmod q \equiv g_{i_1}^{-2} \cdots g_{i_l}^{-2} \bmod q. \tag{9}$$

Thus if the numbers $g_i^{-2} \bmod q$ are precomputed and stored for each $i = 1, \cdots, r$ and for each q, the computation of $a^{-2} \bmod q$ need not require any more inversions. It appears as if $l-1$ multiplications mod q are necessary when (9) is used to compute $a^{-2} \bmod q$. But if we form a new l-tuple of g's by trading just one g for a new one and saving an intermediate calculation, it takes only one multiplication mod q.

Here is another idea for sieve initialization that might be practical. Let $p_1, \cdots, p_l$ be a set of factor base primes with each $p_i \approx 500$ and with l as large as possible so that $K = p_1 \cdots p_l$ is still small compared with $\sqrt{N}/M$. Let $f(x) = ax^2 + 2bx + c$ be a polynomial with $b^2 - ac = N$, $|b| < a/2$, $a \approx \sqrt{2N}/KM$. Consider the solutions u of $f(x) \equiv 0 \bmod K$. For each solution u, let

$$g_u(x) = \frac{1}{K} f(u + xK).$$

There are 2^l choices of u and for each choice we obtain a polynomial $g_u(x)$ which may be sieved for x in $[-M, M)$. Suppressing the details, it turns out that with a small amount of precomputation, the sieve initialization data for each polynomial g_u may be computed with 2 additions mod q for each q. For $N \approx 10^{100}$, we may be able to take l as large as 14 or 15, so that 2^{14} or 2^{15} different polynomials may be generated in this way for the one value of K.

5. Implementation. Our implementation of the qs algorithm will have 5 stages: (1) preprocessing, (2) sieve initialization, (3) pipe i/o, (4) pipe, (5) postprocessing. Stages (2), (3), (4) occur simultaneously on three different devices that interact frequently. As their names suggest, stage (1) is completed before the other stages are begun and stage (5) is done only after all other stages have ended their work.

Stage (1): Preprocessing. This relatively minor stage involves the creation of various constants that are used in later stages of the algorithm. These include (i) choice of a multiplier, (ii) creation of the factor base, (iii) solution of the congruences $t^2 \equiv N \bmod q$ for each power $q \leqq B$ of a prime in the factor base, (iv) construction of a list of primes $g_1, \cdots, g_r$ and solutions of the congruences $t^2 \equiv N \bmod g_i^2$ as discussed in § 4, (v) computation of $g_i^{-2} \bmod q$ for each $i = 1, \cdots, r$ and each power $q \leqq B$ of a prime in the factor base.

The necessary inputs from which all of these numbers are created are N (the number to be factored), B (the bound on the factor base), M (half the length of the interval on which a polynomial is sieved), and r (where $4\binom{r}{3}$ is a sufficient number of polynomials to complete the factorization of N).

Preprocessing can be completed on virtually any computer with a large memory. For example, a SUN 3 workstation would be sufficient, even for very large numbers.

Stage (2): Sieve initialization. In this stage a three-element subset i, j, k is selected from $\{1, \cdots, r\}$ and from this triple a polynomial $w(x)$ is constructed together with sieve initialization data. Indeed, from the preprocessed data, the sieve initializer lets $a = g_i g_j g_k$ and chooses b, c to satisfy (4). This then defines a polynomial $w(x) = a^2x^2 + 2bx + c$. Next the sieve initializer computes A_q^j for each power $q \leqq B$ of a prime in the factor base via the formulas (8) and (9).

The actual sieving of $w(x)$ is performed in the next two stages. The sieve initializer has a direct link to the pipe i/o which controls the sieving. As soon as the sieve initialization data has been prepared, the pipe i/o and pipe cease their work on the previous polynomial and the pipe i/o receives the data for the next polynomial.

This configuration of tasks shows how the parameters B and M are related. The time for the sieve initializer to prepare a polynomial and sieve initialization data depends only on B, while the time for the sieving units to sieve the polynomial on $[-M, M)$ with the powers $q \leqq B$ of the primes in the factor base depends on both M and B. We choose the parameters B, M so that these two times are equal. In practice, we shall choose B first and then determine empirically the value of M that works.

