Load Balanced Block Lanczos Algorithm over GF(2) for Factorization of Large Keys

Wontae Hwang and Dongseung Kim
Department of Electrical Engineering
Korea University, Seoul, Korea (Rep. of)
dkim@classic.korea.ac.kr

Abstract. Researchers use NFS (Number Field Sieve) method with Lanczos algorithm to analyze big-sized RSA keys. NFS method includes the integer factorization process and nullspace computation of huge sparse matrices. Parallel processing is indispensible since sequential computation requires weeks (even months) of CPU time with supercomputers even for 150-digit RSA keys. This paper presents details of improved block Lanczos algorithm based on previous implementation[4,10]. It includes a new load balancing scheme by partitioning the matrix such that the numbers of nonzero components in the submatrices become equal. Experimentally, a speedup up to 6 and the maximum of efficiency of 0.74 have been achieved using an 8-node cluster with Myrinet interconnection.

Keywords: parallel/cluster computing, cryptology, RSA key, load balancing, sparse matrix

1. Introduction

Eigenvalue problems often used in mechanical structure engineering and quantum mechanical engineering are very computation intensive. Sequential computing is never suitable since it takes days even weeks of running time. Lanczos algorithm [3,7] to solve eigenproblems $Ax = \lambda x$ is widely used when A is a large, sparse, and symmetric matrix. Parallelization of the algorithm has been drawn special attention of many people. Especially, researchers in cryptology apply Lanczos algorithm to factorize long integers of RSA keys [15,16] often requiring thousands of MIPS year to compute by known algorithms. For example, a 140-digit RSA key requires 4,671,181 x 4,704,451 B matrix, which demands 585Mbytes (= 4,671,181 * 32.86 * 4bytes) if there are 32.86 nonzeros per column [1]. Thus, only a good data structure can store and keep track of nonzero elements, instead of storing the two dimensional matrix as it is. In addition, it was reported that the computation took five days of CPU time on Cray C916 with 100 CPUs and 810 Mbyte main memory [1]. Thus, parallel computing is necessary to reduce the computation time and to accommodate huge matrices in the main memory.

This paper presents a parallel block Lanczos algorithm over GF(2) for the factorization by NFS(number field sieve) method [1]. The algorithm is to achieve efficient parallelization and load balanced data partitioning. Naive partitioning by allocating an equal number of rows/columns of the matrices to each processor results in uneven work load among processors, due to uneven allocation of nonzeros in the sparse matrix. By rearranging rows, we distribute even work load to all processors, and obtain improved performance. In addition, we develop a parallel program with proper group communication scheduling for minimizing communication overhead on distributed- memory parallel machine like a cluster computer.

2. Factorization of a Large Integer by NFS Algorithm

Recent RSA challenge [2,16-18] shows how difficult and complex the factoring problems are, such as a 120-digit key solved in 1993 and a 193-digit key in 2005. The analytical CPU times to solve the keys are impractically long even with today's very high performance computer. Many of the successful results are know to adopt NFS algorithm [8,9].

To factorize a number *N*, NFS algorithm finds (X, Y) integer pairs satisfying the following relationship:

$X^2 \equiv 1 \pmod{N}$, $Y^2 \equiv 1 \pmod{N}$, and $\gcd(XY, N) = 1$ **(1)**

where *gcd* represents the greatest common divisor. The reason of finding such (X, Y) pairs comes from the fact that $\gcd(X - Y, N)$ is a factor of N [10]. In finding all pairs of X and Y, the algorithm starts to find integer pairs of (a, b) that satisfies $a \equiv b \pmod{N}$, where a and b are either squares or a square times of a smooth number. A number *M* is called smooth with respect to a bound δ if its prime factors are under the bound δ.

From the set S consisting of (a_i, b_i), $1 \le i \le k$, we need to find nonempty subsets *S'* such that for some integer *K*

$$\prod_{(a,b)\in S'} a \prod_{(a,b)\in S'} b = K^2$$

Then, the product of the a's and b's can be written in quadratic form like (1) above.

