A Parallel GNFS Algorithm with Reliable Look Ahead Block Lanczos for Integer Factorization

Li Xu, Laurence T. Yang, Man Lin
Department of Computer Science
St. Francis Xavier University, NS, B2G 2W5, Canada
{x2002uwf,lyang,mlin}@stfx.ca

Abstract

RSA is a very popular and secure public key cryptosystem, but the security relies on the difficulty of factoring large integers. The General Number Field Sieve (GNFS) algorithm is currently the best known method for factoring large integers over 110 digits. Our previous work on the parallel GNFS algorithm, which integrated the Montgomery's block Lanczos [13] algorithm to solve the large and sparse linear systems over GF(2), is less reliable.

In this paper, we successfully implemented the parallel General Number Field Sieve (GNFS) algorithm and integrated with a new algorithm for solving large and sparse linear system called lookahead block Lanczos algorithm [6]. This new algorithm is based on the look-ahead technique, which can provide more reliability. The algorithm can find more dependencies than Montgomery's block Lanczos method using less iterations. The detailed experimental results on a SUN cluster will be presented as well.

1 Introduction

Today, Rivest-Shamir-Adleman (RSA) algorithm [17] is the most popular algorithm in public-key cryptosystem. It also has been widely used in the real-world applications such as: internet explorer, email systems, online banking, cell phones and many other places. The security of this algorithm mainly relies on the difficulty of factoring large integers. Many integer factorization algorithms have been developed. Examples are: Trial division [18], Pollard's p-1 algorithm [15], Lenstra Elliptic Curve Factorization (ECM) [10], Quadratic Sieve (QS) [16] and General Number Field Sieve (GNFS) [1, 2, 3, 12] algorithm. GNFS is the best known method for factoring large composite numbers over 110 digits so far.

Although GNFS algorithm is the fastest one so far, it still takes a long time to factor large integers. In order to reduce the execution time, one natural solution is to use parallel computers. The GNFS algorithm contains several steps. The most time consuming step is sieving which is used to generate enough relations. This step is very suitable for parallelization because the relation generations are independent. Another possible step is, in GNFS, the Montgomery's block Lanczos [13] algorithm. It is used to solve large and sparse linear systems over GF(2) generated by the GNFS algorithm. The disadvantage of this block Lanczos is less reliable. The lookahead block Lanczos [6] has overcome this disadvantage and improved the reliability of block Lanczos algorithm. In this paper we have successfully implemented the lookahead block Lanczos algorithm together with the GNFS algorithm.

The left of the paper is organized as follows: we first briefly describe the original GNFS algorithm in section 2. Then we present two block Lanczos algorithms, namely the Montgomery's block Lanczos [13] and the lookahead block Lanczos algorithm [6] in section 3 and 4 respectively. Section 5 shows the detailed implementation and corresponding parallel performance results.

2 The GNFS Algorithm

The General Number Field Sieve (GNFS) algorithm [2, 3, 12] is derived from the number fields sieve (NFS) algorithm, developed by Lenstra et al [11]. It is the fastest known algorithm for integer factorization. The idea of GNFS is from the congruence of squares algorithm [9].

Suppose we want to factor an integer n where n has two prime factors p and q. Let's assume we have two integers s and r, such that s^2 and r^2 are perfect squares and satisfy the constraint $s^2 \equiv r^2 (mod\ n)$. Since $n = pq$, the following conditions must hold [2]:

$$\begin{aligned} pq|(s^2-r^2) &\Rightarrow pq|(s-r)(s+r) \\ &\Rightarrow p|(s-r)(s+r) \text{ and } q|(s-r)(s+r). \end{aligned}$$

We know that, if $c|ab$ and $gcd(b,c) = 1$, then $c|a$.

So *p, q, r and s* must satisfy $p|(s\text{-}r)$ or $p|(s+r)$ and $q|(s\text{-}r)$ or $q|(s+r)$. Based on this, it can be proved that we can find factors of n by computing the greatest common divisor *gcd(n,(s+r))* and *gcd(n,(s-r))* with the possibility of *2/3* (see [2]).

