Parallelization of the Lanczos Algorithm on Multi-Core Platforms

Souvik Bhattacherjee and Abhijit Das

Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur
India 721302
{souvikb,abhij}@cse.iitkgp.ernet.in

Abstract. In this paper, we report our parallel implementations of the Lanczos sparse linear system solving algorithm over large prime fields, on a multi-core platform. We employ several load-balancing methods suited to these platforms. We have carried out process-level and thread-level parallel implementations under two different arithmetic libraries, and the best speedup obtained is 6.57 on eight cores. To the best of our knowledge, no implementation of the Lanczos algorithm on a multi-core platform is ever reported in the literature. Moreover, we seem to have achieved significantly larger speedup compared to all previously reported implementations of this algorithm.

Key words: sparse linear system, Lanczos algorithm, modular arithmetic, prime field, multi-core machine, parallelization, load balancing

1 Introduction

The discrete logarithm problem over finite fields serves as the basis of several cryptographic primitives. For example, the Diffie-Hellman key-agreement protocol, the ElGamal public-key cryptosystem and the digital signature algorithm (DSA) [1] rely on the difficulty of solving the discrete logarithm problem, for their security. The fastest known algorithms for solving the discrete logarithm problem require the solution of large sparse linear systems over finite rings. As the size of the system of equations increases, standard Gaussian elimination becomes impractical. Some alternative methods prove to be computationally more attractive than Gaussian elimination, particularly for large and sparse linear systems. Efficient implementations of these iterative system solvers are quite challenging, and the linear-algebra phase often turns out to be the practical bottleneck in the context of solving the discrete logarithm problem. The Lanczos method [2–5] and the Wiedemann method [6–8] are two iterative system solvers that outperform Gaussian elimination for large sparse linear systems. In this paper, we concentrate upon solving linear systems modulo primes q using the Lanczos algorithm.

The sieving part of standard discrete-logarithm algorithms turns out to be massively parallelizable. On the contrary, the system-solving part offers some

resistance to massive parallelization. The main focus of this paper is the establishment of good parallelization potentials of the Lanczos system solver, at least for a limited number of processing elements.

Published results pertaining to parallelizing the Lanczos algorithm are either abstract in nature [9], or focused towards systems over $\mathbf{GF}(2)$ [10]. The best speedup is obtained by [10] and is about 6. The parallel Lanczos implementation over large prime fields, reported by [11], achieves a modest speedup of about 4.5 on 8 processors and a speedup of about 9 on 32 processors. A common feature of all these implementations is that they have been carried out in distributed environments.

In this paper, we report our parallel implementation of the Lanczos algorithm over large finite fields in multi-core shared-memory architectures. We perform both thread-level parallelism using Pthreads and OpenMP [12] and process-level parallelism using shared memory and semaphores. Our process-level implementations outperform our thread-level implementations in terms of scalability. Using novel load-balancing ideas, we have been able to achieve a speedup of 6.57 on 8 cores.

The rest of the paper is organized as follows. In Section 2, we briefly describe the standard Lanczos algorithm. Implementation details of the sequential Lanczos algorithm are presented in Section 3. In Section 4, we describe our parallel implementations with emphasis on load-balancing strategies and issues involved in thread-level and process-level parallelism. The experimental results are presented in Section 5. We conclude the paper in Section 6 after highlighting some directions for future research.

2 The Lanczos Algorithm

We are given an $m \times n$ matrix B over a prime field $\mathbf{GF}(q)$ with $m > n$ to represent the linear system

$$B\boldsymbol{x} \equiv \boldsymbol{u} \pmod{q}. \tag{1}$$

We assume that the equations are consistent and $\boldsymbol{u}$ is in the column space of B. The computation of discrete logarithms requires the solution $\boldsymbol{x}$ to be unique, that is, the matrix B to be of full column rank. Since the system is overdetermined, this requirement is ensured with a high probability. The Lanczos algorithm is classically applicable to systems of the form

