Parallel and Distributed Implementations of the Wiedemann and the Block-Wiedemann Methods over GF(2)

Rahul Roy, Abhijit Das and Dipanwita Roy Chowdhury
Crypto Research Lab, Department of Computer Science and Engineering, IIT Kharagpur, Kharagpur, India

Keywords: RSA, Integer Factoring, GNFSM, Large Sparse Linear Systems, Wiedemann Method, Block-Wiedemann Method, Multi-core Parallelization, Load Balancing, Distributed Parallelization, Computation-communication Overlapping.

Abstract: Finding the prime factors of large composite integers is the fundamental computational problem in number theory. Currently, the fastest known integer-factoring algorithm is the General Number Field Sieve method (GNFSM) which has been used by the research community to factor RSA moduli of sizes 500–800 bits. One of the steps of this method involves finding non-zero solutions of the linear system available from the sieving stage. Since the linear systems involved in GNFSM are necessarily sparse, special iterative system solvers are used. One such solver is called the Wiedemann method. This paper reports our efficient implementation of the Wiedemann method, and its block version. We start with a single-core sequential implementation, and then make efforts to parallelize the implementation to run on multiple cores of a single machine. Special load-balancing techniques are designed to reduce synchronization overheads after each iteration. Finally, we distribute the computation across multiple computing nodes. Our load-balancing ideas are refined, and computation-communication overlapping techniques are explored in order to absorb the communication overheads. Speed-up figures achieved by the different improvements incorporated in our implementations are reported. To the best of our knowledge, we are the first to report distributed implementations of the Wiedemann method.

1 INTRODUCTION

Public-Key Cryptographic (PKC) algorithms like RSA (Rivest et al., 1978) serve as the fundamental security primitives to ensure confidentiality, authenticity, and non-repudiation of Internet communications and data storage. Like other public-key encryption algorithms, RSA uses two keys: one public key or encryption key available to all, and a private key or decryption key which is kept secret. RSA involves four significant steps: (i) key generation, (ii) key distribution, (iii) encryption, and (iv) decryption. The security of RSA depends on the difficulty of factoring the product of two large prime numbers. Many general-purpose integer-factoring algorithms are developed in the last few decades. The fastest known such algorithm is the General Number Field Sieve Method (GNFSM) (Cowie et al., 1996; Case, 2003) which is a generalization of another algorithm known as the Special Number Field Sieve Method (SNFSM) (Lenstra et al., 1993; Montgomery et al., 1997). Besides cryptographic applications, factoring large composite integers is the fundamental computational problem in number theory.

Many integer-factoring algorithms (including GNFSM) require a linear-algebra step. A large linear system of equations is generated in the sieving stage. Subsequently, multiple non-zero vectors in the null space of this system are computed. The systems available from GNFSM are necessarily sparse. Therefore, a study of sparse system solvers is important from number-theoretic and cryptographic points of view.

A way to solve linear systems of equations is *structured Gaussian elimination* (SGE) (Bender and Canfield, 1999; Pomerance and Smith, 1992). SGE can significantly reduce the system size by eliminating many rows and columns, eventually leading to a significantly dense system which is then solved by a general-purpose system solver. Although SGE can be used as a precomputation to the final system-solving algorithm, it is not efficient in practice for very large systems. Current literature suggests that for solving large sparse systems, iterative algorithms collectively

Roy, R., Das, A. and Roy Chowdhury, D.
Parallel and Distributed Implementations of the Wiedemann and the Block-Wiedemann Methods over GF(2).
DOI: 10.5220/0011267700003283
In *Proceedings of the 19th International Conference on Security and Cryptography (SECRYPT 2022)*, pages 540-547
ISBN: 978-989-758-590-6; ISSN: 2184-7711

known as Krylov-space methods (Krylov, 1931) work more efficiently than SGE. Two popular variants of Krylov-space methods are the *Lanczos method* (Lanczos, 1952) and the *Wiedemann method* (Wiedemann, 1986). Both these methods use a black-box model for solving large sparse systems (unlike SGE which exploits the structural properties of the matrices). They perform a linear number of matrix-vector products and other vector operations. Their complexity is measured in terms of the dimensions of the input matrices.

Speeding up the performances of the sparse system solvers involves several optimization techniques. One such optimization technique introduces the concept of blocking for both the methods in order to enhance parallelization possibilities and to get multiple solutions simultaneously. (Coppersmith, 1994) introduces the block version of the Wiedemann method, and (Montgomery, 1995) introduces the block version of the Lanczos method.