Therefore, the essence of GNFS algorithm is based on the idea of the factoring n by computing the *gcd(n, s+r)* and *gcd(n, s-r)*. There are six major steps [12]:

1. Selecting parameters: Choose an integer $m \in Z$ and a polynomial f which satisfy $f(m) \equiv 0 \pmod n$.
2. Defining three factor bases: rational factor base R, algebraic factor base A and quadratic character base Q.
3. Sieving: Generate enough pairs *(a,b)* (relations) to build a linear dependence.
4. Processing relations: Filter out useful pairs *(a,b)* that were found from sieving.
5. Building up a large and sparse linear system over GF(2) and solve it.
6. Squaring root, use the results from the previous step to generate two perfect squares, then factor n.

3 Montgomery's Block Lanczos Algorithm

Montgomery's block Lanczos algorithm was proposed by Montgomery in 1995 [13]. This block Lanczos algorithm is a variant of standard Lanczos method [7, 8]. Both Lanczos algorithms are used to solve linear systems. In the standard Lanczos algorithm, suppose we have a symmetric matrix $A \in R^{n \times n}$. Based on the notations used in [13], the algorithm can be described as follows:

$$\begin{aligned} w_0 &= b, \\ w_i &= \mathbf{A}w_{i-1} - \sum_{j=0}^{i-1} \frac{w_j^T \mathbf{A}^2 w_{i-1}}{w_j^T \mathbf{A} w_j}. \end{aligned} \tag{1}$$

The iteration will stop when $w_i=0$. $\{w_0, w_1, \dots w_i\}$ are basis of $span\{b, \mathbf{A}b, \mathbf{A}^2b, \dots\}$ with the properties:

$$\forall 0 \le i < m, \quad w_i^T \mathbf{A} w_i \neq 0, \tag{2}$$

$$\forall 0 \le i < j < m, \quad w_i^T \mathbf{A} w_j = w_j^T \mathbf{A} w_i = 0. \tag{3}$$

The solution x can be computed as follows:

$$x = \sum_{j=0}^{m-1} \frac{w_i^T b}{w_i T \mathbf{A} w_i} w_i. \tag{4}$$

Furthermore the iteration of w_i can be simplified as follows:

$$\begin{aligned} w_i &= \mathbf{A}w_{i-1} - \frac{(\mathbf{A}w_{i-1})^T(\mathbf{A}w_{i-1})}{w_{i-1}^T(\mathbf{A}w_{i-1})} w_{i-1} \\ &- \frac{(\mathbf{A}w_{i-2})^T(\mathbf{A}w_{i-1})}{w_{i-2}^T(\mathbf{A}w_{i-2})} w_{i-2}. \end{aligned} \tag{5}$$

The total time for the Standard Lanczos algorithm is $O(dn^2)+O(n^2)$.

The block Lanczos algorithm is an extension of the Standard Lanczos algorithm by applying it over field GF(2). There are some good properties on GF(2), for example, we can apply matrix to N vectors at a time (N is the computer word) and we can also do bitwise operations. Instead of using vectors for iteration, we using subspace instead. First we generate subspace:

$$\begin{array}{ll} \mathcal{W}_i & is\ \mathbf{A}-invertible, \\ \mathcal{W}_i^T \mathbf{A} \mathcal{W}_i = \{0\}, & \{i \neq j\}, \\ \mathbf{A}\mathcal{W} \subseteq \mathcal{W}, & \mathcal{W} = \mathcal{W}_0 + \mathcal{W}_1 + \ldots + \mathcal{W}_{m-1}. \end{array} \tag{6}$$