Since the sieve initializer will be working as long as we are sieving, it would be desirable for it to be a dedicated piece of hardware. It is also desirable, but not crucial for the sieve initializer to be fast. A 50 percent speed-up of the sieve initializer may yield only a 15 percent speed-up in the total factorization time for a 100-digit number. While not negligible, this shows that resources might be more profitably allocated elsewhere. We are planning on dedicating a SUN 3 workstation to sieve initialization. It is likely we could build a custom processor for sieve initialization with the same performance as the SUN for less than \$5000. Although we currently do not anticipate building this custom processor, we nevertheless use the figure \$5000 for the cost of a sieve initializer in our estimate of the cost of the entire project.

Stages (3) and (4): Pipe i/o and pipe. It makes the best sense to give a joint overview of these two stages since the two units work in tandem to sieve a polynomial $w(x)$ on the interval $[-M, M)$ with the powers $q \leqq B$ of the primes in the factor base. These units form the heart of our factorization project and will be described in detail in §§ 5 and 6.

Since the number M will be relatively large in our implementation (for example, we may choose $M \approx 10^8$) it would take a large memory to sieve these $2M$ values all at once. It would be a more efficient use of resources to use a somewhat smaller memory; denote its length by I. (In one configuration of our machine we have chosen $I = 2^{20}$.) After the first I values are sieved, we then sieve the next I values and so on.

Thus the "polynomial interval" $[-M, M)$ is broken into a number of subintervals of length I.

The pipe then is a unit that can sieve I consecutive values of a polynomial $w(x)$. The pipe i/o is a unit that (i) receives from the sieve initializer the sieving data for a polynomial, (ii) initializes the pipe, (iii) sends out the sieving data $(A \bmod q, \Lambda(q))$ one arithmetic progression at a time into one end of the pipe, (iv) receives processed sieving data from the other end of the pipe and readies it for the next subinterval, (v) receives reported "successes" (locations x where $w(x)$ has been completely factored) from the pipe and sends them out to a host computer, probably the sieve initializer.

We will custom build the pipe i/o and pipe units. It is with these units, which will be specifically designed to sieve quickly, that we hope to achieve gains over previous implementations of the qs algorithm. The pipeline architecture is particularly well suited to sieving with "adjustable stride" which the qs algorithm demands. The usual problem of pipeline architectures, that of having software that keeps the pipe filled, is met here by custom tailoring of the hardware and software in the same project.

The pipe i/o and pipe will not need any diagnostic circuitry. Errors caused by hardware fault will either be detected during reporting of successes (either too few or too many reports will signal an error) or during post processing (a false report is detected here). It also should be noted that it is not necessary to design special checkpoint/restart capability since the operation of the algorithm involves sieving a new polynomial every five to ten seconds. Unlike primality testing where one glitch can throw out an entire primality proof, the quadratic sieve is a robust algorithm where local errors will not propagate.

Stage (5): Post-processing. Each reported success consists of four integers a, b, c, x such that

$$(a^2x+b)^2 \equiv a^2(a^2x^2+2bx+c) \bmod N$$

and such that $w(x) = a^2x^2+2bx+c$ completely factors over the factor base except possibly for one larger prime (see § 4 for a description of the large prime variation). The first step in post-processing is to compute the actual prime factorizations of the various successful numbers $w(x)$. This can be found by trial division. However, it is possible for the pipe i/o and pipe to immediately resieve in a special mode any subinterval in which a success is found. This special mode reports the prime powers which "hit," that is, divide, the number $w(x)$. If this is done then the prime factorization of $w(x)$ will be nearly complete and the postprocessor will have little work for this step. (Thanks are due to R. Schroeppel and S. S. Wagstaff, Jr. for this idea.) Without this resieving mode, as many as 10^{11} multiprecision divides would be necessary in post-processing (assuming a factor base of 10^5 and 10^6 reports). By resieving, we would have instead about the same number of low precision additions performed on specially tailored hardware.

