CUCS-283-87

Parallel Computers, Number Theory Problems, and Experimental Results[1]

Mark Lerner

Computer Science Department
Columbia University

Abstract

This paper discusses the number theoretic problems of primality testing and factorization. It presents algorithms for these problems, and reports on six implementations. Original work is presented for three machines (DADO [Stolfo 83], the Intel Hypercube [Hypercube 86] and the Sequent Balance [Balance 86]). DADO runs primality testing and trial-divisors factorization. The Balance and Hypercube execute factorization by elliptic curves [Lenstra 86]. Work by other researchers is due to [Batcher 80, Pomerance 86, Silverman 86a, Wunderlich 86]; these implementations use a network of workstations, a special-purpose parallel sieve machine, and the Massively Parallel Processor (MPP). The 16K/processor memory of the current DADO2 machine was too small to execute the elliptic curve program, though it was ample for trial-divisors factorization.

1. Introduction

Section 2 presents algorithms for factorization and primality detection. Section 3 gives a description of machines that run these algorithms, and section 4 describes the algorithm implementations. Section 5 provides experimental results.

One observation from these implementations is the importance of matching the program's communication requirements with the architecture's topology and bandwidth. For example, the Ethernet-based "batching network" of [Silverman 86a] uses a different variation of the quadratic sieve than the pipelined hardware of [Pomerance 86]; the MPP [Wunderlich 86] utilizes a different algorithm (continued fraction, CFRAC).

Factorization programs, due to their long running time, have special reliability requirements; these can be provided by algorithm-based fault tolerance [HuangAbraham 84], as well as hardware [Carter 84] or software [Pradhan 86] methods. Reliability is thus an important aspect of factorization algorithms, though it is not the topic of the current paper.

2. Algorithms for Factorization and Primality

Two major problems in number theory which have practical applications are the recognition of prime numbers, and the factorization of composite numbers. The methods developed for these problems include [Dixon 84, Knuth 81, Pomerance 86, Rabin 80, Riesel 85, Silverman 86a, Wunderlich 86]. Primality tests include the Fermat test and the Solovay-Strassen test. Methods for factorization include the continued fraction method (CFRAC), the quadratic sieve method (QS), and the elliptic curve method. An introduction to the material can be found in [Schroeder 86].

Factorization is considerably harder than primality testing. The best known algorithms for factorization use nearly exponential time: $e^{O(\sqrt{log(n)\, loglog(n)})}$. On the other hand, primality recognition is easy; the best deterministic primality testing routines are almost polynomial: $(log\ n)^{O(logloglog(n))}$.

2.1. Primality Testing

The easier of the two problems is recognition of prime numbers. Under the extended Riemann's hypothesis this is computed in polynomial time [Miller 76].

A particularly helpful theorem on prime numbers is Fermat's Little Theorem: for prime p and integer b not divisible by p, the following congruence holds: $b^{p-1} \equiv 1 (mod\ p), p \nmid b$.

To test if a number w is prime, check if $b^{w-1} \equiv 1 (mod\ w)$ for randomly selected values of b. Many such tests can be performed in parallel. If a b can be found for which the congruence does not hold, then the number w has been demonstrated to be composite (i.e., not prime). By execution of many independent trials of the same algorithm a decision (prime/not-prime) can be made with exponentially increasing certainty, although certain rare integers (called the Carmichael numbers) also pass the test. Nevertheless the Fermat test is studied empirically because it is easy to compute; indeed, "In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a 'correct' algorithm" [Abelson 85].

The various provably correct primality tests include the deterministic algorithm of Adelman-Pomerance-Rumley [Adelman 83]. The probabilistic methods of Solovay-Strassen and Rabin [Rabin 80] are practical and have negligible errors. The APR algorithm is deterministic, and for all large n it will terminate within $(log\ n)^{O(logloglog(n))}$. A probabilistic expected polynomial time algorithm is given by [Goldwasser 86].

2.1.1. Finding the First N Primes; Possible Speedup Beyond Sieve of Eratosthenes

In various applications the first N primes are needed. The Sieve of Eratosthenes finds all prime numbers between 2 and N [Schroeder 86]. The Sieve works by initializing an array, called the sieve, to all the possibly-prime numbers less than N. It repeatedly finds the smallest number (which is necessarily prime) and removes all

[1]Supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165 and the New York State Science and Technology Foundation NYSSTFCAT(84)-15, as well as grants by AT&T, HP, and IBM.

Research facilities for the Intel Hypercube and Balance Sequent were provided by the University of Colorado at Boulder and the National Science Foundation under grant NSF-1537669. Facilities at Columbia University Computer Science Department house the DADO computer, designed and constructed at Columbia.

To appear in the Third International Conference on Supercomputing, May 15-20, 1988.

multiples of this number from the sieve. The lower bound for the running time of this algorithm [Pritchard 81] is *$O(N \log \log N)$ additions* with storage as low as *$O(\sqrt{N})$ bits*.

As compared with the Sieve of Eratosthenes, the sublinear additive sieve [Pritchard 81] has lower complexity, $O(N/\log \log N)$ addition operations. For large N it may provide better practical performance as well.

I describe here a simple new algorithm that does better when massively parallel hardware is available. The system would be constructed from many thousands of simple components. Each executes a primality test with a small amount of control circuitry. This massively parallel algorithm, shown in figure 2-1, can also be used to find prime numbers in an interval $[M,N]$, $M >> 1$, or when N is extremely large and not every prime in the interval must be identified.

1. Store possible primes into unique processors,
2. Execute a fast primality test in each processor,
3. Read out the results.

Figure 2-1: Parallel Primality Detection

The data can be generated within the processors (thus no communication cost) by a simple function of the iteration number and the processor ID; for example, the value tested during iteration i in processor N is $2 \times (I+N)+1$; more powerful functions can be used to exclude products of the small primes.

The largest value of N for which the process is expected to obtain a prime number in execution of step 2 can be calculated from the density of prime numbers ($P(N) = \lim_{n \to \infty} N/\log(N)$) under the assumptions of *(a)* 10^6 processors, and *(b)* the procedure is useful only when interprime distance is less than the number of processors, to assure a new prime is discovered on each iteration. The maximum size occurs at $(X+10^6)/\log(X+10^6) = X/\log(X)+1$; thus $X = e^{10^6}$.

2.2. Factorization

Factorization, naively or by sophisticated methods, has received considerable attention because its difficulty is the basis of many data encryption techniques. A naive technique, called the method of trial divisors, is to simply divide a number by the first N primes. Powerful methods use quadratic congruences or elliptic curves.

The congruence methods (CFRAC, QS) solve the congruence $X^2 \equiv Y^2 \bmod N$ to obtain factors $gcd(X+Y, N)$ or $gcd(X-Y, N)$. The elliptic curve method is a recent technique developed by [Lenstra 86].