Then we define x to be:

$$x = \sum_{i=0}^{m-1} \frac{\mathbf{W}_j \mathbf{W}_j^T b}{\mathbf{W}_j^T}, \tag{7}$$

where $\mathbf{W}$ is a basis of $\mathcal{W}$. The iteration in the standard Lanczos algorithm will be changed to:

$$\begin{aligned} \mathbf{W}_i &= \mathbf{V}_i \mathbf{S}_i, \\ \mathbf{V}_{i+1} &= \mathbf{A}\mathbf{W}_i \mathbf{S}_i^T + \mathbf{V}_i - \sum_{j=0}^{i} \mathbf{W}_i \mathbf{C}_{i+1,j} \quad (i \ge 0), \\ \mathcal{W}_i &= \langle \mathbf{W}_i, \rangle \end{aligned} \tag{8}$$

in which

$$\mathbf{C}_{i+1,j} = \frac{\mathbf{W}_j^T \mathbf{A}(\mathbf{A}\mathbf{W}_i \mathbf{S}_i^T + \mathbf{V}_i)}{\mathbf{W}_j^T \mathbf{A} \mathbf{W}_j}. \tag{9}$$

This iteration will stop when $\mathbf{V}_i{}^T \mathbf{A} \mathbf{V}_i=0$ where $i = m$. The iteration can also be simplified as follows:

$$\mathbf{V}_{i+1} = \mathbf{A}\mathbf{V}_i \mathbf{S}_i \mathbf{S}_i^T + \mathbf{V}_i \mathbf{D}_{i+1} + \mathbf{V}_{i-1}\mathbf{E}_{i+1} + \mathbf{V}_{i-2}\mathbf{F}_{i+1}.$$

name	number
tst100$_{30}$	727563736353655223147641208603 = 743774339337499•978204944528897
F7$_{39}$	680564733841876926926749214863536422914 = 5704689200685129054721•59649589127497217
tst150$_{45}$	799356282580692644127991443712991753990450969 = 32823111293257851893153•24353458617583497303673
Briggs$_{51}$	556158012756522140970101270050308458769458529626977 = 1236405128000120870775846228354119184397•449818591141
tst200$_{61}$	1241445153765162090376032461564730757085137334450817128010073 = 1127192007137697372923951166979•1101360855918052649813406915187
tst250$_{76}$	3675041894739039405533259197211548846143110109152323761665377505538520830273 = 69119855780815625390997974542224894323•53169119831396634916152282437374262651

Table 1. The composite number n and the results after integer factorization

where $\mathbf{D}_{i+1}, \mathbf{D}_{i+1}, \mathbf{D}_{i+1}$ can be computed:

$$\begin{aligned}
\mathbf{D}_{i+1} &= \mathbf{I}_N - \mathbf{W}_i^{inv}(\mathbf{V}_i^T\mathbf{A}^2\mathbf{V}_i\mathbf{S}_i\mathbf{S}_i^T + \mathbf{V}_i^T\mathbf{A}\mathbf{V}_i), \\
\mathbf{E}_{i+1} &= -\mathbf{W}_{i-1}^{inv}\mathbf{V}_i^T\mathbf{A}\mathbf{V}_i\mathbf{S}_i\mathbf{S}_i^T, \qquad (10) \\
\mathbf{F}_{i+1} &= -\mathbf{W}_{i-2}^{inv}(\mathbf{I}_N - \mathbf{V}_{i-1}^T\mathbf{A}\mathbf{V}_{i-1}\mathbf{W}_{i-1}^{inv}) \\
&\quad (\mathbf{V}_{i-1}^T\mathbf{A}^2\mathbf{V}_{i-1}\mathbf{S}_{i-1}\mathbf{S}_{i-1}^T + \mathbf{V}_{i-1}^T\mathbf{A}\mathbf{V}_{i-1})\mathbf{S}_i\mathbf{S}_i^T.
\end{aligned}$$

$\mathbf{S}_i$ is an $N \times N_i$ projection matrix. We can reduce the iteration time from *O(n)* to *O(n/N)* using the block Lanczos algorithm.

4 Look-ahead Block Lanczos Algorithm

The Look-ahead block Lanczos algorithm proposed in Hovinen [6] has some advantages compared with Montgomery's block Lanczos algorithm: first of all, this algorithm is bi-orthogonalizing, so the input matrix does not need to be symmetric. Secondly, it solves the *breakdown* problem, namely $(\mathcal{W}_i{}^T\mathbf{A}\mathcal{W}_i=\{0\})$.