Corresponding to each factorization of a $w(x)$ we have a (sparse) 0, 1 vector of exponents on the primes in the factor base reduced mod 2 (see § 2). The second step in post-processing is to find several linear dependencies mod 2 among these vectors. The third step in post-processing is to use a dependency to assemble two integers X, Y with $X^2 \equiv Y^2 \bmod N$, as discussed in § 2. The fourth and final step in post-processing is to compute $(X - Y, N)$ by Euclid's algorithm. If this gives only a trivial divisor of N, another dependency is used to assemble another pair X', Y', etc.

The most complex of these steps is the linear algebra required to find the dependencies. The length of the vectors depends on how large a value of B is chosen. The optimal choice of B for sieving may well be larger than 10^6. This would lead to vectors

of length about 39,000 or more (even ignoring the problem of encoding a large prime involved in a factorization in the same 0, 1 format). To factor very large numbers we may even wish to use vectors of length 100,000. Note that we need about as many vectors as their length. A 10^5 square matrix is probably too big to store in virtual memory on any commercially available computer (with the possible exception of a Cray 2), even given that each matrix entry is and will remain a single bit.

Here are several options that are available for the storage and processing of such a huge matrix. The problem of the large primes, ignored above, can be very easily solved using a sparse encoding of the vectors and quickly eliminating large primes by Gaussian elimination. This is quick and there is little fill-in since any factored $w(x)$ has at most one large prime in the interval (B, B^2).

This Gaussian elimination might be continued further, but now fill-in will begin to occur. It may be possible to then switch to the 0, 1 encoding and continue with Gaussian elimination on this smaller, but no longer sparse, problem.

A promising option is to use a sparse encoding and Wiedemann's elimination algorithm [13] for sparse matrices over a finite field (after the large primes have been eliminated as described above).

An unimaginative but possible plan is to use Gaussian elimination and the 0, 1 encoding (after the large primes are eliminated), but process only two slim slices of the matrix at any given time. This would involve a certain amount of i/o between the central memory and disc storage.

The matrix portion of post-processing will be performed on a large mainframe computer, perhaps the Cyber 205 at the University of Georgia. The other stages of post-processing will be performed on the same computer or perhaps on our SUN 3. In all, post-processing should not be time-critical for factoring; its difficulties will be solved in software on conventional computers.

6. The pipe. As mentioned above, the pipe is a unit capable of sieving I consecutive values of a polynomial $w(x)$ with the powers $q \leqq B$ of the primes in the factor base. We now describe details of its organization.

Block processors. The pipe is segmented with each segment consisting of a section of store and some arithmetic capability used in sieving. We call the section of store a *block*, and the arithmetic capability together with the store a *block processor* (BP). The size of the storage on each BP is denoted $|\mathrm{BP}|$. It is necessary to choose $|\mathrm{BP}|$ a power of 2; our working figure is $|\mathrm{BP}| = 2^{16}$ which we shall assume in the following. The word length of this store is nominally eight bits, since this will provide the resolution required for the approximate logarithms in the sieving process. The number of BP's in the pipe ($\#\mathrm{BP}$) is also a power of 2; our working figure is 16. The total store in the pipe is $\#\mathrm{BP} \times |\mathrm{BP}| = 2^{20} = I$, the length of a subinterval that we sieve at a given time.

Subinterval processing. First, the pipe must be initialized for the subinterval. This involves setting the contents of the BP store to a constant (we use 0), and setting the threshold value T at which each BP will detect a result. The pipe i/o will direct these operations.

Next, the sieve must be run. The pipe i/o will send out the progression records. These are entered into the pipe as rapidly as possible. Then when they exit the pipe they are stored in the pipe i/o for use in successive subintervals. During sieving, the BP will add $\Lambda(q)$ to each location at which the progression $A \bmod q$ hits. Some locations may then reach the threshold level T.

Reports. If a BP detects that a location has exceeded the threshold value, it will immediately report this to the pipe i/o. At the end of sieving, the address(es) that exceeded the value is (are) reported for use in post-processing.