Let B be a matrix of size n_1 x n_2 where the elements are equal to the exponent *modulo 2* of the prime factors. n_1 is the number of prime factors of a and b, and n_2 is the total number of pairs found during the sieving phase.

Now, observe that finding pairs (X, Y) to satisfy (1) is equivalent to choosing a set of columns from the matrix B. Whether or not selecting a column is represented by a vector of size n_2 in which 1 is marked at the corresponding component if the column is selected, 0 otherwise. Thus, the subsets *S'* are obtained by finding the nullvectors *x* satisfying the relation $Bx \equiv 0 \pmod{2}$.

3. Lanczos Algorithm of Finding Nullspace of $Bx = 0$

NFS finds solutions of the equation $Bx = 0$ where B and x are an $n_1 \times n_2$ matrix and an $n_2 \times 1$ vector, respectively. In the factoring application, many elements of *B* are zeros, thus *B* is a sparse matrix. Numerous methods are devised to solve it including Gaussian elimination. Conjugate Gradient(CG) and Lanczos algorithm are known suitable for such a sparse *B* matrix [3,5]. Since Lanczos algorithm requires a symmetric matrix, a new symmetric matrix $A = B^T B$ is generated. All the solutions of $Bx = 0$ also satisfy the relation of $Ax = 0$ in this case. Extended version of the algorithm is called block Lanczos algorithm, which finds *N* vectors at a time. Practically, *N* is chosen as the word size of a computer used. Detailed derivation and theoretical background can be found in [3,10].

The block Lanczos algorithm [4,10] has been developed for the factorization. Our implementation was based on the algorithm in [4], shown in Figure 1. Especially, for the solution of NFS, the number space is limited to $Z_2 = \{0,1\}$ of GF(2), thus, the matrices *A* and *B* consist of 0s and 1s, and numerical computation can be converted to logical operations.

Time complexity of the algorithm is introduced now. Let d be the average count of nonzeros per column in *C*. We assume that $n = n_1 \approx n_2$ for simplicity. Since we are dealing with matrices of big columns and rows, it is not unusual that $n_1, n_2 >> N$ where N is the word size of the computer for parallel bitwise computation to be explained later. Let m denote the number of iterations of the block Lanczos algorithm, which is approximately $m = n / N$ [10]. The overall time complexity is obtained as [10]

$$T_{Block\ \ Lanczos} \approx O(m(nd)) + O(m(nN))$$

Since $n >> N$ and $m = n / N$, the complexity of the algorithm is

$$T_{BlockLanczos} = O(\frac{n^2 d}{N}) + O(n^2) \cdot \qquad \textbf{(2)}$$

4. Parallel Block Lanczos Algorithm

Parallel block Lanczos algorithm is established by including bit-parallel operations of the matrix computation, data partitioning with proper communication, and work load balancing to achieve greatest utilization of the computing resources.

4.1 Bit-Parallel Operations by Logical Operations

If the values in the matrix computation are restricted to {0,1} under GF(2), the matrix computation can be done by simple logical operations such as AND and XOR(exclusive-or). Computation under GF(2) of multiplication and addition/subtraction can be performed by logical operations AND(&) and XOR($\wedge$). Multiple logical values can be computed simultaneously by packing them into one word and such bitwise operations speeds up by *N* times with better use of the main memory assuming an N -bit vector is stored in one integer word.

The matrix of *E = CD* can be found by computing their *inner products*. In the parallel Lanczos algorithm, *outer products* instead of inner products are computed and added to the corresponding *E*-components, until the complete results of the multiplication are found as shown below.

$$E = \sum_{k=1}^{m} c_k \cdot d_k$$

where c_k and d_k represent vectors composed of the k th column of *C* and the k th row of *D*, respectively. While the cost is the same as the inner product method, the computation of outer product in each stage can be done *in parallel* as summarized in Figure 2 [4].

4.2 Data Partitioning and Interprocessor Communication

Parallel algorithm shown in Figure 1 has been developed that consists of a few types of parallelization with associated group communication, depending on the matrix and vector involved as described below. After the partitioning and allocation of matrix to processors, data residing in separate processors are frequently needed. The broadcast, reduce, scatter, and so on in the algorithm are functions for the group communication.