2.2.1. Trial Divisors

The technique of trial divisors is a simple method, and more powerful methods (CFRAC, for example) depend on it. It factors N by many divisions of small prime numbers. There is a good chance that some of the primes will divide N. If the numbers fail to factor N, the primes may be used as seeds to generate additional possible factors, as described in [Riesel 85]. The computation k^*p+1 (p prime) for small values of k will generate possible factors that can be tested as trial divisors with no additional data. This method has been demonstrated on the DADO machine, see section 4.4.

2.2.2. Quadratic Sieve (QS)

Fast factorization by [Silverman 86a, Pomerance 86] solves the above congruence by solution of quadratic equations to generate possible solutions. The basic algorithm as described in [Silverman 86b] works by:

1. Selection of a factor base, FB
2. Solution of a quadratic equation for all primes $p_i \in$ FB
3. Initialization of a sieve array over [-M,M)
4. Sieving, in which $\forall p_i \in$ FB, $\log(p_i)$ is added to the sieve array according to the roots found in step (2)
5. Scanning of the sieve array for the roots of the quadratic equation of step (2)
6. Collection of factorizations, and testing in the congruence.

Several algebraic methods have been described to make this process efficient. These include the selection of quadratic polynomials and bounding the size of the factor base. Other methods, such as the "large prime variance" further accelerate the process [Pomerance 86].

2.2.3. Continued Fraction (CFRAC)

The continued fraction method, like QS, tries a slightly different way to construct pairs (X, Y) that solve $X^2 \equiv Y^2 \bmod N$. The CFRAC method uses trial divisions at a key step. Thus it is well suited for a machine that performs parallel trial divisors. The MPP [Wunderlich 86] is one such machine.

CFRAC finds (X, Y) by generating *sequences* $\{Z_i\}$ and $\{Q_i\}$ such that $Z_i^2 \equiv Q_i \bmod N$, and then determines the set $Q = \{Q_{i_1}, Q_{i_2} \ldots Q_{i_k}\}$ such that $\prod_{j=1}^{k} Q_{i_j} = Y^2$. The Q_i are trial divided and the subsequent processing uses only the values that completely factor. For $X = \prod_{j=1}^{k} Z_{i_j} \bmod N$ the method may obtain a solution to the congruence $X^2 \equiv Y^2 \bmod N$.

The difficulties with the CFRAC are to determine $\{Z_i\}$, $\{Q_i\}$, and to determine Q. The first problem solved by computation of the continued fraction expansion, $\phi = (P+\sqrt{D})/D$, $(P, D, Q \in I, D>0)$.

The second problem is solved by establishing a prime base $P = \{p_0, p_1 \ldots p_k\}$, where p_i are distinct primes and each Q_i is completely divisible by primes in the base, that is $Q_i = \prod_{j=0}^{k} p_j^{\alpha_{ij}}$, $\alpha_{ij} \in Z$. If enough Q_i are known then linear dependencies can be found among the α_{ij} values. These dependencies are used to construct Y^2.

2.2.4. Elliptic Curve

A method for factorization which is not based on the quadratic congruence is the elliptic curve. This method has a running time that depends on the size of the prime factors of n. According to its inventor W. H. Lenstra Jr., the method works by

> select[ion] [of] a random pair (E, P), where E is an elliptic curve over $\mathbb{Z}/n\mathbb{Z}$ and P is a point on $E(\mathbb{Z}/n\mathbb{Z})$. Next one calculates $P_k = \text{lcm}\{1, 2, \ldots, k\} \cdot P$ for k = 1, 2, ... and one looks whether P_k reduces to the zero element of $E(\mathbb{Z}/n\mathbb{Z})$ for some non-trivial divisor d of n. [Lenstra 85]

Parallelization of the elliptic curve can be accomplished in two ways. The first is use of multiple curves, and the second is to accelerate the operations on each curve. Speedup linear in the number of processors can be accomplished by use of many processors. Each of the processors can be accelerated further by providing pipelined components.

By factoring with several curves one can exploit the probabilistic version of the elliptic curve method. This is mentioned in [Lenstra]: "Draw three elements $a, x, y \in \mathbb{Z}/n\mathbb{Z}$ [and factor each with one curve]." This process can be repeated until a non-trivial divisor of n is found.

2.3. Randomized Algorithms

Random numbers are needed by the probabilistic tests and the elliptic curve method. Statistical independence [Knuth 81] is needed between each sequence. In the parallel environment this independence may be challenged by variations in computational load or system reconfiguration. This section describes literature and methods to address this problem.

The first method of generating random numbers uses a single source, for example, a central host processor to supply each processing node with random numbers. This requires a communication channel to distribute the values. It is resilient against changes in load because the random numbers are distributed as needed. However, it may be impossible to repeat the sequence, as required for debugging in particular.

The second method places the random generator program into each processor, and initializes each copy with a different seed. Because the seeds are different, the sequences will be different; the acceptability of this arrangement is application dependent. A difficulty occurs if several processors produce the same sequences, except out of phase with each other. Unwanted correlations between the sequences might thereby result.

A variant of the above approach guarantees that sequences do not overlap. A preprocessing step runs the generator for sufficiently many cycles. A number N of intermediate seed values are saved, where N is the number of parallel processors to use. The seeds are saved at an interval larger than the number of primes that will be needed by any processor. One of these saved seed values is subsequently assigned to each processor to initialize the random number generator. In this variant, the system reconfiguration may change the distribution of random values. Repeatability, however, is available within each processor.

Parallel generation of random numbers is discussed in [Kalos 86]. Methods include Tausworthe generators, composite generators, and Lehmer trees.

3. Machine Descriptions

As stated in the introduction, the six machines (DADO, Hypercube, Balance, workstation networks, parallel pipeline, and MPP) factor numbers in different ways. The DADO machine executes a primality test and trial divisors factorization. The Hypercube and Balance machines run the new elliptic curve method. A network of 10 SUN workstations [Silverman 86b] has factored the 87 digit number $2^{178}+1$ in a week [Silverman 86a] by use of the QS algorithm. Even faster results are obtained with special-purpose sieving hardware [Pomerance 86]; this carefully tuned pipeline machine can factor a 100 digit number in a month and is both fast and cost-effective. The Massively Parallel Processor (MPP) supports an implementation of CFRAC [Wunderlich 86].

3.1. Batching Network

Silverman uses a network of 8 - 10 SUN-3/75 workstations with Ethernet interconnect. The stations can execute independently, and can also be arranged into a logical star configuration. Indirect synchronization is achieved between the host and each satellite. The host blocks, awaiting output by a satellite or a timer interrupt [Silverman 86a].

Provided the central "hub" remains functional the machine will not fail, though its performance may be degraded. If the "hub" fails the system must be restarted, and checkpoint information retained on stable storage allows a restart from the previous iteration.

3.2. Parallel Pipeline Sieve

Pomerance, Smith and Tuler [Pomerance 86] have proposed special-purpose sieving hardware for QS factorization. The use of such hardware should solve the problem in a cost effective manner (recalling it is almost exponentially hard). Their design is carefully customized for the problem.