One refinement which increases the utility of the reports is to report not only the address, but each of the prime powers that hit at that location. This will reduce the work required in post-processing. We can accomplish this goal by running the sieve "backwards" from the new set of A's we have calculated, observing those that hit the marked location. (There is no special need to "mark" locations, however. By setting each $\Lambda(q) = 0$ for the re-sieving, only locations already exceeding the report tolerance from before cause a report to be made.) The pipe is wired so that information "flows" only in one direction, so sieving "backwards" must be simulated by reversing the order of the store in the pipe.

When a reportable result is found, the BP will raise a flag called "report request," which activates a daisy chain protocol. This signal will stop all the BP's simultaneously in the middle of the next cycle. If the pipe i/o has been in sieve mode, it reads from the BP the address that caused the threshold to be surpassed and then continues to sieve normally the rest of the subinterval. The pipe i/o then enters the re-sieve mode. This begins with reversing the order of the store in the pipe. (In fact, it is only necessary to reverse the order of the store of the reported address or addresses. In practice it may be simpler to just re-initialize the key location(s) with some preset value that we know will be above the threshold.) Next each BP is set to the re-sieve mode; this entails subtracting, rather than adding in the address register. Recall that each $\Lambda(q)$ has been set to 0. Thus there will be report requests at only those marked locations that have been preset with above threshold values. Now when there is a report request, the pipe i/o transfers the prime power from the BP, not the address. When the subinterval has been completely re-sieved the pipe i/o returns to sieve mode. When the polynomial has been completely sieved, the reports are transferred to the host.

Pipe arithmetic. The arithmetic capability which is on the BP is concentrated into address and data arithmetic units which do the following operations:

$$A \leftarrow A + q,$$

$$D \leftarrow S(A),$$

$$S(A) \leftarrow D + \Lambda(q).$$

Since there are two separate arithmetic units for address and data arithmetic, these arithmetic operations can proceed in parallel. This is an important consideration in the performance of the BP.

This arithmetic capability, while simple, allows a BP to perform several different functions:

```
Initializing:  While (A < |BP|){
                    S(A) ← 0;
                    A ← A + 1;
                    }

Sieving:       While (A < |BP|){
                    S(A) ← S(A) + Λ(q);
                    If (S(A) > T) {report A}
                    A ← A + q;
                    }
```

```
Reporting:  While (A > 0){
                S(A) ← S(A)+0;
                If (S(A) > T) {report q}
                A ← A − q;
                }
```

Interconnects. Each BP is connected to both of its neighbor BP's (predecessor and successor). The predecessor of the first BP and successor of the last BP is the pipe i/o unit. In addition, each BP has a bus connection to the pipe i/o unit. From the predecessor, each BP receives the data items A, q, $\Lambda(q)$ which are the progression record for sieving and sends the signal "BUSY," which while on tells the predecessor not to send anything. To the successor, each BP sends the data items A, q, $\Lambda(q)$ and receives the signal "BUSY."

From the pipe i/o bus, the BP receives initialization instructions and a report tolerance T (valid for this BP in this subinterval run). To the pipe i/o bus, the BP sends the signal "SUCCESS" if some $S(A) > T$. In this case, the A and q report values are sent to the pipe i/o unit over this bus. The BP control modes are broadcast to the pipe from the pipe i/o unit over this bus as well.

Performance. The fundamental performance parameter of the BP is the cycle time of the BP store, C. All the other performance values can be stated in terms of this value.

The time of a prime power q in a BP is at most $\lceil |\mathrm{BP}|/q \rceil \times 2C$. That is, each sieve step is accomplished in two cycles. To see how this can be done we consider the worst possible case, namely when several arithmetic progressions each successively hit exactly once in a particular BP. Thus in two cycles, the BP needs to receive the sieving data, recognize that there is a hit in this BP, do the sieving, see if there is a SUCCESS, recognize that the arithmetic progression does not hit a second time, and send out the altered sieving data. Also we shall see in the next section that the sieving data is not sent all in one cycle, but the A value is sent in one cycle and the q, $\Lambda(q)$ values are sent in the next cycle. This worst case is outlined in Table 1 which shows what happens to two consecutive sieving records A_i, q_i, $\Lambda(q_i)$ for $i = 0, 1$.