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{2}$$

where A is a symmetric, positive-definite matrix over the field of real numbers. In order to adapt this algorithm to the case of finite fields, we transform Eqn.(1) to Eqn.(2) by letting

$$A = B^t B, \tag{3}$$

$$\boldsymbol{b} = B^t \boldsymbol{u}, \tag{4}$$

where B^t denotes the transpose of B. Now, A is a symmetric matrix, but the requirement of positive definiteness makes no sense in a setting of finite fields. The algorithm continues to work if we instead require that $\boldsymbol{w}_i^t A \boldsymbol{w}_i \neq 0$ for $\boldsymbol{w}_i \neq 0$. If the modulus q is large, this condition is satisfied with a very high probability. A solution of Eqn.(1) is definitely a solution of Eqn.(2). The converse too is expected with a high probability.

The standard Lanczos algorithm solves Eqn.(2) by starting with the initializations: $\boldsymbol{w}_0 = \boldsymbol{b}$, $\boldsymbol{v}_1 = A\boldsymbol{w}_0$, $\boldsymbol{w}_1 = \boldsymbol{v}_1 - \boldsymbol{w}_0(\boldsymbol{v}_1^t A\boldsymbol{w}_0)/(\boldsymbol{w}_0^t A\boldsymbol{w}_0)$, $a_0 = (\boldsymbol{w}_0^t \boldsymbol{w}_0)/(\boldsymbol{w}_0^t A\boldsymbol{w}_0)$, $\boldsymbol{x}_0 = a_0\boldsymbol{w}_0$. Subsequently, for $i = 1, 2, 3, \ldots$, the steps in Algorithm 1 are repeated until $\boldsymbol{w}_i^t A\boldsymbol{w}_i = 0$, which is equivalent to the condition $\boldsymbol{w}_i = 0$ (with high probability). When this condition is satisfied, the vector $\boldsymbol{x}_{i-1}$ is a solution to Eqn.(2).

Algorithm 1 An iteration in the Lanczos Algorithm

1: $\boldsymbol{v}_{i+1} = A\boldsymbol{w}_i$

2: $\boldsymbol{w}_{i+1} = \boldsymbol{v}_{i+1} - \dfrac{\boldsymbol{w}_i(\boldsymbol{v}_{i+1}^t A\boldsymbol{w}_i)}{(\boldsymbol{w}_i^t A\boldsymbol{w}_i)} - \dfrac{\boldsymbol{w}_{i-1}(\boldsymbol{v}_{i+1}^t A\boldsymbol{w}_{i-1})}{(\boldsymbol{w}_{i-1}^t A\boldsymbol{w}_{i-1})}$

3: $a_i = \dfrac{(\boldsymbol{w}_i^t \boldsymbol{b})}{(\boldsymbol{w}_i^t A\boldsymbol{w}_i)}$

4: $\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + a_i\boldsymbol{w}_{i-1}$

For more details on the Lanczos algorithm, we refer the reader to [5].

3 Sequential Implementation

In this section, we describe our sequential implementation of the standard Lanczos algorithm, modulo a large prime q. This sequential implementation is parallelized later. Systems of linear equations were available to us from an implementation of the linear sieve method [13]. Larger systems were generated randomly in accordance with the statistics followed by the entries in the linear-sieve matrices.

3.1 Representing the Matrix B

Due to the nature of the sieving algorithm, the coefficients of the system of equations have very small positive or negative magnitudes. For the current set of matrices, we have the coefficients $\in [-2, c]$, where $c \leq 50$. Most of these coefficients are ± 1. Each such coefficient can be stored as a signed single-precision (32-bit) integer.

Matrices generated by the sieve algorithms are necessarily very sparse. In our case, each row contains only $\mathrm{O}(\log q)$ non-zero entries. We store the matrix in a compressed row storage (*CRS*) format, where each row is represented by an array of coefficient-column index pairs (*val, col_ind*) for the non-zero entries only. We concatenate the arrays for different rows into a single one-dimensional

array, and use a separate array *row_ptr* to mark the start indices of the rows in the concatenated (*val, col_ind*) array. If there are N non-zero entries in the matrix, then the concatenated array of (*val, col_ind*) pairs requires a storage proportional to N. The array *row_ptr* demands a size of $m + 1$ for pointing to the start indices of the rows. As an example, the CRS format of the matrix

$$B = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 \\ 3 & 9 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 \\ 3 & 0 & 8 & 0 & 5 \\ 0 & 8 & 0 & -1 & 0 \\ 0 & 4 & 0 & 0 & 2 \end{bmatrix}$$

is shown in Fig. 1. We assume that array indexing starts from 1.