The special purpose hardware includes a pipe component. This hardware is constructed of block processors, each with about 2^{16} bytes of memory. These are connected with a smart I/O buffer. The pipe can flow information both forward and backward. This unit performs sequences of well-defined arithmetic operations, and two progressions can be in progress simultaneously.

The interconnection network is both a pipeline and a bus. The pipeline is used to pass sieve elements. The bus is for global initialization of instructions, and for control. The capacities of both pipe and bus are designed for the expected demands of sieving, generation of polynomials, and communication of partial results.

3.3. Massively Parallel Processor

The MPP consists of 16384 PEs, each with a 1K bit memory, operating in lock-step. It was intended for image processing. The hardware consists of an I/O control unit (IOCU), a PE control unit (PECU), and a main control unit (MCU). The PECU is microcoded. The MCU can invoke PECU parallel routines. A VAX serves as a frontend. See [Batcher 80, Schneck 87] for more information on this machine.

3.4. Tree Machine — DADO

DADO [Stolfo 83] is a binary tree-structured multiprocessor architecture incorporating thousands of moderately powerful processing elements (PEs). Operational is a DADO2 computer configured with 1023 PEs and 16 megabytes of RAM; this machine is approximately the same hardware complexity as a VAX-11/750. Each PE consists of a programmable microcomputer with a modest amount of local memory (in the range of 16K bytes) and a specialized I/O chip designed to accelerate inter-PE communication. A full-scale production version of the machine may comprise many thousands of processors implemented in VLSI technology.

The execution modes of a DADO PE are unique. Each PE may operate in SIMD (*S*ingle *I*nstruction, *M*ultiple *D*ata stream) mode [Flynn72] whereby instructions are executed as broadcast by some ancestor PE in the tree. Alternatively, a PE may operate in MIMD (*M*ultiple *I*nstruction, *M*ultiple *D*ata stream) mode by executing instructions from its local RAM. Such a PE may, however, broadcast instructions for execution by descendant PEs in SIMD mode.

3.5. Shared Memory — Sequent Balance

The Sequent Balance 8000 [Balance 86] is a shared-memory multicomputer. Several processor boards (eight on this machine) and a number of memory boards are connected by a high-speed bus. Each board runs an autonomous UNIX system. Process synchronization is through signals or semaphores as provided by Unix. The architecture is optimized to diminish bus contention by use of cache memory. Cache consistency is maintained by having one primary copy for each cached memory location; multiple readers are updated when the primary is written. In this manner, semaphores do not put unnecessary load on the bus.

3.6. Message-Passing — Intel Hypercube

The Intel Hypercube [Hypercube 86] is a message-passing architecture with a hypercube interconnection network. The machine used for this experiment has 32 processors. Each processor can communicate with the 4 adjacent corners of the five dimensional hypercube. Each processor supports multiple processes and communication channels. Message-passing routines allow communication between other processors and the host processor.

4. Implementations

4.1. Batching Network (QS)

The Mitre corporation has implemented the quadratic sieve in [Silverman 86b, Silverman 86a]. These implementations of the quadratic sieve make use of a network of workstations. By use of *multiple polynomials*, the parallel implementations achieve speedup almost linear in the number of processors. Each processor is of conventional design.

Two methods of parallelization are reported. Both use only standard Ethernet hardware and UNIX communication.

The first method has a standalone version of QS on many machines with different starting values. Each machine does its own $Q(x)$ factorization and maintains its own restart files on independent disks. Each machine runs independently and a N-fold speedup with N machines results.

The second method uses a star topology of a central host logically connected to satellite processors. The host provides special functions, and the satellites sieve for possible factors. This helps to assure reliability in the face of processor failures. The host responsibilities are:

- The host computes the sieve polynomials, and keeps a stack of values to keep the satellites busy
- The host stores factorizations as reported by the satellite processors
- The host monitors the satellite processors. When a satellite becomes available, it is loaded with software, a partial factorization, and a sieve interval. It then begins factorization independently from the other workstations.

The host prepares fresh polynomials while the satellites sieve. The polynomial selection can also be parallelized. It needs to be done quickly and efficiently, since "with an efficient algorithm for doing this [computation of (1/2A) mod p] such as the extended Euclidean algorithm, one must typically do [it] thousands of times when changing polynomials" [Silverman 86b].

The efficiency of this approach is improved by several methods. These include *(1)* estimation of logs to allow use of a sieve array built of single-byte cells, *(2)* use of a sufficiently large wordsize to store the factor base without use of multiple precision arithmetic, and *(3)* acceleration of the sieve by small-prime and large-prime variation. In particular, the algorithm is highly dependent on fast multiplication and division. For example, there is an order of magnitude speedup when the processor is changed from a 16x16 bit multiply to a 32x32 bit multiply.

The PEs note if they should report a result. They report, if appropriate, any factors which are sent to the host processor when time is available.

4.2. Parallel Pipeline Sieve (QS)

Pomerance's implementation of the quadratic sieve uses 5 stages, each constructed from off-the-shelf hardware.

Stage 1 preprocesses the data to solve a congruence and manipulate the factor base. Stage 2 initializes the sieve to prepare a polynomial and data. This is done in a sequential piece of hardware. In stages 3 and 4 it breaks the polynomial interval into several pieces. It then uses several purpose parallel pipe units to sieve in parallel. Stage 5 computes linear dependencies. This algebra can require large amounts of storage. Alternatively it can be solved by "sparse encoding of vectors and quick elimination of large primes by Gaussian elimination," [Pomerance 86] or other techniques. Pomerance uses a CDC Cyber computer to solve this by an elimination process.

Unlike [Silverman 86a], fault tolerance is not described in the current design. If the machine consists of 15,000 components, each with a failure rate of 10^{-7}/hour, then at least one fault is expected to occur during the month required to factor a 100 digit number. This assumes that no errors occur during communication between the phases of computation, and is optimistic because of the fairly fast $70ns$ components. Factorization of a 150 digit number requires a full year, and many faults would probably occur during this time. Error-correcting components, particularly memory, can improve the reliability considerably.

4.3. MPP (CFRAC)

The work of Wunderlich and Williams [Wunderlich 86] implements the continued fraction method (CFRAC) on the MPP machine. The MPP was built for image processing. It exploits 16K small processors.

The implementation is to generate pairs (Q, A) and perform trial factorization over P of the Q's. The formulas

$$P_{k+1} = q_k Q_k - P_k,$$
$$Q_{k+1} = (D - P_{k+1}^2)/Q_k,$$
$$q_{k+1} = [(P_{k+1} + d)/Q_{k+1}] \quad (i=0,1,2,\ldots)$$

are expanded in parallel on the MPP by computing tuples $S_i = ((-1)^{n_i} Q_{n_i}, P_{n_i}, A_{n_i-1}, A_{n_i-2})$ where $(i = 1, 2, 3, \ldots, 16384)$.