First we choose v_0 and u_0 from $\mathbb{R}^{n\times N}$. v_0 and u_0 are choosing uniformly and randomly. Then we will compute $v_1, v_2, \cdots, v_{m-1}$ and $u_1, u_2, \cdots, u_{m-1}$. For each iteration, we build up two matrices ξ_i and ω_i satisfy the following conditions:

1. $\xi_i A\omega_i$ is invertible.
2. For each $0 \le i < j < m$, $\xi_i A v_i$=0 and $u_i^T A\omega_i$=0.
3. $\bigoplus_{i=0}^{m-1}\langle\xi_i\rangle=\bigoplus_{i=0}^{\infty}\langle A^T u_0\rangle$, $\bigoplus_{i=0}^{m-1}\langle\omega_i\rangle=\bigoplus_{i=0}^{\infty}\langle Av_0\rangle$.

We also define variables $\bar{v}_i$, $\bar{u}_i$, $\hat{v}_i$, $\hat{v}_i$, $\breve{v}_i^i$, $\breve{u}_i^i$, σ_i^v and σ_i^u have the properties:

1. $\hat{v}_i^T A v_i = 0$;
2. $u_i^T A\hat{v}_i$;
3. $\bar{u}_i^T A\bar{v}_i$ is invertible.

σ_i^v and σ_i^u are two matrices in $\mathbb{K}^{N\times N}$ and $\breve{v}_i$=$v_i\sigma_i^v$, $\breve{u}_i$=$u_i\sigma_i^u$. v_{i+1} and u_{i+1} can be compute by:

$$u_{i+1} = A^T u_i - \sum_{k=0}^{i} \dot{u}_k^i((\bar{v}_k^i)^T A^T \dot{u}_k^i)^{-1}(\bar{v}_k^i)^T(A^T)^2 u_i, \quad (11)$$

$$v_{i+1} = Av_i - \sum_{k=0}^{i} \dot{v}_k^i((\bar{u}_k^i)^T A\dot{v}_k^i)^{-1}(\bar{u}_k^i)^T A^2 v_i. \quad (12)$$

If $\hat{u}_i^T A^2 v_i$ is full rank, $\dot{v}_i^{i+1}$ can be defined as:

$$\dot{v}_i^{i+1} = (\bar{v}_i|\eta_{i+1}), \quad (13)$$

otherwise

$$\dot{v}_i^j = (\dot{v}_i^{j-1}|\eta_j). \quad (14)$$

η_{i+1} is a matrix consisting of columns of $\breve{v}_{i+1}$. Finally, the solution x can be calculated by:

$$x = \sum_{i=0}^{m-1} \frac{\dot{u}_i^m(\bar{u}_i^m)^T b}{(\bar{u}_i^m)^T A\dot{v}_i^m}. \quad (15)$$

5 Implementation Details

As we mentioned before, the most time consuming part in GNFS is the sieving part. This part has already been parallelized in our previous work [19]. Another part in GNFS program can be significantly improved is to solve the large and sparse linear systems over GF(2) in parallel by look-ahead block Lanczos algorithm, instead of the Montgomery's block Lanczos algorithm which is less reliable. Our parallel code is built on the sequential source GNFS code from Monico [12].

5.1 Parallel sieving

The sieving step in sequential GNFS is very suitable for parallel. The purpose of sieving is to find enough *(a,b)* pairs. The way we do sieving is: fixing *b*, let a change from *-N* to *N* (*N* is a integer, usually larger than *500*), then we check each *(a,b)* pair whether it smooth over factor bases. The sieving for next *b* can not start until the previous sieving is finished.

In parallel, we use several processors do sieving simultaneously, each slave node takes a small range of *b*, then send results back to master node. The detailed parallel sieving implementation can be found in [19].

