2009 International Conference on Parallel and Distributed Computing, Applications and Technologies

SMP Based Solver For Large Binary Linear Systems

Nikhil Jain, Brajesh Pande, Phalguni Gupta
Indian Institute of Technology Kanpur
Kanpur - 208016, India
nikhjain@in.ibm.com, {brajesh, pg}@iitk.ac.in

Abstract

Solving a large sparse system of linear equations (SSLE) over the binary field is an important step in the general number field seive (GNFS) algorithm that is often used to factorize large integers used in the RSA algorithm. Block Lanczos method proposed by Montgomery is an efficient and reliable method for solving such systems. A number of implementations of Block Lanczos method on distributed systems have been proposed in past. This paper discusses the parallel Montgomery's Block Lanczos method for solving binary system of equations on shared multiprocessors (SMP). A simple experiment shows that the speed of convergence of this algorithm is dependent on a wise choice of initial guess for kick-starting the algorithm. A somewhat dense choice of the initial guess is suggested rather than a random choice for faster convergence. Devoid of any communication overheads of a loosely coupled system, the improved method gives good performance on SMP systems which can provide an able alternative to the more popular distributed systems. Details of implementation of this algorithm on SMP and experimental results are also provided in the paper.

1. Introduction

RSA [1] is the first algorithm known to be suitable for signing as well as encryption. High level of security of a RSA system is owing to difficulties faced in factoring very large integers. Factoring of large integers is either computationally very expensive or too complicated to implement. GNFS [2] is the most efficient and easy to implement algorithm for factoring large integers.

GNFS consists of two computationally intensive steps: sieving and solving a large SSLE over GF(2). It takes several months to factor large integers of using these two steps. Montgomery's Block Lanczos method [3] is the most common method used to solve large sparse linear systems over GF(2). In this paper, an improved parallel Montgomery's block Lanczos method for SMP is proposed. The method leads to selection of almost all the columns in invertible subspace in each iteration. Implementation on SMP helps in eliminating communication overheads which are immense, and assists in building an efficient and good alternative to DS based implementations. OpenMP, which has unified industry protocol for SMP, has been used to facilitate the usage of implementation across the community.

2 Preliminaries

RSA can be attacked directly by factorizing publicly available *n* using GNFS [2]. This produces the integers *p* and *q* and the private key can then be generated. GNFS uses six steps to factorize a given composite number. First two steps of GNFS require selection of a polynomial and factor bases. Sieving is the next step that involves generation of pairs *(x,y)* to build a linear dependence which is filtered in the fourth step. Using the generated data, a large SSLE over GF(2) is built and solved in the next step. Finally, the solution of SSLE is used to factorize *n* in the sixth step.

Step 3 and Step 5 are the most time consuming steps in GNFS. The size of the SSLE produced in Step 5 is dependent on the number of digits in *n*. A SSLE of dimension of the order of $O(10^5 \text{ x } 10^5)$ is produced for a 110 digit number. A 120 digit number requires SSLE of size in excess of 10^6 x 10^6. However the number of non zero entries in the SSLE is extremely low. The density of such SSLE lies in the region of 0.001% to 0.005%. As such, an iterative method can be used to solve the equations.

2.1 Lanczos Algorithm

Suppose $\mathbf{A} = \mathbf{B^T B}$ is a symmetric positive definite matrix over GF(2) satisfying $\overline{\mathbf{b}} = \mathbf{B^T b}$. The standard Lanczos algorithm [4] can be used to solve $\mathbf{Ax} = \overline{\mathbf{b}}$ by iterating

$$v_i \;=\; Av_{i-1} - \sum_{j=0}^{i-1} c_{ij} v_j \qquad (1)$$

where coefficients c_{ij} are defined as

$$c_{ij} = \frac{v_j^T A^2 v_{i-1}}{v_j^T A v_j} \tag{2}$$

The iteration continues until $v_i = 0$ for some $i = m$. The relation $v_i^T A v_j = 0$ if $i \neq j$ can be easily verified by an induction on max*(i,j)* and using the symmetry of matrix **A**. If a solution **x** to **Ax** = $\overline{\mathbf{b}}$ exists in **K(A,b)** = $span\{b, Ab, A^2b\cdots\}$, it can be recovered by projecting **b** onto each of the Krylov basis vectors:

$$x = \sum_{i=0}^{m-1} \frac{v_i^T b}{v_i^T A v_i} v_i \tag{3}$$

Lanczos algorithm has a critical issue when applied over finite fields. All the operations in prime field GF(**p**) are modulo **p**. In such a case, $v_m^T A v_m = 0$ may hold even when $v_m \neq 0$. This forces a breakdown of the algorithm and it halts long before termination condition is achieved. The probability of such a breakdown is inversely proportional to the size of the field. While working on GF(2) such a situation arises half the times and can be handled using Block Lanczos algorithm [3].

2.2 Block Lanczos Algorithm

Block Lanczos algorithm [3] computes a sequence of subspaces $W_0, W_1 \cdots W_m$ (combination of independent column vectors) such that

- W_i is A-invertible i.e. the subspace has a basis of column vectors V_i such that $V_i^T A V_i$ is invertible.
- W_i and W_j are A-orthogonal if $i \neq j$.
- AW $\subseteq$ W where $W = W_0 + W_1 + \cdots + W_{m-1}$.

The solution vector **x** can be recovered by projecting **b** onto the basis vectors V_i's of the subspaces W_i's using

$$x = \sum_{j=0}^{m-1} V_j (V_j^T A V_j)^{-1} V_j^T b \tag{4}$$

The computer word size – *N* – is a good candidate for the number of column vectors in W_i owing to possible usage of bit vector operations to perform parallel computation. However A invertibility of the subspace needs to be ensured by special means. The first vector subspace, W_0, consists of *N* randomly generated vectors. The basis, V_0, is constructed by selecting as many columns of W_0 as possible subjected to A-invertibility requirement. Mathematically this can be represented as post multiplication of W_0 by a matrix S_0 i.e. $V_0 = W_0 S_0$. Using the recurrence W_1, which is A-orthogonal to V_0, is constructed. V_1 is obtained from W_1 using the same method which has been used for obtaining V_0 from W_0. In general in the i^{th} iteration, matrix W_i which is A-orthogonal to all earlier V_j is constructed followed by selection of V_i from W_i. Denoting $V_i^{inv} = S_i(V_i^T A V_i)^{-1} S_i^T$, the three terms recurrence relation can be reduced to

$$W_{i+1} = AW_iS_iS_i^T + W_iD_{i+1} + W_{i-1}E_{i+1} + W_{i-2}F_{i+1} \tag{5}$$

$$\begin{aligned} D_{i+1} =& \; I_N - V_i^{inv}(W_i^T A^2 W_i S_i S_i^T + W_i^T A W_i) \\ E_{i+1} =& \; -V_{i-1}^{inv} W_i^T A W_i S_i S_i^T \\ F_{i+1} =& \; -V_{i-2}^{inv}(I_N - W_{i-1}^T A W_{i-1} V_{i-1}^{inv}) \\ & (W_{i-1}^T A^2 W_{i-1} S_{i-1} S_{i-1}^T + W_{i-1}^T A W_{i-1}) S_i S_i^T \end{aligned}$$

The details of this derivation are presented in [3]

Selection of *N* vectors in a subspace results in usage of bit vectors - vectors which are bit packed form of matrices. Every entry in a bit vector represents *N* entries of the original matrix. As such, position of a bit in an entry of a bit vector determines its position in original matrix. Usage of bit vectors facilitate parallel operations using bit wise operators and save storage space. Algorithm 1 presents the pseudo for the Block Lanczos algorithm. In the pseudo code, matrices **D**, **E**, **F**, **prod** and **val** are bit vectors of dimension *N x N* and matrices **W** and **V** are bit vectors of dimension *n x N*.