These values are loaded into the ARU of the MPP to factor the Q values. The factorization is done by trial divisors in parallel. Each PE stores a unique Q value. Each PE also stores the same sequence of 15 prime numbers. These primes are divided into the Q values simultaneously. Pipelining further increases performance. (See section 5.4 for a description of the trial divisors task on the DADO computer.)

4.4. Tree Machine (DADO) with Trial Divisors

Observe that all divisions can be performed independently in parallel. All divisions are (nearly) equally likely to divide n. The parallelization is:

1. Store unique primes p into each PE
2. Store n into all PEs
3. Each PE divides values of p and $k*p+1$ into n
4. Print the values which divide n with zero remainder.

This trial-division technique can accelerate many algorithms. For example, it can be used as a subroutine for the quadratic sieve. A similar technique has been used by [Wunderlich 86] in an implementation of the continued fraction method (see above).

4.4.1. Primality Testing

To execute probabilistic primality tests and experimentally address the resource utilization question, a load balancing scheme was developed. Each processing element (PE) stores software for a randomized algorithm, and a unique processor identification number (*PE_ID*). The software consists of a random number generator, any randomized routine (in this case a probabilistic primality-tester), and the communication/control routines necessary to coordinate activity with the host. A decimal arithmetic package (modulo 255 arithmetic) permits storage and arithmetic on large numbers.

The PEs are organized into clusters. Each cluster is the same size. The size is an execution-time parameter, and was expected to be important since larger clusters can compute more iterations in the same amount of time. Thus the testing of a number can be completed more rapidly in a larger cluster. On the other hand a cluster that is too large will necessarily result in wasted computer time. For example, if 5 iterations are sufficient to find a disproof of primality, then the use of 10 PEs is wasteful.

The other two parameters are the "number of iterations for acceptance" (ACCTRIALS) and the "number of iterations between communication" (COMMTRIALS). The first is the number of *passes* of the probabilistic test required to certify the number as "probably prime." Larger values of ACCTRIAL cause more iterations, therefore greater certainty that an accepted number is prime. The value of ACCTRIAL is a property of the primality test, not a parameter to the resource allocation.

The host periodically checks the DADO tree to read out results and provide additional data. Excessive checking may be wasteful because of the communication it causes. Infrequent testing, likewise, is also wasteful because many processors may be underutilized. Therefore it is of interest to check various values of COMMTRIAL to find the best way the machine performs.

Resource utilization is improved (on this synchronously communicating prototype) by balancing the pre-processing done on the sequential host, with parallel processing done in the DADO. The host program first runs one test (either trial divisors or a probabilistic test) to quickly discard the surely composite numbers. The possible primes that remain are sent into the DADO machine.

The program flow is:

1. Read the # of PEs per cluster, the # of iterations for acceptance (m), and the interval between communication (n).
2. Initialize the DADO machine to this configuration.
3. Generate test numbers which are likely to be prime. For example, exclude integers divisible by $i < 500$.
4. Store a unique test number into each cluster (the same number is placed into each PE of the cluster).
5. Run n iterations in each PE.
6. Read out the results:
   a. Proof the number is composite, or
   b. m trials complete (conclude the number is prime), or
   c. Not yet known, run another n trials.
7. The host generates more likely primes, while DADO runs the randomized algorithm.
8. Update statistics and repeat from step 4.

4.5. Shared Memory — Sequent Balance (Elliptic Curve)

A program for elliptic factorization has been implemented in the C language. This program has been executed on the DEC VAX 750, Sequent Balance, and Intel Hypercube. Both the Sequent and the Intel showed a linear speedup in the number of processors.

The first source of parallelism (as mentioned by Lenstra) is to pick many elliptic curves randomly, and try each in parallel. This is medium-grain parallelism. The following experiments exploit this parallelism.

A second source of speedup, fine-grain parallelism, was not implemented yet could be exploited within the medium-grain program. For example the evaluation of:

$$\prod_{r=2}^{w} r^{e(r)}$$

can be performed in parallel. A pipeline or vector processor can evaluate this effectively.

For example, a code segment from one implementation of the elliptic curve method executes the loops:

```
for i:=1 to 2^k-1 do ExpSpecMod(v,i,m)
for i:=1 to 2^k-1 do Inverse(u,m)
for i:=1 to 2^k-1 do Remainder(u1,a)
for i:=1 to 2^k-1 do Remainder(u2,a)
```

The above can be parallelized as follows, as there are no side effects in this particular code.

```
for i:=1 to 2^k-1
  PAR_BEGIN
    ExpSpecMod(v,i,m)
    Inverse(u,m)
    Remainder(u1,a)
    Remainder(u2,a)
  END
```

4.5.1. Sequent Balance Implementation

The Sequent Balance is a shared memory computer. The primary source of parallelism is a *m_fork* command. For the factorization task one processor does all initialization, including the computation of the factor base. Many copies of the factorization routine are then created by the *m_fork* call. The Sequent's shared memory includes a sizable cache to decrease bus contention for shared structures such as the factor base. The control structure is listed in figure 4-1; comments are italicized.

The *m_fork* routine creates one copy of the factorization routine (*attempt*) in each processor. The *m_next* routine provides a globally controlled counter. This provides a simple way to prevent redundant calculations. In this program, the value is used to initialize the random number generator in each processor, and also to control the total number of iterations performed.

When a processor finds a factorization it enters a *critical section* by use of the *m_lock* and *m_unlock* routines; these allow it to set the shared variable *done* to true. Other processors will terminate processing when they poll this variable. It is also be possible to terminate those processors asynchronously.

```
main()
{ ...
  seed=p2(ml,ml*2);                     Obtain starting point.
  basisdyadic(seed,b);                  Compute basis vector.
  m_fork(attempt,n,ml,numatt,b);        Start parallel trials.
  ...
}

attempt(n,ml,numatt,b)
int *n, ml, numatt, b;
{ ...
   me=m_get_myid()                      Identify processor uniquely.
   for(;;)
   { if(done)return;                    Check if any processor done.
     i=trialno=m_next();                Get next global trial.
     ...
     if(answer)                         Set done flag if an answer
     { m_lock();
       done=TRUE;
       m_unlock(); } } }
```

Figure 4-1: Shared Memory Control Structure

4.6. Message-Passing — Hypercube (Elliptic Curve)

The implementation on the Hypercube uses two programs: a host program and a node program. The host program starts the processors by sending a data stream to each one. The processors communicate *asynchronously* with the host. When a processor finds a factorization it reports to the host. The host then kills all the other processes.

Communication between the host and the nodes is through a well-defined set of communication codes. These are declared through a .h file, and provide a clean interface between the different processes. The clean interface is of great help in the programming of a message-passing machine, due to the complexity of message-passing in a multiple-CPU environment. The codes are listed in figure 4-2, and the main loop of the host code is in figure 4-3.