Thus during an even-numbered cycle in this worst-case scenario, the BP performs the five operations listed for cycle 2 above. During an odd-numbered cycle, the BP

TABLE 1

Cycle	Progression 0	Progression 1
0	In $[A_0]$: BPID $[=]$ $A \leftarrow \mathrm{IN}[A_0]$	
1	$D \leftarrow S(A)$ $A\mathrm{in} \leftarrow A + q\mathrm{in}$; $A\mathrm{in}$: BPID $[\neq]$ $q, \Lambda(q) \leftarrow \mathrm{IN}[q_0, \Lambda(q_0)]$	
2	$\mathrm{OUT}[A'_0] \leftarrow A\mathrm{in}$ $S(A) \leftarrow D + \Lambda(q)$; $S(A)$: T $[<]$	IN $[A_1]$: BPID $[=]$ $A \leftarrow \mathrm{IN}[A_1]$
3	$\mathrm{OUT}[q_0, \Lambda(q_0)] \leftarrow q, \Lambda(q)$	$D \leftarrow S(A)$ $A\mathrm{in} \leftarrow A + q\mathrm{in}$; $A\mathrm{in}$: BPID $[\neq]$ $q, \Lambda(q) \leftarrow \mathrm{IN}[q_1, \Lambda(q_1)]$
4		$\mathrm{OUT}[A'_1] \leftarrow A\mathrm{in}$ $S(A) \leftarrow D + \Lambda(q)$; $S(A)$: T $[<]$
5		$\mathrm{OUT}[q_1, \Lambda(q_1)] \leftarrow q, \Lambda(q)$

performs the five operations listed in cycle 3. Although an arithmetic progression involves the BP for 4 cycles, it can travel down the pipe spending 2 cycles in each BP.

If the first compare with BPID is [$\neq$], this means the arithmetic progression does not hit in this BP and it can be sent on through to the next BP. If there is a hit (as shown in the pseudocode above) and the second compare with BPID is [=], then this means that the arithmetic progression hits a second time in this BP in which case it is, of course, not sent out right away.

Collision avoidance. To avoid collisions a BP will register "BUSY" during the time a prime power occupies the BP. We classify prime powers q in three categories. A value q is "small" if $q < |\mathrm{BP}|$, so that q has the potential to hit more than one location in a BP. It is "moderate" if $|\mathrm{BP}| \leqq q \leqq I$, so that q hits at most once in any BP, but will definitely hit at least one BP in the subinterval. Finally, q is "big" if $I < q \leqq B$. Big prime powers hit at most one BP in a subinterval. The progressions are sent through a subinterval in the following order: (i) big prime powers that actually hit some BP in the subinterval; (ii) moderate prime powers ordered by decreasing size; (iii) small prime powers ordered by decreasing size. This organization of the progressions keeps delays in the sieve caused by "BUSY" signals to a minimum. In fact there are no delays at all with big and moderate prime powers.

7. The pipe i/o unit. The pipe i/o unit is an interface, storage, and control mechanism for the sieving process. The pipe i/o unit interfaces to the sieve initializer to receive progression records for each polynomial, and to send reports of the sieve's successes. It stores the progression records, since the length I of the subinterval that is sieved at one time is considerably less than the length $2M$ of the polynomial interval. The pipe i/o sends the progression records in a proper order to the pipe, then receives the modified records from the pipe and stores them for the next subinterval. The pipe i/o contains the control and sequencing logic that controls the pipe and its modes of operation.

Interfaces. The pipe i/o unit has a data path to the sieve initializer (SI). When a polynomial change is to occur (determined by the SI), the operation of the pipe is halted. The storage on the pipe i/o unit is then loaded with the progression records for the next polynomial interval. When this operation is completed, the pipe i/o begins the sieving sequence for the new polynomial by sending out the progressions in order. The time required to load the pipe i/o store is a function of the amount of data to be transferred, the width of the data path between the sieve initializer and the pipe i/o, and the bandwidth of the respective memories. (We assume that the loading will be a direct memory-to-memory transfer.)