Input: $B_{part} : (n_1 / nprocs) \times n_2 , Y :\ n_2 \times N$ Output: X and V_m
Initialize: $W_{-2}^{inv\ \ N\times N} = W_{-1}^{inv} = 0 ;\ (V_{-2}^{n2\times N})_{part} = (V_{-1})_{part} = 0$ $BV_{-1part}^{(n1/nprocs)\times N} = 0 ; SS_{-1}^{T}{}_{N\times N} = I_N ; X_{part}^{(n2/nprocs)\times N} = 0$ *Broadcast(Y)* $V_{0some} = (B_{part})^T * (B_{part} * Y) ;$ V_0 =*Reduce*(V_{0some});V_{0part} =*Scatter*(V_0) *Broadcast*(V_0) $(BV_0)_{part} = B_{part} * V_0$

$(Cond_0^{N\times N})_{some} = (BV_{0\,part})^T * (BV_0)_{part}$

$(Cond_0)$ =*Reduce*($(Cond_0)_{some}$)

$i = 0$

While $Cond_i \neq 0$ do

if(node==0)

$[W_i^{inv}, SS_i^T]$ =generateMatrixWandS($Cond_i, SS_{i-1}^T, N, i$)

Reduce($(V_{ipart})^T * V_{0\,part}$))

Broadcast($W_i^{inv} * (V_i^T * V_0)$)

$X_{part} = X_{part} + V_{ipart} * (W_i^{inv} * (V_i^T * V_0))$

$(B^T BV_i)_{some} = (B_{part})^T * BV_{ipart}$

$B^T BV_i$ =*Reduce*($(B^T BV_i)_{some}$)

Broadcast($B^T BV_i$)

$C^{N\times N}$ =*Reduce*($(BV_{ipart})^T * (B_{part} * (B^T BV_i))$)

if(node==0) { $K_i^{N\times N} = C^{N\times N} * SS_i^T + Cond_i$

$D_{i+1}^{N\times N} = I_N - W_i^{inv}(K_i)$

$E_{i+1}^{N\times N} = W_{i-1}^{inv}(Cond_i * SS_i^T)$

$F_{i+1}^{N\times N} = -W_{i-1}^{inv}(I_N - Cond_{i-1} * W_{i-1}^{inv})(K_{i-1})SS_i^T$ }

Broadcast(SS_i^T); *Broadcast*(D_{i+1})

Broadcast(E_{i+1}); *Broadcast*(F_{i+1})

$(V_{i+1})_{part} = (B^T BV_i)_{part} * SS_i^T + (V_i)_{part} * D_{i+1}$

$+ (V_{i+1})_{part} * E_{i+1} + (V_{i-2})_{part} * F_{i+1}$

V_{i+1} =*Gather*($(V_{i+1})_{part}$)

Broadcast(V_{i+1})

$(BV_{i+1})_{part} = B_{part} * V_{i+1}$

$(Cond_i)_{some} = ((BV_{i+1})_{part})^T * (BV_{i+1})_{part}$

Reduce($(Cond_i)_{some}$)

$i = i + 1$

if(node==0) $V_m = V_i$

X =*Gather*(X_{part})

Return X and V_m

Figure 1. The pseudo code of parallel Block Lanczos algorithm

As the first and most fundamental way of parallelization, matrix B is divided into P $(n_1/P)\times n_2$ submatrices (each is denoted by B_{part}) and allocated to P processors. $B_{part}Y$ is computed in each processor with a common (unpartitioned) matrix Y, where Y is supplied to every processor by a broadcast. Figures 3 & 4 depict the details for such matrix multiplication, where only two processors are used for simplicity. Another type of parallelization is devised for the computation of $B^T \cdot BY$. Since B_{part} is stored in each processor, the partial computation of $B^T \cdot BY$ can be done in each processor. Now matching part of BY in each processor is used and the partial product is evaluated. Then, since each processor retains an incomplete product, the complete matrix is obtained by combining them together using the collective communication *Reduce*.

Description of other types of parallelization [6] is omitted here.