The hypercube implementation is shown in figure 4-4. The italicized routines provide for communication and control by invokation of system functions in a structured manner.

The host initiates the program by loading the node program into the hypercube. The host then sends parameters to the cube *(store_parms)*, sends a start message to the cube *(start_cmd)*, and then enters the loop shown above to read answers from the cube *(read_cube)*. The host echoes the data sent from the cube.

Independent node executions in each processor read parameters from the host, wait to receive a start message, and repeatedly run the factorization routine *attempt*. When a factor is found the *ANS_OK* message is transmitted to the host.

```
#define MSG_M1          101 /* Sending M1 value (factor base)      */
#define MSG_B           102 /* Sending B value (expansion base)    */
#define MSG_ATT         103 /* Sending number of attempts per pe  */
#define MSG_N           104 /* Sending N, the number to factor     */
#define MSG_DO_CMD      001 /* Execute routine (buff can id rtn)   */

#define MSG_RES_CLK     106 /* Reset clock                         */
#define MSG_STA_CLK     107 /* Start clock                         */
#define MSG_STO_CLK     108 /* Stop  clock                         */
#define MSG_STA_RTN     109 /* Start routine                       */
#define MSG_SEND_CLK    110 /* Send clock to host                  */
#define MSG_SEND_ANS    111 /* Send answer to host                 */

#define ANS_FAIL        501 /* Routine failed                      */
#define ANS_SUCC        502 /* Routine succeeded                   */
#define ANS_NOTE        503 /* Comment information received.       */
#define ANS_ABORT       504 /* Program aborted abnormally.         */
```

Figure 4-2: Hypercube communication codes

```
read_cube()
{ got_answer=FALSE;
  do
  { while(!probemsg(ALL_NODES));    /* Wait for a message */
    recvmsg(chan,&msg_type,msg_buf,MSGSIZ,
               &msg_cnt,&fr_node,&fr_pid);
    switch(msg_type) {
    case ANS_FAIL: nodes_running--;
                   break;
    case ANS_SUCC: nodes_running--;
                   printf("%s from node %d pid %d.\n",
                                msg_buf,fr_node,fr_pid);
                   got_answer=TRUE;
                   break;
    case ANS_NOTE: printf("%03d   %s\n",fr_node,msg_buf);
         }
 while(nodes_running && !got_answer);
}
```

Figure 4-3: Message-Passing Generic Host Code

```
Host code:   store_parms();                 Store parameters.
             start_cmd();                   Start command running.
             read_cube();                   Read result from cube.
             lkill(ALL_N,ALL_P);            Done, so kill ...
             lwaitall(ALL_N,ALL_P);         ... and wait.

Node code:   get_parms();                   Receive parameters.
             rcv_start();                   Wait to start.
             attempt(n,m1,numatt);          Attempt to factor.

             attempt(n,m1,numatt,b)
               long m1,numatt,b;
               char *n;
             { ...
               for(i=1;i<=numatt;i++)       Repeat for trials
               { ...                        Do computation
                 if(answer)
                 { ToHost(msg_buf,ANS_OK);
                   break; } } }
```

Figure 4-4: Message-Passing Control Structure

5. Experimental Results

5.1. Batching Network of Workstations

The overall performance of the batching network program is very efficient from a workstation utilization perspective. Silverman finds that:

> utilization of satellite processors is virtually 100 percent efficient ... one can hook up enough satellites to overwhelm the host, but in that case one can implement multiple stars ... and hook the various hosts together. [Silverman 86a].

The program factored a "typical 60 digit number" in less than an hour using 8 processors, in contrast to 6 hours for only one processor. The statistics are reproduced in figure 5-1, which was extracted from more detailed information in [Silverman 86a].

Caron and Silverman have not measured the resource utilization, such as the actual communication, paging, and CPU cycles. There is therefore some uncertainty about precise resource utilization. This stands in stark contrast to the detailed resource analyses presented for the special-purpose hardware approach in [Pomerance 86]. Nevertheless, the speedup is significant.

DIGITS	CPU Time (minutes)	DIGITS	CPU Time (minutes)
69	75	76	480
70	88	77	590
71	135	77	560
72	195	78	650
72	210	78	663
74	345	79	1720
76	425	81	1260

Figure 5-1: Quadratic Sieve Parallel Factorizations

5.2. Parallel Pipeline Sieve

The machine should be able to factor any 100 digit number in 28 days [Pomerance 86]. The basis of the above estimate is that the machine needs 5 seconds to generate a new polynomial, and during this time sieves at the rate of 27,600,000 values/second. This is based upon a 70 nanosecond cycle time, a memory size of 2^{16} in each pipe unit, and 2^4 pipes.

The estimated performance for 100 digit numbers is based upon assumptions of overlap and good hardware utilization. It is expected that the custom-tailored design will result in the efficient utilization required. The hardware has been kept somewhat flexible to accommodate modifications.

5.3. Massive Parallel Processor — MPP

The program factored a 62 digit number in 14 hours (plus time on the CDC for final processing). Use of several variations to CFRAC, such as the large prime variation strategy [Pomerance 86], are expected to improve performance. In conjunction with a CDC 7600 (for the final elimination phase) "we could expect to factor a 60 digit number in a total of less than 35 minutes" [Wunderlich 86].

5.4. Tree Machine (DADO) — Trial Divisors

The following shows the use of trial divisors without generation of possible-primes from prime seeds. In this example it is executed on 10 and 14 digit numbers, using the 1023 node DADO2 computer.

5.4.1. Fermat Test

The Fermat test on DADO machine works both as a benchmark, and also to generate large prime numbers. Several 50 digit pseudo-primes were generated on the DADO machine in about 1 minute each. By use of 2^{10} processors a 1000-fold speedup is obtained, assuming each processor always performs useful work. This can be assured by giving each PE a queue of numbers to test, or by generation of unique numbers in each processing element.

Significantly, by testing a different possible-prime in each processor, the *random sequences do not even have to be different in each processor to find large pseudo-primes*, and thus a linear speedup can be obtained in this process. This is done by storing a different possible-prime number in each processor. By use of *N* processors, the machine can test *N* numbers in the time it would have taken a uniprocessor to test just one number.

5.4.2. Resource Use in the Fermat Test

The effective use of parallelism requires that, for any given possible prime, each Fermat test is independent. A simple model provides the basis for static load balancing, in which the initial allocation of work avoids hot-spots. The host computer provides numbers that are likely to be prime, by virtue of having passed one iteration of a probabilistic (or other) test. The parallel machine then performs additional tests on these pre-filtered numbers.

Due to the synchronous nature of communication on DADO2 it is useful to balance the work allocated to each processor. There is enough information to statically load-balance between the host and the parallel machine; in particular, the prime density function gives the density of prime numbers in a given interval. The convergence behavior of the algorithm is known, as are the rates at which the host and the DADO can conduct a test (see figure 5-3).