5.2 Hardware and programming environment

The whole implementation uses two software packages, the sequential GNFS code from Monico [12] (Written in ANSI C) and sequential Look-ahead block Lanczos code from Hovinen [6] (Written in C++). For parallel implementation, MPI (Message Passing Interface) [5] library is used. In order to do arbitrary precision arithmetic, the GMP 4.x is also used [4]. We use GUN compiler to compile whole program and MPICH1 [14] for our MPI library. The version of MPICH2 is 1.2.5.2. The cluster we use is a Sun cluster from University of New Brunswick Canada whose system configurations is:

- Model: Sun Microsystems V60.
- Architecture: x86 cluster.
- Processor count: 164.
- Master node: 3 GB registered DDR-266 ECC SDRAM.
- Slave node: 2 to 3 GB registered DDR-266 ECC SDRAM.

In the program, Each slave node only communicates with the master node. Figure 1 shows the flow chart of our parallel program.

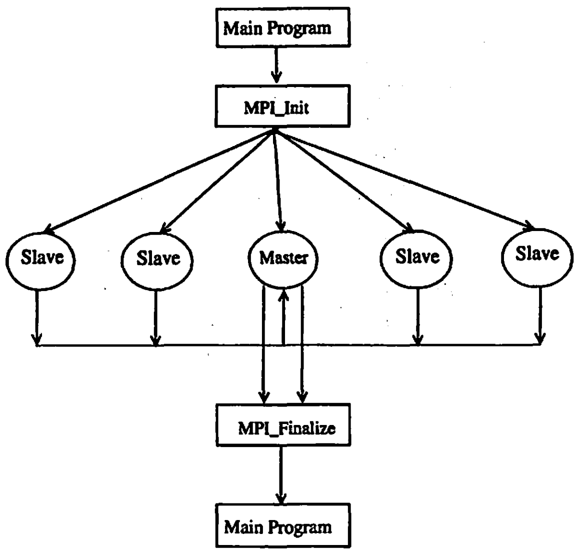

Figure 1. Each processors do the sieving at the same time, and all the slave nodes send the result back to master node

6 Performance Evaluation

We have six test cases, each test case have a different size of n, all are listed in Table 1.

The sieving time increases when the size of n increases. Table 2 shows the average sieving time for each n with one processor. Table 3 shows the number of processors we use for each test case. Figure 2 and 3 show the total execution time for each test case in seconds.

The total sieve time for test case: tst100, F7, tst150, Briggs and tst200 are presented in Figure 4. Figure 5 gives the speed-ups and parallel efficiency for each test case with different processor numbers.

One explanation of inefficiency in this parallel implementation is the communication time. In this program, we assume b change from b_0 to b_1, use p processors, the total message send and receive is $(3(b_0-b_1)-\frac{b_0-b_1}{p})$ (every sieve result contains three messages), these messages will be buffered and send to master when sieving is finished in each processor. This cause a lots of communications and makes the master node very busy.

Another explanation source of inefficiency is the loading balance. The master node has to wait until all the messages are received from slave nodes. Further improvement is being conducted. The detailed results will be reported in the near future.

7 Acknowledgements

We would like to thank C. Monico of Texas Tech University and B. Hovinen of University of Waterloo. Our work is based on their sequential source codes. They also helped us with some technical problems through emails. Dr. Silva from IBM advised us many good ideas for our parallel program.

References

[1] M. E. Briggs. An introduction to the general number field sieve. Master's thesis, Virginia Polytechnic Institute and State University, 1998.

[2] M. Case. A beginner's guide to the general number field sieve. pages 1–4, Winter 2003.

[3] J. Dreibellbis. Implementing the general number field sieve. pages 5–14, June 2003.

name	number of sieve	average sieve time(s)
$tst100_{30}$	1	35.6
$F7_{39}$	1	28.8
$tst150_{45}$	5	50.6
$Briggs_{51}$	3	85.67
$tst200_{61}$	7	560.6
$tst250_{76}$	7	4757.91

Table 2. Average sieving time for each n

name	number of slave processors
$tst100_{30}$	1,4,8,16
$F7_{39}$	1,4,8,16
$tst150_{45}$	1,4,8,16
$Briggs_{51}$	1,4,8,16
$tst200_{61}$	1,4,8,16
$tst250_{76}$	1