One step of the Wiedemann method finds a minimal generating polynomial of a sequence of matrix-vector products. The block version is extended to minimal generating matrix polynomials (Kaltofen, 1995; Villard, 1997). To improve the efficiency of the method, a particular effort is made by (Thomé, 2002) that focuses on optimizing the step of computing the minimal generating matrix polynomial. An initial attempt to implement the block-Wiedemann method in a distributed environment is reported by (Kaltofen and Lobo, 1999). However, the matrices considered are smaller than those generated after the post-sieving stages. In 2009, the block version of the method is implemented in a grid platform by the team of international researchers to break the 768-bit RSA challenge key (Kleinjung et al., 2010). It is reportedly used in the factorization of RSA keys of different other lengths (Bai et al., 2012; Bai et al., 2016). The effort of (Yang et al., 2017) on the factorizations of 381-bit RSA keys is made on a cloud platform, and also uses the block-Wiedemann method for the linear-algebra step. Other papers focus on implementations in multi-core environment (Penninga, 1998) or on linear systems generated over large prime fields (Barbulescu et al., 2014). (Bhateja and Kannan, 2017) propose a cache-optimized version of both the Lanczos and the Wiedemann methods. (Cavallar et al., 2000) and (Chen et al., 2008) both use the block-Lanczos method in their efforts to factor 512-bit RSA keys.

In this paper, we report optimized versions of the Wiedemann and the block-Wiedemann methods, their implementation details, and the experimental results for the sequential, parallel (single-node), and distributed (multi-node) settings. The rest of the paper is organized as follow. In Section 2, the Wiedemann and the block-Wiedemann methods are described. Section 3 describes our implementational details of the Wiedemann and the block-Wiedemann methods, where we describe our load-balancing strategies and other practical optimizations. In section 4, experimental setup and results achieved are shown. Section 5 concludes the paper after enumerating some scopes of extending our study.

2 THE WIEDEMANN AND BLOCK-WIEDEMANN METHOD

This section starts with a detailed foundation of the Wiedemann method, together with the Berlekamp–Massey method which is used for finding the minimal polynomial. This is followed by an introduction to Coppersmith's version of the block-Wiedemann method.

2.1 The Wiedemann Method over GF(2)

Introduced by Douglas Wiedemann (Wiedemann, 1986) to solve large sparse linear systems over finite fields, the Wiedemann method is a Krylov-space method. It is a randomized Las Vegas algorithm that always provides a correct output or reports failure. Unlike the Lanczos method (Lanczos, 1952), it does not require the matrix to be symmetric or positive definite. It involves only the arithmetic operations of GF(2). The Wiedemann method uses an external algorithm called the Berlekamp–Massey method (Berlekamp, 1968; Massey, 1969) for finding minimal polynomials.

Given a matrix B of size $M \times N$ over GF(2) with $M > N$ and a non-zero vector u, we want to solve the linear system

$$Bx \equiv u \pmod 2. \tag{1}$$

The objective is to find multiple solutions for the vector x. The Wiedemann algorithm requires the system to be in the form

$$Ax = b, \tag{2}$$

where A is an $N \times N$ matrix. In order to adapt the algorithm, the input system (1) needs to be converted to equation (2) as

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

and

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

where B^t is the transpose of the input matrix B. The characteristic polynomial of A is defined as $\chi_A(x) =$

$\det(xI - A)$, where I is the $N \times N$ identity matrix. By the Cayley–Hamilton theorem, A satisfies $\chi_A(x)$, that is, $\chi_A(A) = 0$. The minimal polynomial of A is the monic non-zero polynomial of the smallest degree for which $\mu_A(A) = 0$. It is known that $\mu_A(x)$ divides the characteristic polynomial $\chi_A(x)$, that is, $\mu_A(x) \mid \chi_A(x)$ in GF(2)[x]. Wiedemann's method computes the minimal polynomial of A as

$$\mu_A(x) = x^d + u_{d-1}x^{d-1} + u_{d-2}x^{d-2} + \cdots + u_1 x + u_0. \quad (5)$$

Here, $d = \deg(\mu_A(x)) \leq N$. The coefficients of $\mu_A(x)$ are solved using the Berlekamp–Massey algorithm. This requires the computation of $A^k\mathbf{v}$ for a non-zero vector $\mathbf{v}$, and for $k = 0, 1, 2, \ldots, 2d-1$. Since $\mu_A(A) = 0$, for any $k \geq d$, we have

$$A^k v - u_{d-1}A^{k-1}v - \cdots - u_1 A^{k-d+1}v - u_0 A^{k-d}v = 0. \quad (6)$$

A solution for x is obtained by putting $v = b$ as

$$x = -u_0^{-1}(A^{d-1}b + u_{d-1}A^{d-2}b + u_{d-2}A^{k-3}b + u_1 Ab). \quad (7)$$

We consider the solution if $x \neq 0$ and $Ax = 0$. One of the most time-consuming task here is computing the matrix-vector product in each iteration of the two loops.