Algorithm 1 Block Lanczos Algorithm

```
Input: Matrix B of size n x n and vector b.
Output: Vector x such that Bx = b.
 1: Initialize i = 0, count = 0, x = 0, $S_{-1} = I_N$, $W_0$ = Random value.
 2: $prod1_0 = B * W_0$.
 3: $prod2_0 = prod1_0^T * prod1_0$.
 4: while $W_i \neq 0$ and count $< n$ do
 5:     $\{V_i^{inv}, S_i\}$ = findVinvS ($prod2_i, S_{i-1}$).
 6:     x = x + $(W_i * (V_i^{inv} * W_i^T * W_i))$.
 7:     $prod3_i = B^T * prod1_i$.
 8:     $val_i = ((prod1_i^T * B * prod3_i) * S_iS_i^T) + prod2_i$.
 9:     $D_{i+1} = I_N - (V_i^{inv} * val_i)$.
10:     $E_{i+1} = -V_{i-1}^{inv} * prod2_i * S_iS_i^T$.
11:     $F_{i+1} = -V_{i-2}^{inv} * (I_N - (prod2_{i-1} * V_{i-1}^{inv})) * val_{i-1} * S_iS_i^T$.
12:     $W_{i+1} = (prod3_i * S_iS_i^T) + (W_i * D_{i+1}) + (W_{i-1} * E_{i+1}) + (W_{i-2} * F_{i+1})$.
13:     $prod1_{i+1} = B * W_{i+1}$.
14:     $prod2_{i+1} = prod1_{i+1}^T * prod1_{i+1}$.
15:     i = i + 1.
16: end while
17: if count $< n$ then
18:     return x.
19: end if
```

3 Implementation Details

This section discusses the implementation of the algorithm discussed in the previous section on SMP machines.

The algorithm is sensitive to the initial choice of the subspace W_0. Implementation on SSLE over GF(2) has implications on the storage structure, addition and multiplication of packed bits and data distribution. Discussion on such issues form the core part of this section.

3.1 Initializing W_0

Montgomery's Block Lanczos algorithm [3] does not specify any module or constraint for initializing subspace W_0. The algorithm correctly solves an independent system of equations irrespective of the value of W_0. However, value of subspace W_0 has an effect on the number of columns that gets selected in the basis V_i from the subspace W_i. This selection in turn has a direct effect on the number of iterations performed before the termination condition is achieved. As such, initialization routine of the subspace W_0 has a major influence on the run time of the algorithm.

Density of W_0	Iterations Required
0.01%	263
0.02%	190
0.05%	184
0.10%	169
1.00%	158
5.00%	158

Table 1. Iteration count on system with n = 10^4 and density 0.1%

Initialization using very sparse subspace (which may happen in case of random initialization) results in selection of only $\frac{N}{2}$ columns in basis V_i from subspace W_i. Usage of W_0 based on vector **b** suffers from similar problem. As is clear from Table 1, selection of a random subspace W_0 with density greater than 1% is recommended for a faster convergence of the algorithm. If the density of W_0 is greater than 1%, selection of greater than *N-2* columns can be ensured in basis V_i from subspace W_i. Hence, the number of iterations required for achieving termination condition is reduced drastically which in turn results in reduction of total execution time of the algorithm.

3.2 Storage of Sparse Matrix

The sparse matrix can be represented by a set of variables - integer '*n* and *m*' for storing dimensions, integer array '*nonZero[]*' for storing number of non zeroes cumulatively in rows and a 2-dimensional array '*vals[][]*' for storing the non zero column numbers. The array *nonZero* in its i^{th} entry stores the number of non zero entries in the first *i* rows of the matrix. Matrix *vals* stores the non zero column numbers in the i^{th} row of the matrix **B** in its i^{th} row as shown in Figure 1. Storing a sparse matrix with .1% non zero entries takes 1000 GB if a standard 2-dimensional character array is used. This format provides huge memory gains by storing the same matrix in 8 GB of memory.