Input: C and D of sizes $p\times q$ and $q\times N$, respectively Output: $E = CD$ where E is a $p\times N$ vector.

Initialize as E = **0.**
for $i = 1,\cdots, p$ **do**
  for $k = 1,\cdots, q$ **do**
  if $c_{ik} = 1$ **then**
  $e_k = e_i \wedge d_k$

Figure 2. Parallel matrix multiplication by outer product computation

4.3 Load Balancing Strategy

Since the outer product computation in $E = CD$ omits XOR operation if the corresponding component of C is zero as shown in Figure 2, the actual work load in the sparse matrix multiplication relies on the number of nonzero elements in the matrix. Thus, naive partitioning that allocates even rows/columns of B to individual processors does not guarantee even distribution of work load. In our algorithm, an equal number of rows to each node is assigned, and then rows are later interchanged among nodes in such a way that each partition contains approximately the same number of nonzeros. The swapping does not alter the nullspace of $Bx = 0$. Figure 5 illustrates the scheme where an 8X6 matrix B and a 6X2 matrix Y are employed. Only nonzero components of B are shown, moved to the left of the corresponding rows for illustration. Straightforward method (called *static partitioning*) of B shown in Figure 5a, which only divides the matrix with an equal number of contiguous rows (four rows in this case), leaves 18 and 13 nonzeros to Node1 and Node 2, respectively. However, *load balanced partitioning* shown in Figure 5b allows 15 and 16 nonzero elements, thus, the

computational load of two nodes is nearly equal. In the implementation, for both simplicity and load balancing in the remaining computation, an equal number of rows to each node is to be assigned at first. Then, rows are later interchanged among them as depicted in Figure 6. The swapping does not alter the nullspace of $Bx = 0$.

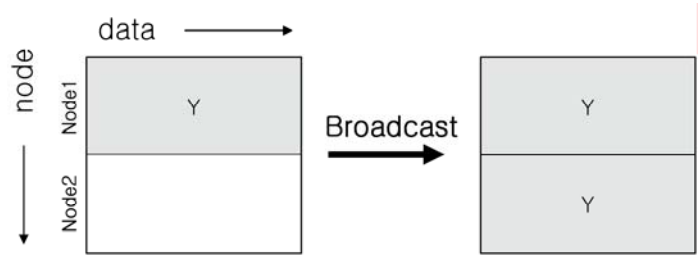


Figure 3. Broadcast of matrix/vector

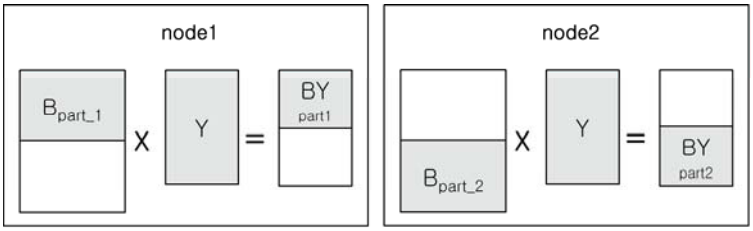


Figure 4. Parallel computation of Type-B

We know there remains more to be done for perfect balancing. *B* is known at the start of the execution and never changes, but other matrices and vectors in the algorithm change while the computation is in progress; thus, it is not possible to partition with even load (nonzeros) in advance. Partitioning them equally during the iterations seems too complicated, thus, only *B* is equally partitioned.

4.4 Time Complexity of the Algorithm

Let α, β, and γ be the setup time, the inverse of bandwidth of the network, and a unit computation cost of one REDUCTION operation, respectively. The complexities of various group (collective) communications involving words under P processors are modeled respectively as

$$T_{scatter}(P,n) = \lceil \log(P) \rceil \alpha + \frac{P-1}{P} n\beta$$

$$T_{gather}(P,n) = \lceil \log(P) \rceil \alpha + \frac{P-1}{P} n\beta$$

$$T_{broadcast}(P,n) = \lceil \log(P) \rceil (\alpha + n\beta)$$

$$T_{reduce}(P,n) = \lceil \log(P) \rceil (\alpha + l\beta + n\gamma)$$

Overall communication cost per iteration of the algorithm in Figure 1 is summed to