Table 3. Number of processors for each test case

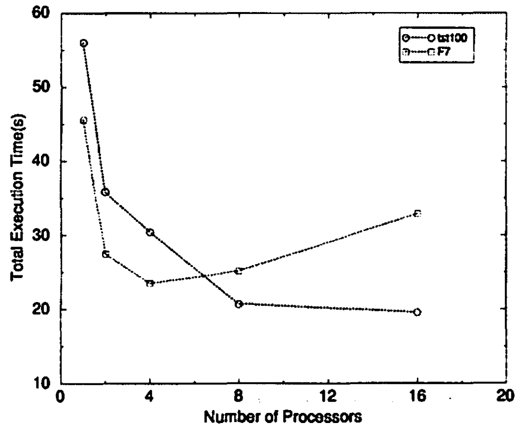

Figure 2. Execution time for tst100 and F7

[4] T. Granlund. *The GNU Multiple Precision Arithmetic Library*. TMG Datakonsult, Boston, MA, USA, 2.0.2 edition, June 1996.

[5] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1994.

[6] B. Hovinen. Blocked lanczos-style algorithms over small finite fields. Master Thesis of Mathematics, University of Waterloo, Canada, 2004.

[7] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. In *Journal of Research of the National Bureau of Standards*, volume 45, pages 255–282, 1950.

[8] C. Lanczos. Solutions of linread equations by minimized iterations. In *Journal of Research of the National Bureau of Standards*, volume 45, pages 33–53, 1952.

[9] A. K. Lenstra. Integer factoring. *Des. Codes Cryptography*, 19(2-3):101–128, 2000.

[10] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics(2)*, 126:649–673, 1987.

[11] H. W. Lenstra, C. Pomerance, and J. P. Buhler. Factoring integers with the number field sieve. In *The Development of the Number Field Sieve*, volume 1554, pages 50–94, New York, 1993. Lecture Notes in Mathematics, Springer-Verlag.

[12] C. Monico. General number field sieve documentation. Nov, 2004.

[13] P. L. Montgomery. A block lanczos algorithm for finding dependencies over gf(2). In *EUROCRYPT '95*, volume 921 of *LNCS*, pages 106–120. Springer, 1995.

[14] MPICH. http://www-unix.mcs.anl.gov/mpi/mpich/.

[15] J. M. Pollard. Theorems on factorization and primality testing. In *Proceedings of the Cambridge Philosophical Society*, pages 521–528, 1974.

[16] C. Pomerance. The quadratic sieve factoring algorithm. In *Proceeding of the EUROCRYPT 84 Workshop on Advances in Cryptology: Theory and Applications of Cryptographic Techniques*, pages 169–182. Springer-Verlag, 1985.

[17] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[18] M. C. Wunderlich and J. L. Selfridge. A design for a number theory package with an optimized trial division routine. *Communications of ACM*, 17(5):272–276, 1974.

[19] L. Xu, L. T. Yang, and M. Lin. Parallel general number field sieve method for integer factorization. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-05)*, pages 1017–1023, Las Vegas, USA, June, 2005.