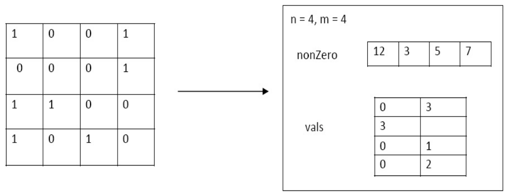


Figure 1. Storage of Sparse Matrix

3.3 Addition and Multiplication

Addition and subtraction of two bits is performed using XOR operation as a general practise while multiplication of bits can be achieved using AND operation. Addition using XOR can be directly extended to bit packed vectors. Given two bit vectors **A** and **B** of dimension *n x N*, addition (or subtraction) can be done as follows:

$$c_i \;=\; a_i\, XOR\, b_i \quad for\; i \;=\; 1,\, 2, \cdots\, n \tag{6}$$

Given matrices **A** (*n x m*) and **B** (*m x N*), matrix **C** (*n x N*) = **A** * **B** can be obtained using bit vector operations in an indirect manner. Algorithm 2 illustrates the usage of outer product methodology and bit vector operations to multiply two matrices. Outer product has been used whenever possible as the cost of multiplication using outer product is much less than the cost using other methods.

Algorithm 2 Multiplication Algorithm

```
Input: Matrices A (n x m) and B (m x N) where B is a bit vector.
Output: Bit vector C (n x N) such that C = A * B.
    Initialize $c_i$ = 0 for $i$ = 1, 2 ··· $n$.
    for $i$ = 1, 2 ··· n do
        for $j$ = non zero column numbers in $i^{th}$ row do
            $c_i$ = $c_i$ XOR $b_j$. if C = A * B
            $c_j$ = $c_j$ XOR $b_i$. if C = $A^T$ * B
        end for
    end for
```

3.4 Algorithm for Finding V^{inv}

Algorithm 3 presents the algorithm used for finding V^{inv} and S with the help of the row-echelon process and row operations. To find the inverse of matrix $W_i^T A W_i$, the left half of the matrix $[W_i^T A W_i \| \; I_N]$ needs to be brought into row-echelon form. Then the matrix $[I_N \| \; (W_i^T A W_i)^{-1}]$ contains the inverse of the input matrix in the right half.

Algorithm 3 findVinvS - Algorithm to find V^{inv} and S

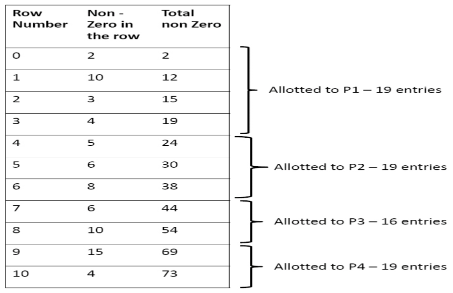
```
Input: $prod_i = W_i^T B^T B W_i$ and $S_{i-1}$.
Output: $S_i$ and $V_i^{inv}$ such that $V_i^{inv} = S_i(S_i^T * prod_i * S_i)^{-1} S_i^T$.
 1: Initialize M[$1 \cdots N$] = $prod_i[1 \cdots N]$ and M[$N \cdots 2N$] = $I_N[1 \cdots N]$.
 2: Initialize $S_i$ = NULL, columnCount = 0.
 3: for i = 1 to N do
 4:     for j = i to N do
 5:         if $M_{ji} \neq 0$ then
 6:             Exchange row i and j. Break the loop.
 7:         end if
 8:     end for
 9:     if $M_{ii} \neq 0$ then
10:         Zero rest of the column i.
11:         $S_i = S_i \cup \{i\}$.
12:         columnCount = columnCount + 1.
13:     else
14:         for j = i to N do
15:             if $M_{j,i+N} \neq 0$ then
16:                 Exchange row i and j. Break the loop.
17:             end if
18:         end for
19:         Zero rest of the column i+N.
20:         Set row i to zero.
21:     end if
22: end for
23: Set $V_i^{inv}$ = M[$N+1 \cdots 2N$].
24: return columnCount. =0
```