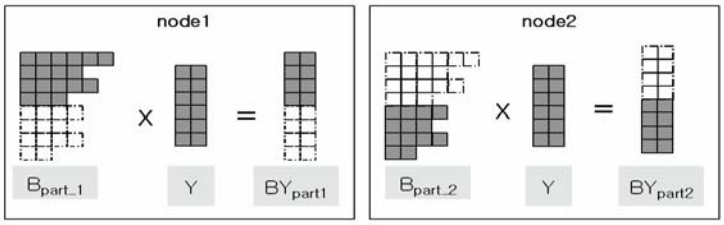
$$T_{Lanczos-comm} =$$

$$2(\lceil \log(P) \rceil (\alpha + n\beta)) + 5(\lceil \log(P) \rceil (\alpha + N\beta)) +$$

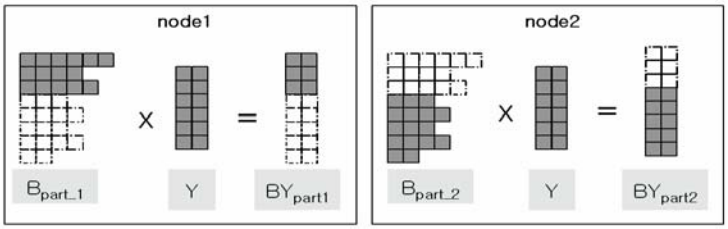
$$3(\lceil \log(p) \rceil (\alpha + N\beta + N\gamma)) + (\lceil \log(P) \rceil (\alpha + n\beta + n\gamma)) +$$

$$m * (\lceil \log(P) \rceil \alpha + \frac{P-1}{P}(\frac{n}{P})\beta)$$

$$\approx O(\lceil \log(P) \rceil (\alpha + n\beta + n\gamma))$$



(a) Straightforward/naive partitioning



(b) Partition with even load distribution

Figure 5. Two data partitioning strategies

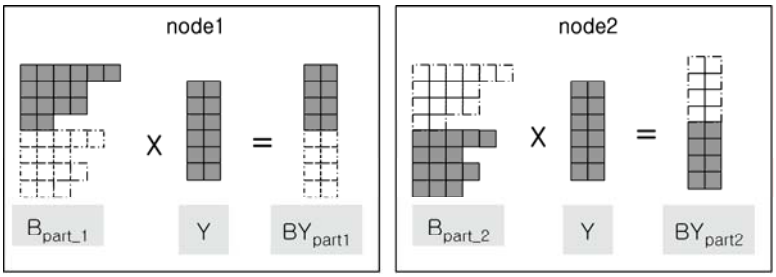


Figure 6. New partitioning that takes into account both the number of rows and nonzero counts.

Under the assumption of perfectly load balanced condition, pure computation cost can be estimated by dividing the sequential complexity by *P*. Thus, the execution time is found as [4,5]

$$T_{Lanczos-comp} = O(m(\frac{nd}{P})) + O(m(\frac{nN}{P})).$$

The total running time is estimated as

$$T_{ParrelLanczos} = T_{Lanczos-comp} + T_{Lanczos-comm} =$$

$$O(m(\frac{nd}{P})) + O(m(\frac{nN}{P})) + O(\lceil \log P \rceil(\alpha + n\beta + n\gamma)m).$$

Applying the relationship of $m = n/M$ results in the total complexity as follows:

$$T_{ParrelLanczos} = O(\frac{n^2 d}{NP}) + O(\frac{n^2}{P}) + O(\lceil \log P \rceil \frac{\alpha n + n^2\beta + n^2\gamma}{N}) \quad \textbf{(3)}$$

Due to the term log *P*, the speedup may not improve linearly with the increase of parallel computing processors.