val	10	−2	3	9	7	8	7	3	8	5	8	−1	4	2
col_ind	1	5	1	2	2	3	4	1	3	5	2	4	2	5

row_ptr	1	3	5	8	11	13	15

Fig. 1. The Compressed Row Storage format of B

val	10	3	3	9	7	8	4	8	8	7	−1	−2	5	2
row_ind	1	2	4	2	3	5	6	3	4	3	5	1	4	6

col_ptr	1	4	8	10	12	15

Fig. 2. The Compressed Column Storage format of B

We apply the Lanczos method on the modified matrix $A = B^t B$. Since A is expected to be significantly less sparse than B, we do not compute A explicitly. We instead store both B and B^t. The multiplication $A\boldsymbol{w}_i$ is computed as $B^t(B\boldsymbol{w}_i)$. The CRS format of storage of B is not suitable during the outer multiplication. We need B^t in the CRS format, or equivalently B in the compressed column storage (CCS) format, as illustrated in Fig. 2. Although this leads to duplicate storage for the same matrix, the resulting overhead in the running time turns out to be negligible, and the extra space requirement tolerable.

3.2 The Structure of the Matrix *B*

The most time-consuming step in the Lanczos iteration is the matrix-vector multiplication $A\boldsymbol{w}_i = B^t(B\boldsymbol{w}_i)$. Optimizing strategies to speed up this operation call for an investigation of the structure of the matrix B. As we see later, this structure also has important bearings on load balancing.

The n columns of B are composed of two blocks. The first t columns of B correspond to small primes in the factor base (see the linear-sieve algorithm [13]). The remaining columns of B correspond to the $2M+1$ variables arising out of the sieving interval. For $1 \leq i \leq t$, the i-th column heuristically contains about m/p_i

non-zero entries, where p_i is the i-th prime. For small values of i, these columns are, therefore, rather dense. The last $2M+1$ entries in each row contain exactly two non-zero entries which are -1. The two -1 values may happen to coincide, resulting in a single non-zero entry of -2, but this event has a very low probability and is ignored in our subsequent discussion. Each of the last $2M+1$ columns contains $2m/n$ non-zero entries on an average. For $m \leq 2n$, this value is ≤ 4.

About three-fourth of the non-zero entries of B are $+1$. Most of the remaining non-zero entries are -1. While multiplying a vector by B or B^t, one gives special attention to the matrix entries ± 1. In a typical sum of the form $\sum_r b_r w_{ir}$ with non-zero entries b_r of B and entries w_{ir} of the vector $\boldsymbol{w}_i$, the addition of the product $b_r w_{ir}$ is replaced by the addition of w_{ir} if $b_r = 1$ or by the subtraction of w_{ir} if $b_r = -1$. Moreover, if we *know* beforehand a particular value of b_r (like $b_r = -1$ in the last $2M+1$ rows of B^t), a multi-way branching based upon the value of b_r may be replaced by a single unconditional operation (like subtraction for $b_r = -1$). Finally, as pointed out in [11], a good strategy to speed up the matrix-vector multiplication is to perform the modulo q reduction after the entire expression $\sum_r b_r w_{ir}$ is evaluated. Since b_r are single-precision integers and w_{ir} are general elements of $\mathbf{GF}(q)$, the word size of $\sum_r b_r w_{ir}$ is only slightly larger than that of q, even when there are many terms in the sum (like during multiplication by the first row of B^t).