The other operational use of the data path from SI to pipe i/o is for the reporting of results. When the sieve has a result to report, it will send polynomial coefficients, a polynomial argument, and a list of powers of primes from the factor base that divide the polynomial at the argument. The sieve initializer will receive these reports and retain them for the post-processing step.

Store. The storage of the pipe i/o must be large enough to accommodate usually two progression records for each power $q \leqq B$ of a prime in the factor base. At the same time, it must be fast enough to feed the pipe at "full guzzle" while sieving by moderate and big prime powers. This storage unit is one of the major challenges of the qs processor.

Each progression record consists of A, q, $\Lambda(q)$ for the sieving, and a link field which is used by the pipe i/o to send out only those progressions which will hit in at least one BP in the subinterval. The length of a progression record is

$$\max|A|+\max|q|+\max|\Lambda(q)|+\max|L|=32+24+8+20=84 \text{ bits},$$

nominally. Since the q, $\Lambda(q)$ are identical for usually two progressions, these fields may be shared between the two progressions, reducing the nominal size to 68 bits. There must be a progression record for every solution of (7) for each power $q \leqq B$ of a prime in the factor base. If we choose $B = 2{,}750{,}000$, say, there will be about 200,000 progressions requiring $200{,}000 \times 68$ bits $= 1.7$MB store.

Speed. The pipe i/o store must be able to keep up with the pipe. For this reason, we wish to be able to feed out a progression record in time $2C$, since that is the rate at which the pipe can accept progressions with modulus a moderate or big prime power. We intend to fetch the progression record in parallel (all 84 bits) from the pipe i/o store, store it in a buffer register, then send out the A in one cycle, q, $\Lambda(q)$ in the next (recall that the L field is for the use of the pipe i/o only).

This makes it appear that the cycle time of the pipe i/o store can be double that of the pipe's store. However, this is not the case. Once the transaction record has made it through the pipe, it will be received by the pipe i/o and must be stored for use in the next subinterval. This means that it must be stored in the pipe i/o store. Since receive transactions are occurring while send transactions are still going on, we must be able to WRITE the pipe i/o store once and READ it once in the time $2C$. This means that the cycle time of the pipe i/o store must be the same as that of the pipe.

Partitioning. The pipe i/o requires a large, fast store. More than this, since the pipe i/o store is about twice the size of the factor base, we must partition the store horizontally if at all possible so that the factor base size is not a hard, "designed in" limit on the system.

Operation. Once the pipe i/o store is filled with progression records, operation of the sieve begins. The pipe i/o controls the sieving operation, which consists of a cycle in which the events:

initialize the pipe
sieve by progressions which hit in subinterval i
report any successes in subinterval i
$i \leftarrow i+1$

are repeated until the sieve initializer has the next polynomial ready.

Sending progressions. Since a big prime power may not hit in subinterval i, we keep a linked list of those we know will hit in the subinterval. Thus we only dispatch those big prime powers that will hit in the subinterval. The pipe i/o manages the linked lists on a per-subinterval basis, so that each subinterval has a list of the prime powers that will hit in that subinterval. This list might have the big prime powers out of numerical order, but since they hit only once in i anyway, it will not result in a pipe collision. As mentioned before, progressions corresponding to moderate and small prime powers are sent out in reverse numerical order of modulus. Thus the linked list mechanism is not necessary for these progressions.

Receiving progressions. When a progression record A, q, $\Lambda(q)$ is sent into the pipe, it is processed there to generate the next progression record by modification of the A value. The pipe will add q to the A value until the new A no longer falls in the

subinterval. Then the progression record will exit from the pipe. At the exit from the pipe is the pipe i/o receiver which will store the record for use in the progression's next subinterval.

Recall that |BP| and I are both powers of 2. This means that one field of A can be regarded as the subinterval number i. This field of A is important in the processing of the received record upon exit from the pipe; the value of i will indicate the next subinterval in which the progression will fall. For moderate and small prime powers this is always the next subinterval. For big prime powers, up to $n-1$ subintervals can be skipped over where $n = [B/I]$. There are n linked lists maintained in the pipe i/o, and the progression will be added to the appropriate list upon exit from the pipe.