5 Experimental Results with Discussion

The experiments have been performed on two 8-node clusters; one with Gigabit ethernet interconnection, and another with Myrinet. Processors are 1.6GHz Intel Pentium 4s with 256Mbyte main memory for Gigabit cluster. Myrinet cluster consists of 1.83GHz AMD Athlon XP 2500+ CPUs with 1Gbyte memory. We install MPICH 1.2.6.13 and MPICH_gm 1.2.5.12 [13,14]

Input matrices *B*s are synthetically generated by random placement of 1s with the restriction of *d* nonzeros per column. "t400k" is the largest matrix to fit to the main memory of a node in the cluster. Matrices have a row slightly larger than the column to avoid singularity like 100000 x 100100, 200000 x 200100, 400000 x 400100 for t100k, t200k, and t400k, respectively.

Previous results [4] were obtained on a shared memory parallel computer of SGI Origin 3800 with smaller inputs, thus, they can not be directly compared with ours. We instead compare our results of the load balanced scheme to those of straightforward partitioning algorithm.

Three sets of inputs are applied to the parallel computation with both static partitioning and load balanced allocation. The load balanced method always delivers better results with at least 6.30% up to 14.50% reduction in the execution time as shown in Table 1. The percentage values do not show any tendency as the matrix size grows. One of the reasons could be the fact that partial products computed by individual nodes during the matrix-matrix and matrix-vector computation have uneven number of 0s and 1s, thus, individual processors may not consume the same amount of computing time. In other words, only *B* matrix is partitioned with equal work load in the beginning, and all other matrices used in the algorithm are partitioned to have same number of rows/columns without the knowledge of nonzero count.

Table 1. Comparison of CPU times of static partition and load-balanced partition for t200k on the cluster with Myrinet (unit: hour)

No. of nodes	2	4	8
Static partition	3.09	1.84	1.10
Balanced partition	2.90	1.61	0.99
enhancement(%)	**6.30**	**14.50**	**11.03**

The performance is surely affected by network speed, thus, faster networked Myrinet cluster always gives better results than Gigabit ethernet as observed in Figure 7. They also tell that the speedup increases as the increase of processors. However, the efficiency degrades as described below.

A big matrix of 800,000 x 800,100 (t800k, d =35) is employed on a larger cluster in KISTI supercomputer center [12] to observe the performance in wide range of parallel execution. Cluster in KISTI consists of 2.8GHz Intel Xeon DP processors with 3GB main memory interconnected with Myrinet2000. From Figure 8 we can observe 2.37% to 8.01% improvement in the load balanced method. Although the speedup grows by including more processors, the efficiency degrades, because the communication overhead grows faster than the rate of computing time reduction. Thus, if we want to achieve highest efficiency (rather than speedup), either the network should be sufficiently fast, or only a limited number of processors must be used to avoid the performance loss due to the increase in communication overhead.

6 Conclusions

For the factorization of recent RSA keys of over 150 decimal digits with Number Field Sieve, Lanczos algorithm is used. To reduce the computing time and overcome the limit of main memory capacity of a single processor, parallel computation is necessary. This paper presents an efficient method to maximize the performance of parallel Lanczos computation by allocating even work load to every processor taking into account nonzero counts of the matrix *B*, and by properly arranging group communication among processors to reduce the communication overhead. Research of further improvement of the algorithm on highly parallel machines is under progress.

Acknowledgement: This research was supported in part by KOSEF grant (R01-2006-000-11167-0) and in part by ETRI.