During the matrix-vector multiplication operation, when a particular row of B or B^t is to be multiplied by a vector, only those vector entries which correspond to the indices of the non-zero matrix entries are needed for multiplication. Since B is a sparse matrix, these indices are usually widely apart. This, in turn, indicates that almost every multiplication of a matrix entry by a vector entry encounters a page fault while accessing the vector entry. The effect of these page faults is more pronounced in a parallel implementation, where multiple processes or threads vie for shared L2 cache memory. This problem can be solved by rearranging the rows and columns of B so as to bring the non-zero entries as close to each other as possible. For linear-sieve matrices, the $m \times (2M+1)$ block of the entries with value -1 readily yields to such rearrangement possibilities. Our experience suggests that it is possible to bring the -1 values close to each other for over 50% of the occurrences.

It is important to comment here that although the above optimization tricks are applied to matrices generated from the linear-sieve method, they apply identically to matrices generated by other sieving algorithms. The cubic-sieve method [13] generates matrices with the only exception that exactly three (instead of two) non-zero entries of -1 are present in the last $2M+1$ columns in each row. Matrices generated by the number-field-sieve method [14] do not contain the block of -1's. They instead contain two copies of the block resembling the first t columns of linear-sieve matrices. One of these blocks corresponds to small rational primes and has small positive entries, whereas the other block corresponds to small complex primes and has small negative entries. In any case, most of the non-zero entries of the matrices are ± 1.

4 Parallel Implementation

We now elaborate our efforts to parallelize the Lanczos iteration given as Algorithm 1. The different iterations of the Lanczos loop are inherently sequential, that is, no iteration may start before the previous iteration completes. We instead parallelize each iteration separately. More precisely, each of the following operations is individually parallelized: matrix-vector product, vector-vector product, scalar-vector product, vector-vector addition and subtraction, and vector copy.

4.1 Load Balancing

Suppose that each basic operation is distributed to run in parallel in P processors. In order to avoid long waits during synchronization, each parallelization step must involve careful load balancing among the P processors. For most of the operations just mentioned, an equitable load distribution is straightforward. For example, each addition, subtraction or multiplication operation on two (dense) vectors of size n is most equitably distributed if each processor handles exactly n/P entries of the operand vectors.

Non-trivial efforts are needed by the matrix-vector product $\boldsymbol{v}_{i+1} = B^t(B\boldsymbol{w}_i)$. This operation is actually carried out as two matrix-vector products:

a. $\boldsymbol{z} = B\boldsymbol{w}_i$
b. $\boldsymbol{v}_{i+1} = B^t\boldsymbol{z}$

The first of these products involves multiplication of the vector $\boldsymbol{w}_i$ by the rows of B. The rows of B do not show significant variations among one another, in terms of both the number of non-zero entries and the values of these entries. As a result, it suffices to distribute an equal number m/P of rows to each processor. For small values of P (like $P \leq 8$), the number m/P is large enough to absorb small statistical variations among the different rows.

The second product involves multiplication of $\boldsymbol{z}$ by the columns of B. There exist marked variations among the different columns of B, in terms of both the count of non-zero entries and the values of these entries. A blind distribution of n/P columns of B to each processor leads to very serious load imbalance among the processors. The implementation of [10] starts with the distribution of an equal number of columns to each processor. It subsequently interchanges columns among the processors until each processor gets an approximately equal share of non-zero entries. In the end, this strategy achieves both an equal number of columns and an equal number of non-zero entries, for each processor.

Since we work in a shared-memory architecture, data transmission delays are not of concern to us, so we drop the requirement of equality in the number of columns across different processors. Second, we consider the fact that an equality in the number of non-zero entries is not a good measure of the actual load, since our implementation handles the three types of non-zero values of a matrix entry (1, -1 and anything else) differently. We assign appropriate weights to these three types of entries, based upon the time needed to perform the respective arithmetic operations by the underlying multiple-precision integer library. A scheme that worked well for our implementations is to assign a weight of 1.0 to

each matrix entry $+1$, a weight of 1.2 to each matrix entry -1, and a weight of 1.5 to a non-zero entry other than ± 1.