The statistics here show various cluster sizes, ranging from 1 to 8 processors per cluster. The results have not been scaled to indicate technology advances, and specialized hardware performs significantly faster. For example, the main computational step of the Fermat test is the *expmod* function. Gallium-Arsenide (GaAs) chips have been marketed for this function, and are significantly faster than the 8-bit simulated arithmetic used here. Likewise a faster "big number" arithmetic package could be implemented with significant performance improvements.

The major bottleneck in performance is the 8-bit processors. Each processor is considerably slower than a VAX 750 (both machines run the same multiple-precision software package). By use of the 10^3 processors in DADO there is nevertheless a factor of 10^2 speedup over the VAX, as shown in figure 5-3.

```
che$ dadoload factor DADO2
Program up!
# pes? #3512primes          File of prime numbers
primes?

Root 3,   Mach=3512.        3512 primes loaded

Number? 5112663011          Factor a 10 digit number
* 17
* 17333
* 17351
Ticks=11, time= 0.715

Number? 88842745142147      Factor a 14 digit number
* 17
* 17333
* 17351
* 17377
Ticks=19, time= 1.235

Number?#                    All done
che$ exit
```

che$ is the system prompt.
virterm communicates between the host and the DADO.

Figure 5-2: Execution of Trial Divisors on Tree Machine

Times to verify PRIMES of varying length.
VAX vs. DADO on 1023 trials.

# digits in prime	VAX sec	DADO sec	Speedup on DADO
2	10	.8	12
3	204	2.0	102
4	306	2.4	127
10	2352	16	147
19	12582	112	112
33	56060	538	104
35	73349	465	157

Figure 5-3: Comparison of DADO with VAX for Fermat test

Figure 5-4 shows the utilization as a function of the number of iterations between communication (inter-communication time), for various cluster sizes and preprocessing. An inter-communication time of 1 indicates that each processor communicates with the host after every execution of the Fermat test. Larger values indicate more iterations between communications, thereby reducing the communication overhead but perhaps leaving some processors underutilized. The details in figures 5-5 and 5-6 show the effect of changing inter-communication time.

Figures 5-5 and 5-6 show the detailed running time and utilization for various values of inter-communication time. The numbers being factored are not listed; instead the index in a common data file is provided. This simply saves space and communication time during program execution.

The graph (figure 5-4) distinguishes between "filtered" and "non-filtered" data. Unfiltered data is randomly chosen odd numbers, whereas filtered data excludes those numbers divisible by small primes. As might be expected, the utilization is better for filtered data than for unfiltered data. Performance is most sensitive to the inter-communication interval. The filtered data shows better utilization than the non-filtered data, as expected.

A cluster size of 1 indicates that each possible-prime is stored in only one PE, and thus only one Fermat-test is performed on that number at any time. A larger cluster size indicates the same possible-prime is stored into each of several processors. Each processor will execute independent Fermat tests on that processor. This can accelerate the testing of a number, since more tests can be performed simultaneously.

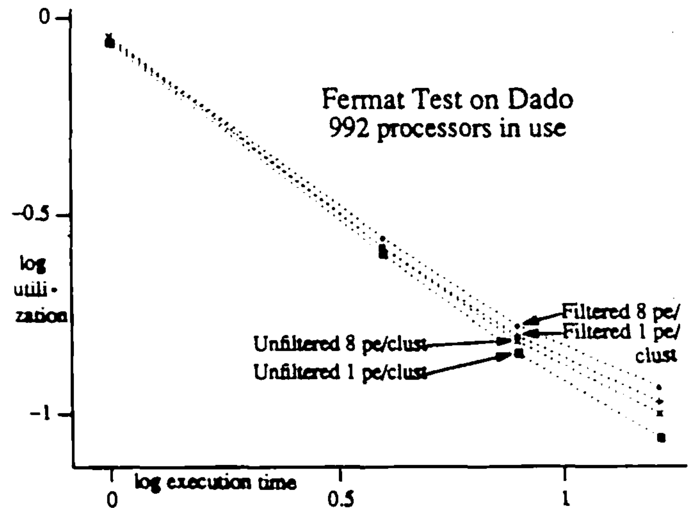

Figure 5-4: Performance of Fermat Test on DADO

The results in figure 5-5 allow only 1 iteration between communications, and show excellent performance. Note, however, that there are 8 processors in each cluster and thus the efficiency may be as low as 1/8 the utilization listed (because it is possible that all processors found the number to be composite, and therefore many of the processors performed redundant work). An improved experiment would record the answers obtained in each processor and determine how many processors found a disproof. The efficiency figure could then be calculated.

Data elements of particular interest have been underlined. Note that the composite numbers in figure 5-6 are processed more quickly than the prime numbers, yet are nevertheless resident in the machine for as long.

The performance is most sensitive to the number of iterations between communication. An inter-communication time of 1 gives the best utilization, about 90% (see figure 5-5). In contrast, when the inter-communication time is 16 (figure 5-6) the quick detection of a composite number results in poor utilization of the processor.

The presence of a prime number in any processor determines the maximum processing rate with the current algorithm, since all programmed trials are executed by all processors in these cases. The result sets therefore show significant variation in utilization. On the other hand, the demonstration of pseudo-primality operates at peak efficiency, unhampered by communication.

In conclusion, both analytic and experimental results show that for this particular randomized algorithm the best cluster size is 1. This occurs because the Fermat test is fairly accurate (except for the Carmichael numbers). This picture might change with a different probabilistic test.

```
** 992 processors in DADO
** 8 processors per cluster
** 1 iterations between communication
** 32 iterations before acceptance
** Number, Prime, Trials, Iters, Cluster,
              MinPeTime, MaxPeTime, HostTime     Utilization
  2 0     8     1     1 103.42 103.42 116.55         88%
  3 0     8     1     2 113.43 113.43 116.55         97%
  4 0     8     1     3 100.17 100.17 116.55         85%
  ...    ...
994 0     8     1    53 110.18 110.18 117.35         94%
995 0     8     1    55 101.86 101.86 117.35         87%
996 0     8     1    56 106.28 106.28 117.35         91%
  1 1   256    32     0 3469.44 3469.44 3713.91      93%
  7 1   256    32     6 3529.76 3529.76 3713.91      95%
  ...    ...
850 1   256    32   110 3240.64 3240.64 3693.47      88%
889 1   256    32    38 3617.12 3617.12 3690.88      98%
993 1   256    32    52 3340.48 3340.48 3678.98      91%
```