4 Parallelization

Computationally intensive steps of Algorithm 1 can be parallelized for substantial gains in computation time. These steps are: reading of the input matrix; multiplication of sparse matrix with bit vectors i.e. multiplication of a *n x m* matrix with a *m x N* matrix ; multiplication of bit vectors with bit vectors i.e. multiplication of a *n x N* matrix with a *N x N* matrix or of *N x n* matrix with b matrix; and the corresponding transpose multiplications. Computations like addition, simple multiplication and pivot search take a very small fraction of the total time. Any attempt to parallelize them may lead to small gains (or may be small losses).

4.1 Distribution of Matrices

A major advantage of working on SMP is availability of the global data set to all the processing units. Communication overheads are minimal as there is no physical distribution of data among processors. However, a logical distribution of data set among the processors is needed for distribution of tasks among processors. A standard method is to distribute equal number of consecutive rows to every processor. In such a case, rows $start = \frac{(n*pid)}{np}$ to $end = \frac{(n*(pid+1))}{np}$ of the matrix are assigned to the processor with id as pid from np total processors. Such a distribution may result in elongated synchronization periods if the matrix under consideration is a sparse matrix. A distribution that aims at equal division of number of non zeroes (nZ) among the processors should be done. A processor with id stored in pid is alloted rows $start$ to end where values of $nZ[start]$ and $nZ[end]$ are closest to $\frac{(nZ*pid)}{np}$ and $\frac{(nZ*(pid+1))}{np}$ respectively as shown in Figure 2.



Row Number	Non - Zero in the row	Total non Zero
0	2	2
1	10	12
2	3	15
3	4	19
4	5	24
5	6	30
6	8	38
7	6	44
8	10	54
9	15	69
10	4	73

Allotted to P1 – 19 entries
Allotted to P2 – 19 entries
Allotted to P3 – 16 entries
Allotted to P4 – 19 entries

Figure 2. Distribution of rows of sparse matrix among processors

4.2 Parallel Read

Instead of employing one processor to read elements from the file, each processor is used to read some part of the file simultaneously. Every processor can directly access the $start$ row of matrix B alloted to it by a file seek. Thereafter all the processors can read till end row of matrix B in parallel. However huge gains can only be obtained if the filesystem does not serialize the request onto the disk.

4.3 Parallel Multiplication

Algorithm 2 suggests that if all entries of i^{th} row of matrix **A** are being processed by a processor, then no other processor should make any changes to i^{th} row of resultant matrix **C**. Hence parallelization of multiplication of the type **C = A*B** can be done by an interleaved division of rows of matrix **A** as shown in Algorithm 4.

Algorithm 4 Parallel Multiplication Algorithm

```
Note: This method is to be run on all processors simultaneously.
1: Initialize $c_i$ = 0 for i = start ··· end.
2: for i = start ··· end do
3:     for j = non zero column numbers in $i^{th}$ row do
4:         $c_i = c_i$ XOR $b_j$.
5:     end for
6: end for =0
```

Multiplication of type $\mathbf{C} = \mathbf{A^T} * \mathbf{B}$ can not be extended directly. Row major storage of matrix A does not provide an assorted list of entries of a particular column. However every processor can make necessary changes to a local copy of the resultant matrix **C** based on the non zero entries in the rows allocated to it. Thereafter, every processor can update the global copy as shown in Algorithm 5. The reordering of the operations is possible as XOR is commutative.