Such a system of assigning a fixed weight to each non-zero value, although justifiable and capable of producing nearly optimal solutions, looks somewhat heuristic and can be improved upon. The trouble is we do not explicitly consider the dependence of the timing of an arithmetic operation on the values of its operands. For example, each non-zero entry of B other than ± 1 involves a multiplication, the running time of which may depend on the value of the matrix entry. Moreover, the value of the accumulating sum also plays a role in the timing. An exact characterization of this dependence appears well beyond the control of a programmer, because several factors (including the implementation of the multiple-precision library, the effects of the compiler, the instruction set of the processor) play potentially important roles in this connection.

In view of this, we suggest the following strategy. We actually record the timing against each non-zero entry of B, when B^t is multiplied by a vector $\boldsymbol{z}$. This vector $\boldsymbol{z}$ may be chosen randomly from $\mathbf{GF}(q)^m$. Another strategy is to run a few iterations of the Lanczos loop itself and record the element-wise timings during the product $B^t\boldsymbol{z}$. (For a few initial iterations, the vector $\boldsymbol{w}_i$ remains sparse, so timing data for, say, the first ten iterations may be discarded.) Data obtained from a reasonable number of multiplications $B^t\boldsymbol{z}$ is averaged to assign a *true* weight to each non-zero entry of B.

The weight of a column is the sum of the weights of its non-zero entries. We distribute columns to P processors in such a way that each processor receives an equal share of the total load. To each processor, we first assign a chunk from the first t columns. The resulting load imbalance is later repaired by distributing chunks from the remaining $2M+1$ columns. In a linear-sieve matrix, $2M+1 \gg t$ (for example, $t = 7,000$ and $2M+1 = 60,001$ for the matrix we have from [13]). Moreover, each of the last $2M+1$ columns is rather low-weight. As a result, a fine load balancing among the processors is easy to achieve.

4.2 Load Balancing in Parallel

A limitation of the above load-balancing scheme is that the timing figures are measured by a sequential version of the code. However, we apply these timing data in a parallel environment which may be quite different from a sequential environment. This means that operations that are equitably distributed according to sequential timing data may lead to improper load balancing. We solve this problem by measuring the timings from runs of the matrix-vector multiplication in parallel. We start with a crude load-balancing scheme in which the columns of B are alternately distributed to the processes/threads. Unfortunately, a practical implementation of this strategy did not yield better performance in our experiments. Nonetheless, we mention this strategy here with the hope that it may prove to be useful in related settings.

4.3 Thread-level vs Process-level Parallelism

We carried out our implementations on an eight-core machine using thread-level parallelism (TLP) provided by Pthreads and OpenMP, and process-level parallelism (PLP) involving shared memory and semaphores. The Lanczos algorithm was initially programmed using functions from the GMP [15] library, prefixed with `mpz`. These functions allocate dynamic memory to output operands, giving rise to heap contention. Assigning each thread its own heap, as can be achieved by the Hoard memory allocator [16], eliminated the problem. As an alternative approach, we considered the fast, low-level `mpn` routines provided by GMP. Unlike `mpz`, the `mpn` functions do not allocate memory to its output operands and assume that an appropriate amount of memory is already allocated to each output operand. Before starting the Lanczos loop, we allocated memory to all loop variables. This made fast GMP routines usable even in a thread-level implementation, without costly waits arising out of heap contention.

We also investigated an alternative implementation of the Lanczos algorithm, using both TLP and PLP. A proprietary library CCrypto was used for this implementation. This library was written and optimized for 32-bit architectures. With some efforts, it could be ported to 64-bit architectures, but absence of platform-specific optimization tricks resulted in a library somewhat slower than GMP. Since our primary focus was to investigate the parallelization possibilities of the Lanczos algorithm, we continued to work with the slow library. Indeed, the highest speedup figures were obtained by PLP under the CCrypto library.

5 Experimental Results

The computations were carried out on an Intel® Xeon® E5410 dual-socket quad-core Linux server. The eight processors run at a clock speed of 2.33 GHz and support 64-bit computations. The machine has 8 GBytes of main memory and a shared L2 cache of size 24 MBytes across 8 cores. One of the linear systems (q-149) used in testing the algorithm was obtained by the linear-sieve method. Larger systems were generated imitating the distribution of elements in the q-149 matrix. Here, we report our implementation on a $1,600,000 \times 2,200,000$ system (q-512) modulo a 512-bit prime. The thread-level parallelism was obtained separately using Pthreads and OpenMP version 4.3.2. We also exploited process-level parallelism under low-level shared-memory and semaphore constructs. All these implementations used GMP version 4.3.1, for multi-precision computations. The CCrypto library, on the other hand, was used in conjunction with Pthreads (TLP) and also with shared memory and semaphores (PLP).