Each progression has a home location in pipe i/o store where it resides. When the progression is sent into the pipe, its home address is entered into a FIFO store. Since the pipe preserves the time order of progressions, the home address can be retrieved from this store when the progression is received. The new A will then be stored at the home address location. Additionally, the progression is added to the proper interval list by the simple procedure of placing the list pointer in the link field of the progression, then placing the progression address in the list pointer.

Receiving this progression information must compete for pipe i/o store with the sending process. Only A and L must be stored, and this will require one cycle of store. Since the A and L are buffered and since the sending process uses only one cycle from every two, storing the progression uses the other cycle.

8. Performance on 100 digit numbers. In this section we give some indication how performance on 100 digit numbers can be estimated. In the sequel, we assume $N \approx 10^{100}$, where N is the number to be factored.

Factor base. We shall assume the multiplier is one (see § 4). We shall consider a factor base of 100,000 primes so that B, the bound for the largest prime in the factor base, is about 2,750,000. We estimate that sieve initialization for this choice of B will take five seconds. The pipe i/o unit will require about 1.7MB of store.

Time to sieve a subinterval. We shall assume that each BP has size 2^{16} and that the pipe consists of 2^4 BP's. This gives the value 2^{20} to I, the subinterval length. We assume the cycle time in the pipe i/o and pipe is 70 nanoseconds. The time to sieve a subinterval has several components (measured in milliseconds):

(i) Initialize the pipe—4.59;
(ii) Small prime powers—11.30;
(iii) Moderate prime powers—10.51;
(iv) Big prime powers—9.87;
(v) Empty the pipe with least progression—1.47.

We shall account for reporting time later. Thus the total time to sieve a subinterval of length $I = 2^{20}$ is 37.74 milliseconds.

With one polynomial, we sieve for 5 seconds, the sieve initialization time. Thus in this time we shall sieve about 1.38×10^8 values, so that $2M \approx 1.38 \times 10^8$.

Success rate. The largest values of a polynomial will have size $M\sqrt{N/2}$. However, the smaller values will give a disproportionately high success rate. Thus we shall assume the "typical" value is

$$\tfrac{1}{3}M\sqrt{N/2} \approx 1.63 \times 10^{57}.$$

The probability that a random number of this size completely factors with the primes up to 2×10^6 can be estimated from the table for "$\rho_1(\alpha)$" given in Knuth and Trabb

Pardo [4]. We take

$$\alpha = \frac{\log(1.63 \times 10^{57})}{\log(2750000)} = 8.8848.$$

We geometrically interpolate in the table to get

$$\rho_1(\alpha) \approx 1.513 \times 10^{-9}.$$

This means that we have to sieve about $1/\rho_1(\alpha) \approx 6.61 \times 10^8$ numbers to find one success. That is, we have one success every 23.9 seconds on average.

Large prime variation. Multiplying 23.9 seconds by 100,000, the nominal number of successes we shall need, we obtain a sieving time of 27.7 days. However, by also reporting polynomial values which factor completely over the factor base except for one large prime in the interval $(B, 100\,B)$, we estimate a speed-up factor of 0.415. This estimate is obtained by splitting $(B, 100\,B)$ into smaller intervals, using the Knuth-Trabb Pardo table to estimate success rates for large primes in the smaller intervals, and then using a "birthday paradox" analysis to estimate the usefulness of these large prime factorizations. Thus the 27.7 day sieving time is reduced to 11.5 days.

Enforced sieve down time. The sieve must be down when the pipe i/o is being loaded from the sieve initializer and when it is in the reporting mode. We assume that loading time per polynomial is 0.2 seconds. This must be done for every five-second polynomial run, or for a total of 0.5 days during the factorization.

With the large prime variation, there will be about 700,000 reports during the run. We shall assume, however, there are 10^6 reports since some of these will be "false." We also assume that a report takes 75 milliseconds (about twice the time to sieve a subinterval) for recall that reporting involves re-sieving a subinterval in a new mode. Thus total reporting time is about 0.9 days.