Algorithm 5 Parallel Transpose Multiplication Algorithm

```
Note: This method is to be run on all processors simultaneously. EXCLUSIVE STEP is a step which can be executed by only one processor at a time (critical section of the code).
1: Initialize local copy $d_i$ = 0 for $i$ = 0, 1 $\cdots$ $n$.
2: for $i$ = start $\cdots$ end do
3:     for $j$ = non zero column numbers in $i^{th}$ row do
4:         $d_j = d_j$ XOR $b_i$.
5:     end for
   EXCLUSIVE STEP - FOR LOOP
6:     for $j$ = 1, 2 $\cdots$ $n$ do
7:         $c_j = c_j$ XOR $d_j$.
8:     end for
9: end for
```

4.4 Synchronization & Mutual Exclusion

Correctness of the synchronized program and maintaining mutual exclusion are key concerns in a program for SMP machines. Synchronization is needed if a value being evaluated is to be used in subsequent code or if a value being used is to be evaluated in the following code. In either case, absence of barriers results in erroneous output. Current implementation of the Block Lanczos Algorithm ensures correctness by putting a barrier after all the parallelized function calls. Barriers have also been put after function call made by only one of processors incase the changes made are being used by other processors. Mutual exclusion has been ensured by adopting two measures - division of writing space and use of critical block. An attempt has been made to divide the writing space whenever possible as it saves a lot of time (like in multiplication). However in certain cases, such a division is not possible and in those places mutual exclusion has been ensured via use of critical blocks (like in transpose multiplication).

5 Experimental Results

The solver has been evaluated on systems of size varying from **n** = 10000 to **n** = 1000000 i.e $O(10^4)$ to $O(10^6)$. The Intel SMP machine considered for this purpose is a Xeon(R) CPU E5450 @ 3.00 GHz with a 64 bit machine architecture, 8 core SMP, 6144 kb Cache per core and a 32 GB Memory that runs on Mandriva. Matrix **B** and vector **x** are generated independently for a given density and size. Using **B** and **x**, right hand side vector **b** is computed. The solver code checks for correctness by solving a given system **(B, b)** and comparing the obtained solution **x** with the generated **x**. The solver code has found the correct solution whenever it exists for the experiments undertaken. The data-sets have been classified into small, moderate and large sized systems based on matrix size. The performance of the solver on these systems has been evaluated by varying the density of the matrices and studying the speed-up.

5.1 Small Sysems

Processors	.1% dense (s)	.5% dense (s)	1% dense (s)	2% dense (s)
1	6.2	7.1	8.4	11.1
2	3.7	4.2	4.7	5.9
4	3.2	3.6	3.2	3.7
6	2.7	2.9	2.9	2.9
8	2.4	2.6	2.8	2.6

Table 2. Run time for 10^4 x 10^4 system

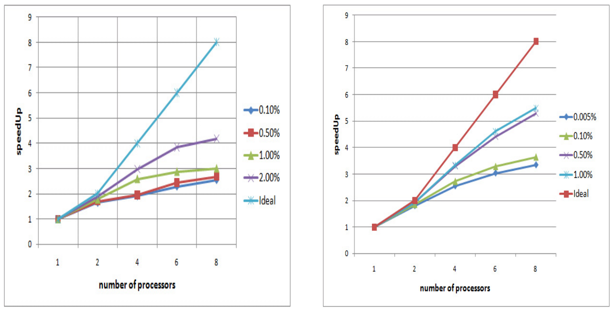


Figure 3. Performance on system with n = 10^4 and n = 10^5

Table 2 presents the run time taken for solving a sparse system of size 10^4 x 10^4. Such systems occur in factorizing composite numbers of less than 100 digits. Table 2 shows that variation of density has no major effect on the run time. This can be attributed to the small size of the matrix. All the graph lines in Figure 3 appear to be coincident, which indicates non dependence of speed up on change in density. However, a closer look suggests that the performance has improved with increase in density. Graph line for 2% density overlaps with other lines at the start but ends much higher in terms of speed up. The results also suggest an overall poor performance of the parallelized code as the speed up never goes beyond 3 for most of the cases. This is largely due to the small number of non zeroes per equation (10 to 200 non zeroes per row) which results in a very small amount of computation between two synchronization points. As such synchronization losses overhauls the gains made by parallelization.