Fig. 3 shows the process-level and the thread-level speedups for the system q-512, obtained as a function of the number of cores, by GMP and CCrypto. We obtained a maximum speedup of 6.57 using a combination of PLP and CCrypto, whereas a maximum speedup of 5.11 was registered using a combination of PLP and GMP. For the thread-level implementations, the maximum speedup obtained was 4.51 with GMP and 5.96 with CCrypto. From Fig. 3(b), we notice that the speedup figures obtained by Pthreads and OpenMP were

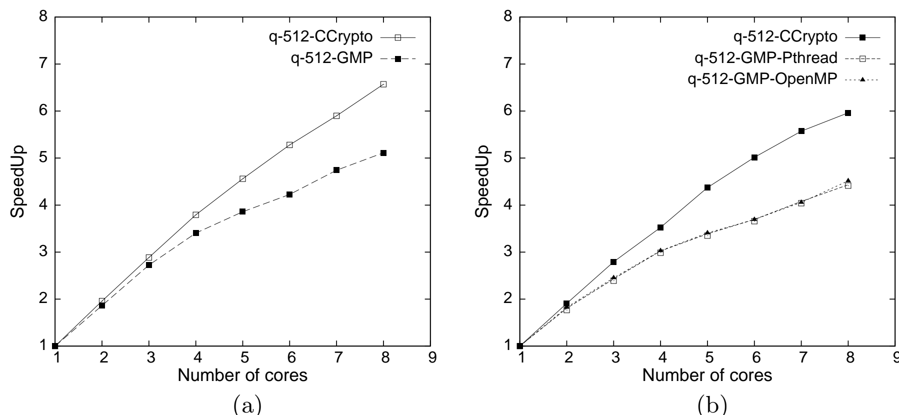

Fig. 3. Speedup: (a) Process-level parallelism (b) Thread-level parallelism

almost identical. However, it remains unexplained why thread-level implementations consistently exhibit somewhat poorer performance than the corresponding process-level implementations.

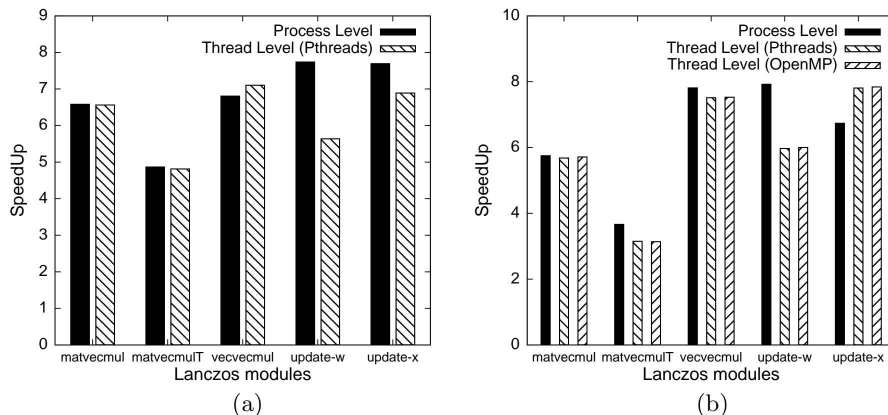

Fig. 4. Speedup obtained by individual modules of Lanczos (a) CCrypto (b) GMP

Fig. 4 shows the speedup values achieved by the major modules of the Lanczos loop. In these figures, `matvecmul` stands for multiplication by B, `matvecmulT` denotes multiplication by B^t, and `vecvecmul` stands for the average over the three products $\boldsymbol{v}_{i+1}^t\boldsymbol{v}_{i+1}$, $\boldsymbol{v}_{i+1}^t\boldsymbol{v}_i$, $\boldsymbol{w}_i^t\boldsymbol{v}_{i+1}$. Finally, `update-w` stands for the computation of $\boldsymbol{w}_{i+1}$ (Step 2 of Algorithm 1) without the vector-vector products, and `update-x` represents the computation of $\boldsymbol{x}_i$ (Steps 3 and 4 of Algorithm 1).