Figure 5-5: Fermat Test Output, 1 iteration per communication

```
** 992 processors in DADO
** 8 processors per cluster
** 16 iterations between communication
** 32 iterations before acceptance
** Number, Prime, Trials, Iters, Cluster,
              MinPeTime, MaxPeTime, HostTime     Utilization
  2 0   128    16     1 103.42 103.42 1743.20        6%
  3 0   128    16     2 113.43 113.43 1743.20        7%
  4 0   128    16     3 100.17 100.17 1743.20        7%
  ...    ...
128 0   128    16     4 104.78 104.78 1743.20        6%
129 0   128    16     5 107.58 107.58 1743.20        6%
  7 1   256    32     6 3456.31 3456.31 3486.40     99% a prime
130 0   128    16     7 105.11 105.11 1743.20        6%
131 0   128    16     8 102.25 102.25 1743.20        6%
  ...    ...
972 0   128    16     8  99.97  99.97 1813.59        6%
850 1   256    32     9 3235.44 3235.44 3627.18     98% a prime
973 0   128    16    10 103.35 103.35 1813.59        6%
  ...    ...
916 1   256    32    75 3395.21 3395.21 3627.18     94%
993 1   256    32    30 3370.77 3370.77 3513.65     96%
```

Figure 5-6: Fermat Test Output, 1 iteration per communication

5.5. Elliptic Curve Factorization with Shared Memory — Sequent Balance

The result of running the elliptic curve algorithm on this machine was a linear speedup for seven processors. The graphs (figure 5-7, 5-8) show the average resource utilization for factorization of 284378461123357 and 377525665707083 using various numbers of processors. Larger and harder numbers were not factored because they take too much time (i.e., hundreds of iterations).

This linear speedup is as expected because each execution is independent. The only interaction is in subscript generation, and this is a very small portion of the execution time.

Arithmetic was performed with an extended decimal arithmetic package coded in C. This package uses C `int` variables, and packs either 2 or 4 decimal digits into each variable, depending on word size. On the Sequent Balance the `int` variable is 32 bits. This allows 4 decimal digits per variable.

The extended arithmetic is reasonably efficient, but could be made better by use of either a modulo 2^{32} representation (instead of modulo 10000), or use of assembly code. A more refined extended integer package would make a substantial improvement in practice. However, the purpose of these experiments was to investigate algorithm parallelization.

Programming of the Balance machine was extremely simple. The sequential VAX implementation was completely transportable to the Balance machine (using only one processor). The parallelization of the outer loop took less than an afternoon. No problems developed either with the system software or with debugging of the system. The quality of the system software — as well as the simplicity of the memory paradigm — are the likely reasons for this.

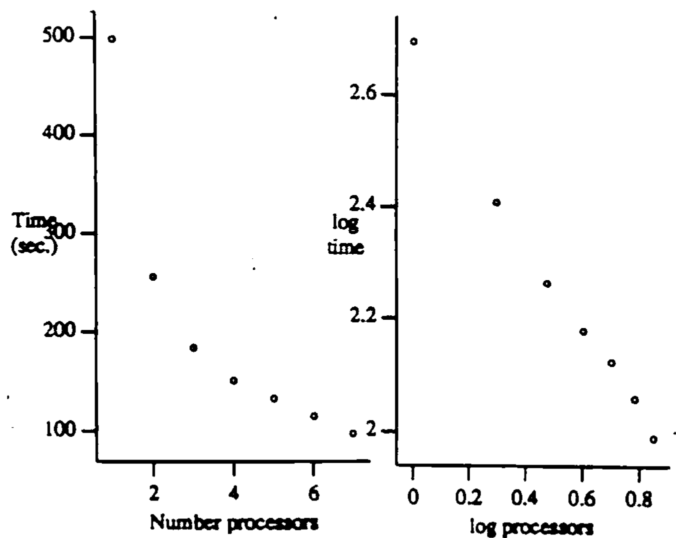

Figure 5-7: Sequent: processors vs time

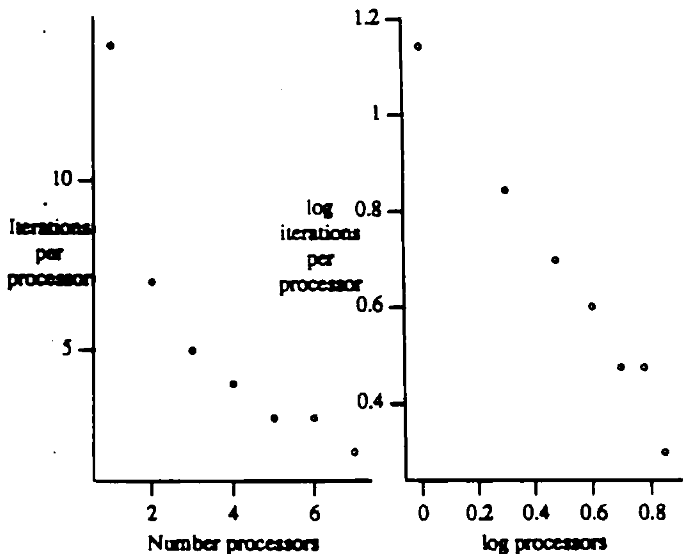

Figure 5-8: Sequent: processors vs iterations

5.6. Elliptic Curve Factorization with Message-Passing — Hypercube

The implementation showed a linear speedup when a sufficient number of processors were used. The graph shows the average resource utilization for factorization using the same numbers as the Sequent. As shown in figure 5-9 the speedup is linear in the number of processors.

The absolute performance can be improved in several ways, though the curve shape is not expected to be affected by such program modifications. First, the default precision of the Intel `int` variables is 16 bits, thus only 2 digits were packed into each variable. This certainly can be improved — for example by use of `long` variables, or perhaps the floating-point coprocessor. Secondly the improved `random` function (instead of the older `rand`) can be used when it is available. Indeed, newer hardware and software have already been installed at many sites.

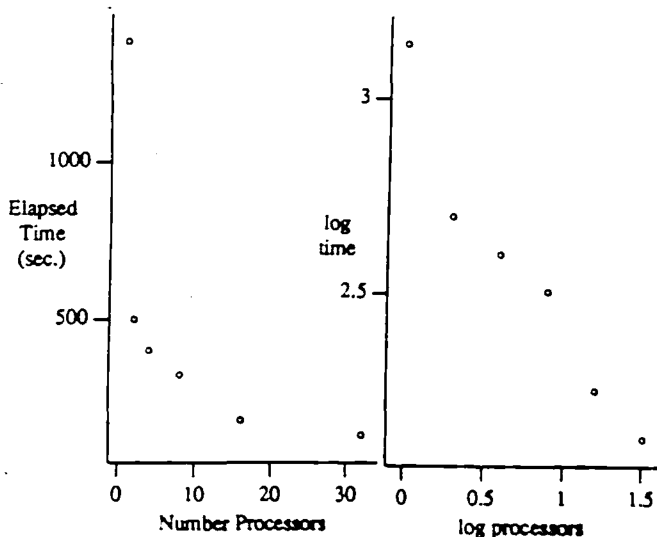

Figure 5-9: Hypercube: processors vs time

The software simulator for the hypercube was very helpful in program development, primarily due to its message logging ability. However there were a few problems with the simulator. In particular, it did not exhibit the same behavior for certain casting and allocation problems. Perhaps this is because the simulator did not capture the exact behavior of the Hypercube. Alternatively, it may be due to running it on a Vax, which is architecturally different from the Intel processors.