5.2 Moderate Systems

Performance on system with **n** = 10^5 has been presented in Table 3 and Figure 3. The amount of computational task has increased as a result of increase in number of non zeroes (due to increase in size of matrix). The imminent effect can be seen in the improved performance of the parallelized program. Speed up, which was below 3 previously, has crossed the barrier mark of 5. A steady increase in run time with increase in density of the matrix can be observed. The run time becomes almost 1.6 times as the density of the matrix is doubled from 0.5% to 1%. However a proportionality relation cannot be framed merely on the basis of results obtained.

Processors	.005% dense (min)	.1% dense (min)	.5% dense (min)	1% dense (min)
1	9.9	14.4	33.3	55.9
2	5.4	7.7	17.1	28.81
4	3.8	5.3	10.1	16.6
6	3.2	4.4	7.5	12.1
8	2.9	3.9	6.3	10.1

Table 3. Run time for 10^5 x 10^5 system

5.3 Large Systems

Sieving phase of integers with more than 115 digits produce systems with size of the order of $O(10^6)$. The solver has taken approximately 70 hours to solve a system of size $n = 10^6$ and density .1% which is better than any publicly available implementation. Approximately 260 hours are required to solve such a system using one processor while four processors have taken 77 hours. Gains of only 7 hours has been made for 4 more processors. Table 4 and Figure 4 suggest the same. The reason for this kind of saturating behavior is the global updation discussed earlier and warrants further study.

No. of Processors	.1% Dense (hours)
1	259.76
2	138.25
4	77.77
6	73.44
8	69.128

Table 4. Run time for 10^6 x 10^6 system

5.4 Space Requirements

Table 5 presents the RAM required for the execution of the proposed parallel solver. Space requirement is independent of the number of processors used. Most of the memory required is for the storage of matrix **B**. The formula for approximating the memory requirement on a machine, which takes 4 bytes to store an integer, is

$$memory\ required = 4 * nonZeroes\ bytes \qquad (7)$$

6 Conclusion

This paper discusses a SMP based parallel solver for solving sparse systems over GF(2). System of size 10^6 and density .1% has been solved in approximately 70 hours using 8 processors. SMP has been used to avoid the communication time losses which become very significant with repetitive matrix vector multiplications. The experimental results show that the optimizations and parallelization have resulted in an efficient SMP based implementation. The solver finds its application in breaking of large RSA keys using GNFS.

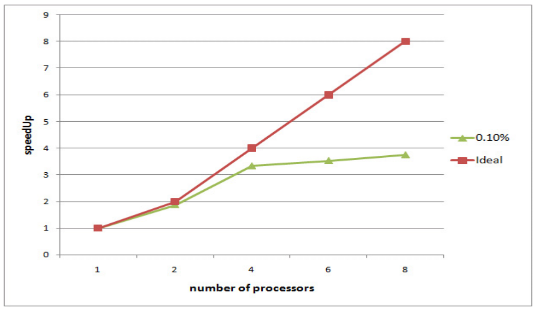


Figure 4. Performance on system with n = 10^6

n (size of system)	Density	Memory required (MB)
10000	0.1%	.5
10000	1.0%	5
10000	2.0%	10
100000	0.1%	50
100000	1.0%	500
100000	2.0%	1000
1000000	0.1%	5000
1000000	1.0%	50000
1000000	2.0%	100000

Table 5. Memory Requirement

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] M. Briggs, "An introduction to the general number field sieve," Master's thesis, Virginia Polytechnic Institute, Blacksburg, Virginia, 1998.

[3] P. L. Montgomery, "A block lanczos algorithm for finding dependencies over GF(2)," in *Advances in Cryptology EUROCRYPT 95*, vol. 921, 1995, pp. 106 –120.

[4] C. Lanczos, "Solution of systems of linear equations by minimized iterations," *J. Res. Nat. Bur. Standards*, vol. 45, pp. 255 – 282, 1952.