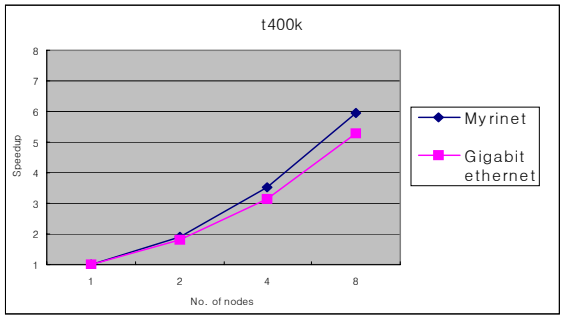


Figure 7. Speedup for t400k matrix

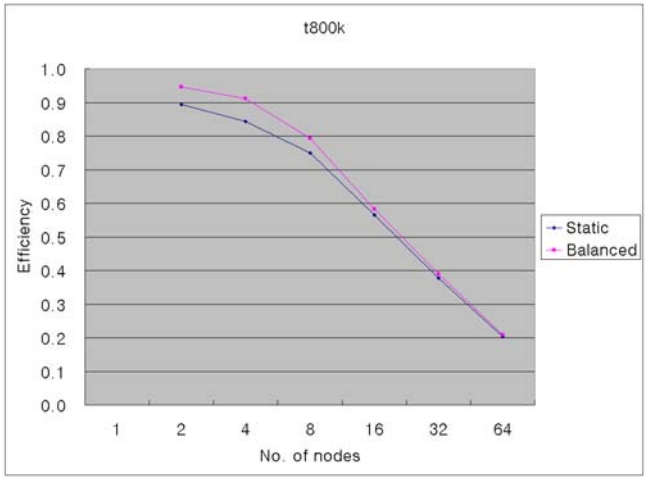


Figure 8. Efficiency of the balanced method

References

1. Cavallar, S., Dodson, B., Lenstra, A. K., Leyland, P. C.,Lioen, W. M., Montgomery, P. L., Murphy, B., te Riele, H., Zimmermann, P.: Factorization of RSA-140 Using the NumberField Sieve. In ASIACRYPT (1999), pp. 195-207.
2. Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W. M.,Montgomery, P. L., Murphy, B., te Riele, H., Aardal, K.,Gilchrist, J., Guillerm, G., Leyland, P. C., Marchand, J.,Morain, F., Muffett, A., Putnam, C., Zim-mermann, P.: Factorization of a 512-bit RSA Modulus. In *Theory and Application of Cryptographic Techniques* (2000), pp. 1-18.
3. Cullum, J. K., and Willoughby, R. A.: *Lanczos Algorithms for Large Symmetric Eigenvalue Computations Vol.I Theory*. Birkhauser, Boston, Basel, Stuttgart, 1985.
4. Flesch, I., Bisseling, R. H.: A New Parallel Approach to the Block Lanczos Algorithm for Finding Nullspaces over GF(2). Master's thesis, Department of Mathematics, Utrecht University, Utrecht, the Netherlands, November 2002.
5. Horn, R. A., Johnson C. R.: *Matrix Analysis*, Cambridge University Press, 1985.
6. Hwang, W.: *Improved Parallel Block Lanczos Algorithm over GF(2) by Load Balancing*, Master Thesis, Korea University, Dec. 2005.
7. Lanczos, C.: An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *Journal of Re-search of the National Bureau of Standards 45*, 4 (Oct. 1950), pp. 255-282.
8. Lenstra, A., Lenstra, H. J., Manasse, M., Pollard, J.: The number field sieve. In *22nd Annual ACM Symposium on the Theory of Computation* (1990), pp. 564-572.
9. Montgomery, P. L.: Square roots of products of algebraic numbers. In *Mathematics of Computation 1943-1993: a Half-Century of Computational Mathematics* (May 1994), Walter Gautschi, Ed., Proceedings of Symposia in Applied Mathematics, American Mathematical Society, pp. 567-571.
10. Montgomery, P. L.: A Block Lanczos Algorithm for Finding Dependencies over GF(2). In *EUROCRYPT '95* (1995), vol. 921, pp. 106-120.
11. Kim, D., Kim, D.: Fast Broadcast by the Divide-and-Conquer Algorithm, 2004 *IEEE International Conference on Cluster Computing*, San Diego, USA, page 487-488, September. 2004.
12. KISTI Supercomputing Cener: http://www.ksc.re.kr
13. MPI: A Message-Passing Interface Standard: MPI Forum, 1995.
14. MPICH - A portable implementation of MPI.http://www.mcs.anl.gov/ mpi/mpich
15. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public key cryptosystems, *Comm. ACM* 21 (Feb. 1978), pp. 120-126,
16. RSA Laboratories: RSA Security Home page - http://www.rsasecurity.com Dec. 2005
17. RSA Laboratory: Security Estimates for 512-bit RSA, RSA Data Security Inc.
18. RSA Lab.: http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html, 1995.