Total running time. We shall assume that preprocessing and post-processing together take at most 0.5 days of computing time. Thus our total running time can now be estimated from the following:

(i) Sieving time—11.5 days;
(ii) Loading time—0.5 days;
(iii) Reporting time—0.9 days;
(iv) Pre- and post-processing—0.5 days;

or 13.4 days.

Estimated cost of processor. Pre- and post-processing are performed on conventional hardware. The most critical step is the matrix processing discussed in § 5. It is assumed that this will not be an important bottleneck. We do not include the cost of buying computer time for these stages since we are considering here only a relatively insignificant portion of the factoring project. In extrapolations to very large numbers, it might be fair to set aside 5–10 percent of monetary resources for pre- and post-processing.

Sieve initialization will be performed on a dedicated SUN 3. It would be possible to build a dedicated processor with equal performance for sieve initialization tasks for $5,000. Even though a SUN 3 costs about ten times as much, we nevertheless use the figure $5,000 for the cost of a sieve initializer.

We estimate the cost of parts for a pipe i/o unit with 1.7MB of store at $10,000. Finally we estimate the cost of a pipe consisting of 16 BP's, each of size $64k \times 8 = 64kb$, together with a power supply, at $10,000.

Thus we estimate a total cost of $25,000 for parts. To figure in development costs, overhead, and a margin for error, we use a multiplier of 2, thus bringing our estimate to $50,000.

9. Summary. We have described the quadratic sieve factorization algorithm and an inexpensive processor on which it can be efficiently run. If it runs as quickly and is as inexpensive as we think, then 144-digit numbers can be factored in a year with a budget of $10,000,000.

Acknowledgment. The authors acknowledge the constructive criticisms and helpful suggestions of Peter Montgomery, Andrew Odlyzko, Richard Schroeppel, Sam Wagstaff and the referees.

REFERENCES

[1] D. COPPERSMITH, A. M. ODLYZKO AND R. SCHROEPPEL, *Discrete logarithms in GF(p)*, Algorithmica, 1 (1986), pp. 1-15.

[2] J. A. DAVIS AND D. B. HOLDRIDGE, *Factorization using the quadratic sieve algorithm*, Tech. Rpt. SAND 83-1346, Sandia National Laboratories, Albuquerque, NM, December, 1983.

[3] J. A. DAVIS, D. B. HOLDRIDGE AND G. J. SIMMONS, *Status report on factoring* (at the Sandia National Laboratories), in Advances in Cryptology, Lecture Notes in Computer Science 209, 1985, pp. 183-215.

[4] D. E. KNUTH AND L. TRABB PARDO, *Analysis of a simple factorization algorithm*, Theoret. Comput. Sci., 3 (1976), pp. 321-348.

[5] M. KRAITCHIK, *Théorie des Nombres*, Tome II, Gauthier-Villars, Paris, 1926.

[6] H. W. LENSTRA, JR., *Factoring integers with elliptic curves*, Ann. of Math., to appear.

[7] M. A. MORRISON AND J. BRILLHART, *A method of factoring and the factorization of F_7*, Math. Comp., 29 (1975), pp. 183-205.

[8] C. POMERANCE, *Analysis and comparison of some integer factoring algorithms*, in Computational Methods in Number Theory, H. W. Lenstra, Jr. and R. Tijdeman, eds., Math. Centrum Tract 154, 1982, pp. 89-139.

[9] ———, *The quadratic sieve factoring algorithm*, in Advances in Cryptology, Lecture Notes in Computer Science 209, 1985, pp. 169-182.

[10] R. RIVEST, A. SHAMIR AND L. ADLEMAN, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM, 21 (1978), pp. 120-126.

[11] C. P. SCHNORR AND H. W. LENSTRA, JR., *A Monte Carlo factoring algorithm with linear storage*, Math. Comp., 43 (1984), pp. 289-311.

[12] R. D. SILVERMAN, *The multiple polynomial quadratic sieve*, Math. Comp., 48 (1987), pp. 329-339.

[13] D. WIEDEMANN, *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory, 32 (1986), pp. 54-62.