6. Conclusion

Modern algorithms and powerful machines, working together, can solve difficult problems quite successfully. The algorithms described include simple division, randomized use of the *expmod* function, and sophisticated sieving methods. Empirical evidence shows speedups on a tree machine (DADO), the Intel Hypercube and the Sequent Balance, as well as an Ethernet connection of SUN stations, the MPP computer, and a special-purpose sieve machine. Several principles emerge.

First, the best parallel method outperforms the best sequential method, due to improved hardware utilization. This practical result is demonstrated by the quadratic sieve on the pipelined sieve unit [Pomerance 86]. All hardware is expected to do useful work, thus little energy is wasted, and the computation time is diminished. This is a special case of the efficiency principles described in [Lipovski 87].

Second, the algorithm should be selected to fit the available parallelism and communication structure. The MPP machine, to utilize its capacity for parallel division, runs the CFRAC method. Moreover, the implementation of an algorithm should be tailored for the particular configuration. This is demonstrated by [Silverman 86b] with two forms of the QS, according to the network configuration. The first runs several copies of the same algorithm with different starting values. The second implementation combines centralized computation of sieve polynomials with satellite computation of factorizations.

Parallel processing is of tremendous importance because it provides orders of magnitude speedup. A wide variety of parallelism can be brought to bear on the problem. Special purpose machines (parallel pipeline sieve, MPP) can be configured to provide both the computational and communication resources in the form used by a particular algorithm. Substantial speedup is also achieved by the general purpose approach, though hardware utilization is not as good. General purpose machines (workstations, Hypercube, Balance) can supply parallelism at both the coarse-grain and fine-grain levels. Massive parallelism (DADO) accelerates key parts of these numeric algorithms.

7. Acknowledgments

I am grateful to Professor Zvi Galil, Professor Gerald Q. Maguire, Jr., and my advisor Professor Salvatore Stolfo for their help and encouragement.

Columbia colleagues, in particular Stuart Haber and Mordechi Yung, helped me to understand the number theory described in the first part of the paper, and guided me toward several references.

Much of the implementation and experimental work was performed by Dave DeMarco and Yoseff Francus, who were graduate students at Columbia University.

The Sequent and Hypercube machines at the University of Colorado (Boulder) were essential for the implementations on those machines. I thank Chairman of Computer Science at the University of Colorado, Lloyd Fosdick, as well as Betty Eskow and Carolyn Schauble.

References

[Abelson 85] Abelson, H., G. J. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, McGraw-Hill Books, Cambridge, Mass, New York, 1985. The MIT Electrical Engineering and Computer Science Series.

[Adelman 83] Adelman, L. M., C. Pomerance, R. Rumley. On distinguishing prime numbers from composite numbers. *Annals of Mathematics* 117:173-206, 1983.

[Balance 86] Osterhaug, A. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., Bearverton, Oregon, 1985, 1986.

[Batcher 80] Batcher, K. E. Design of a Massively Parallel Processor. *IEEE Trans. Computers* C-29:836-840, June 1980.

[Carter 84] Carter, W. C.
A Short Survey of Some Aspects of Hardware Design Techniques for Fault Tolerance.
Technical Report, IBM Research Division, October 84.
IBM RC 10811 #48410.

[Dixon 84] Dixon, J. D.
Factorization and Primality Tests.
American Mathematical Monthly 91(6):333-351, June-July 1984.

[Goldwasser 86] Goldwasser, S., J. Kilian.
Almost All Primes Can be Quickly Certified.
In *Proceedings of the Nineteenth Annual ACM Symposium on the Theory of Computing*, pages 316-329. ACM, 1986.

[HuangAbraham 84]
Huang, K. H., Abraham, J. A.
Algorithm-Based Fault Tolerance for Matrix Operations.
IEEE Trans. Computers C-33:518-528, June 1984.

[Hypercube 86] Intel Corporation.
iPSC Technical Description.
Intel Corporation, U.S.A, 1986.
Order Number 175278-003.

[Kalos 86] Kalos, M. H., P. A. Whitlock.
Monte Carlo Methods.
Wiley, New York, 1986.

[Knuth 81] Knuth, D. E.
The Art of Computer Programming (Volume 2).
Addison-Wesley, Reading, Mass, 1981.

[Lenstra 85] Lenstra, H. W. Jr.
Elliptic Curve Factorization.
February 14, 1985.
Unpublished.

[Lenstra 86] Lenstra, H. W. Jr.
Factoring Integers with Elliptic Curves.
Technical Report, Technical Report, Mathematisch Instituut, Universiteit van Amsterdam, May 12, 1986.

[Lipovski 87] Lipovski, G. J., M. Malek.
Parallel Computing. Theory and Comparisons.
John Wiley and Sons, U.S.A., 1987.
ISBN 0-471-82262-0.

[Miller 76] Miller, G. L.
Reimann's Hypothesis and Tests for Primality.
J. Comput. System Sci. 13:300-317, 1976.

[Pomerance 86] Pomerance, C., J. W. Smith, R. Tuler.
A Pipe-Line Architecture for Factoring Large Integers with the Quadratic Sieve Algorithm.
In *Crypto86*. 1986.
Departments of Mathematics and Computer Science, The University of Georgia, Athens Georgia.

[Pradhan 86] Pradhan, D. K.
Fault Tolerant Computing: Theory and Techniques, Volume 2.
Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Pritchard 81] Pritchard, P.
A Sublinear Additive Sieve for Finding Prime Numbers.
Communications of the Association for Computing Machinery 24, No. 1:18-23, January, 1981.

[Rabin 80] Rabin, M. O.
Probabilistic Algorithm for Testing Primality.
Journal of Number Theory 12:128-138, 1980.

[Riesel 85] Coates, J., S. Helgason (editor).
Prime Numbers and Computer Methods for Factorization.
Birkhauser, Boston, Basel, Stuttgart, 1985.
Progress in Mathematics, Vol. 57.

[Schneck 87] Schneck, P. B.
Supercomputer Architecture.
Kluwer Academic Publishers, Norwell Mass., 1987.

[Schroeder 86] Schroeder, M. R.
Number Theory in Science and Communication.
Springer Verlag, New York, 1986.

[Silverman 86a] Caron, T. R., R. D. Silverman.
Parallel Implementation of the Quadratic Sieve.
In *Crypto86*. 1986.

[Silverman 86b] Silverman, R. D.
The Multiple Polynomial Quadratic Sieve.
1986.
Mitre Corporation.

[Stolfo 83] Stolfo, S. J.
The DADO Parallel Computer.
Technical Report, Department of Computer Science, Columbia University, August, 1983.

[Wunderlich 86] Wunderlich, M. C., H. C. Williams.
A Parallel Version of the Continued Fraction Integer Factoring Algorithm.
In *Crypto86*. 1986.
University of Northern Illinois, and University of Manitoba.