It is evident from Fig. 4 that the matrix-vector products (particularly the multiplication by B^t) constitute the practical bottleneck in the parallelization of the Lanczos algorithm.

6 Conclusion

In this paper, we report an aggressive attempt to parallelize the Lanczos sparse linear system solving algorithm modulo large primes, on a multi-core architecture. Using process-level parallelism in conjunction with shared memory and semaphores, we have been able to achieve a record speedup of 6.57 on eight cores. Our efforts, however, open up a host of questions, the pursuit of which would further our implementation study.

a) The matrix-vector multiplication operation consumes nearly three-fourth of the total parallel execution time and constitutes the practical bottleneck in parallelization attempts. An improved speedup in this operation may reduce the absolute running time considerably, and this calls for further research endeavor tailored to this specific operation only.
b) Use of multiple multi-core machines to parallelize the Lanczos solver indicates a new combination of distributed and shared-memory computation and may lead to more positive results in favor of the parallelizability of the Lanczos algorithm.
c) Structured Gaussian elimination [4] is a technique that reduces the size of a sparse matrix and is often applied before invoking a sparse system solver like the Lanczos algorithm. Typically, structured Gaussian elimination results in smaller but considerably denser matrices. It is an interesting study to compare the performance of the Lanczos algorithm on a (relatively) dense reduced matrix with that on the original sparse matrix.

References

1. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA (1996)
2. Coppersmith, D., Odlyzko, A.M., Schroeppel, R.: Discrete logarithms in GF(p). Algorithmica **1** (1986) 1–15
3. Odlyzko, A.M.: Discrete logarithms in finite fields and their cryptographic significance. In: Advances in Cryptology—EUROCRYPT. Volume 209 of LNCS., Springer (1984) 224–314
4. LaMacchia, B.A., Odlyzko, A.M.: Solving large sparse linear systems over finite fields. In: Advances in Cryptology—CRYPTO. Volume 537 of LNCS., Springer (1990) 109–133
5. Montgomery, P.L.: A block Lanczos algorithm for finding dependencies over GF(2). In: Advances in Cryptology—EUROCRYPT. Volume 921 of LNCS., Springer (1995) 106–120
6. Wiedemann, D.H.: Solving sparse linear equations over finite fields. IEEE Transactions on Information Theory **32** (1986) 54–62

7. Penninga, O.: Finding column dependencies in sparse systems over F_2 by block Wiedemann. Master's thesis, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands (1998)
8. Dumas, J.G., Villard, G.: Computing the rank of large sparse matrices over finite fields. In: Computer Algebra in Scientific Computing CASC, Technische Universität München, Germany (2002)
9. Yang, L.T., Brent, R.P.: The parallel improved Lanczos method for integer factorization over finite fields for public key cryptosystems. In: ICPP Workshops. (2001) 106–114
10. Hwang, W., Kim, D.: Load balanced block Lanczos algorithm over GF(2) for factorization of large keys. In: HiPC. Volume 4297 of LNCS., Springer (2006) 375–386
11. Page, D.: Parallel solution of sparse linear systems defined over GF(p). Technical Report CSTR-05-003, University of Bristol (2004)
12. OpenMP: The OpenMP API Specification for Parallel Programming. (http://www.openmp.org/)
13. Das, A., Veni Madhavan, C.E.: On the cubic sieve method for computing discrete logarithms over prime fields. International Journal of Computer Mathematics **82** (2005) 1481–1495
14. Weber, D.: Computing discrete logarithms with the general number field sieve. In: ANTS: 2nd International Algorithmic Number Theory Symposium (ANTS). Volume 1122 of LNCS., Springer (1996) 337–361
15. GNU: The GNU MP Bignum Library. (http://gmplib.org/)
16. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications. In: ASPLOS. (2000) 117–128