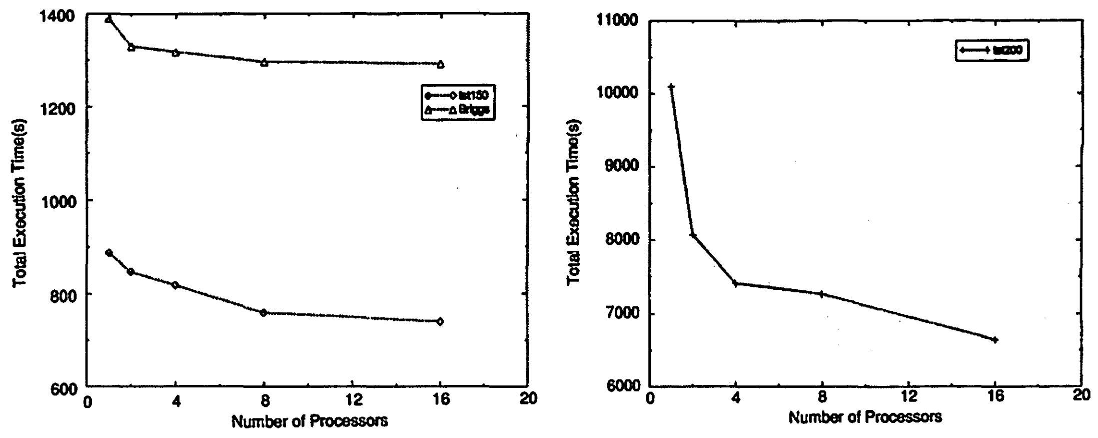

Figure 3. Execution time for tst150, Briggs and tst200

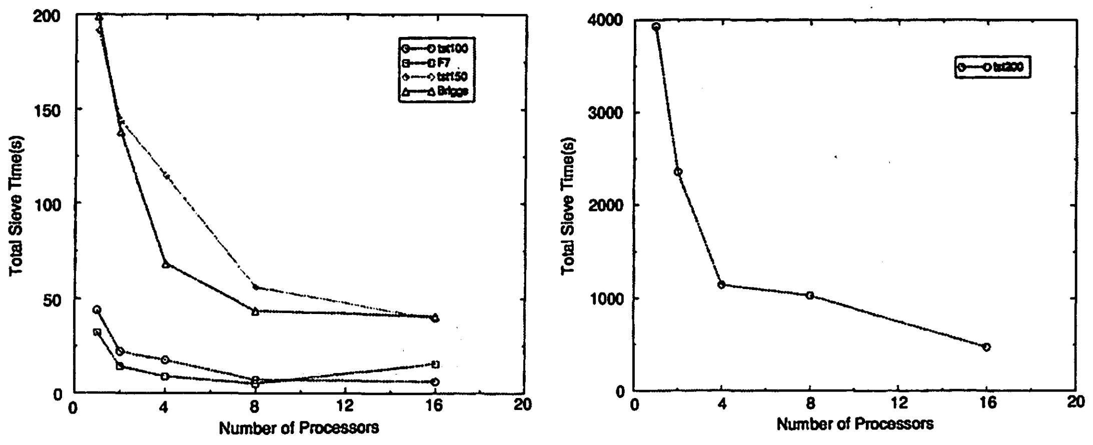

Figure 4. Sieve time for tst100, F7, tst150, Briggs and tst200

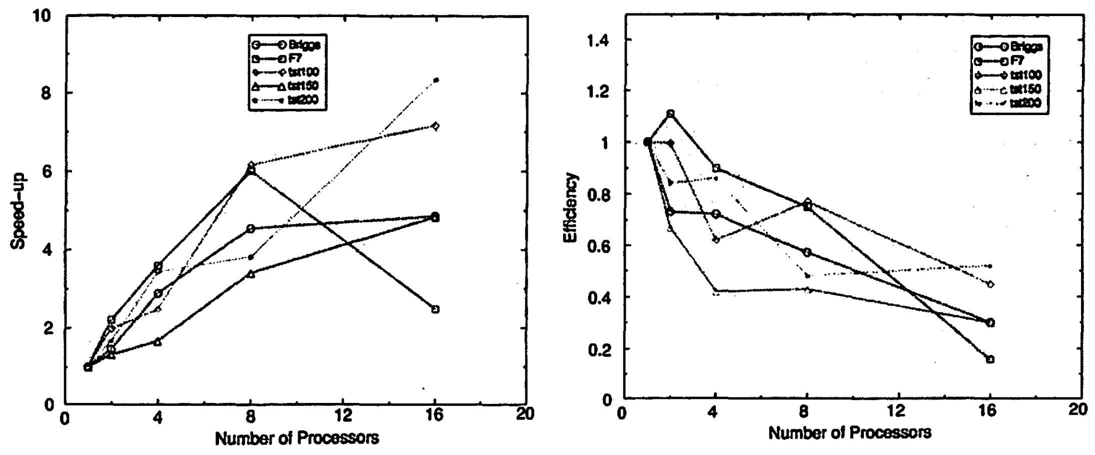

Figure 5. Speedups and parallel efficiency