2.2 The Block-Wiedemann Method over GF(2)

Coppersmith (1994) introduces a block version of the Wiedemann method to increase the scope of parallelization. In the block method, the concept of the scalar sequence by Wiedemann (1986) is replaced by a linearly generated matrix sequence. Subsequently, the minimal matrix polynomial is generated. In order to compute the minimal matrix polynomial, the concept of multivariate Berlekamp–Massey method is introduced by (Coppersmith, 1994; Kaltofen and Yuhasz, 2013). (Kaltofen and Lobo, 1999) uses a homogeneous block Toeplitz system. The Fast Power Hermite-Padé Solver (FPHPS) algorithm of Beckermann et al. (Beckermann and Labahn, 1994) to compute the minimal matrix polynomial is proposed by (Villard, 1997). (Kaltofen and Saunders, 1991) describe an asymptotically faster algorithm than the binary search algorithm that Wiedemann proposes to compute the rank of a black-box matrix over large fields.

In Coppersmith's version of the block-Wiedemann method, The first step (called BW1), consists of computing the sequence

$$A_L{}^{(i)} = (U^T A^i Z). \quad (8)$$

This step starts with two random blocks U of size $N \times m$ and V of size $N \times n$. Multiplying successively by the input matrix A, it computes A^iV, and U is used to compute the projections U^TA^iV of size $N \times n$. To obtain kernel vectors, only the first L coefficients of the sequence are required, where $L = N/m + N/n + O(1)$.

In Step BW2, Coppersmith modifies the Berlekamp–Massey method as the matrix Berlekamp–Massey method or block Berlekamp–Massey method. It takes the sequence $A_L{}^{(i)}$ output by Step BW1, and defines the polynomial $A_L(\lambda) \in GF(2)^{(m \times n)}$ of degree $N/m + N/n + 1$. It generates a matrix sequence $F(\lambda)$ of degree $\lceil \frac{N}{n} \rceil$. This is the main step of the Block Wiedemann algorithm. Unlike the scalar version, this step is the most complex.

Steps BW3 involves $\deg(F)$ matrix-block multiplications and block-vector multiplications. It also performs $\delta(F)$ matrix-vector multiplications, and $\delta(F)$ tests for the nullity of vectors.

3 IMPLEMENTATION DETAILS OF THE WIEDEMANN AND THE BLOCK-WIEDEMANN METHODS OVER GF(2)

In this section, our implementational details of the Wiedemann and the block-Wiedemann methods over GF(2) are explained, including space optimizations, introduction to new load-balancing strategies, and other practical optimizations that help us gain improved speed-up for both the methods.

3.1 Representing the Matrix

To store the sparse matrices, there are various possibilities (Lin et al., 2003; Bhattacherjee and Das, 2010) like the *compressed column-storage format* (CCS), the *compressed row-storage format* (CRS), and the linked-list representation. The CRS and the CCS formats provide efficient storage schemes because of their low memory requirements, so we adopt these formats.

The CRS representation maintains three arrays *value_array*, *column_index*, and *row_pointer*. We scan the matrix A row-wise, and the *value_array* stores the non-zero elements of the matrix. The array *column_index* stores the column numbers of the non-zero values stored in the value array, and *row_pointer* stores the number of non-zero values encountered before the current row of the original matrix A.

The *value_array* in the CCS representation stores the non-zero entries in the column-major order. It maintains an array *row_index* to store the row numbers of non-zero entries and an array *column_pointer* to store the number of non-zero values encountered before the current column of the original matrix A.

If the number of non-zero entries in the matrices is M, then both the CRS and the CCS representations require space proportional to M. The CRS and the CCS representations of the following matrix A over GF(2) are shown in Figure 2 and Figure 2, respectively. We assume that array indexing starts from 0.

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

value_array	1	1	1	1	1	1	1
column_index	0	3	1	3	3	4	4

row_pointer	0	2	4	4	6	7

Figure 1: CCS representation of the matrix A.

value_array	1	1	1	1	1	1	1
row_index	0	1	0	1	3	3	4

column_pointer	0	1	2	2	5	7

Figure 2: CRS representation of the matrix A.

For systems over GF(2) (as our case is), the only non-zero entry is 1, so the *value_array* list is not explicitly needed in both the representations.

The Wiedemann method requires a square system, whereas those available from GNFSM sieves are not square. We therefore work with the square matrix A^tA, and a matrix-vector product $(A^tA)v$ involves two matrix-vector products $v' = Av$ and A^tv'. This is the reason why we store the given matrix A in both representations.

3.2 Parallelization using a Controlled Load-balancing Technique

The first step of the Wiedemann method involves many matrix-vector multiplications. We take a random vector v of dimension $N \times 1$, and keep on pre-multiplying v by the input matrix A. Iteratively it generates the sequence A^iv for $i = 0, 1, \cdots, 2N-1$. In the i-th iteration, A^iv is computed by multiplying the matrix A with the vector $A^{i-1}v$ generated in the previous iteration. In the third step, a random vector b of dimension $N \times 1$ is taken, and repeatedly pre-multiplied by the input matrix A for $i = 0, 1, \cdots, N-1$. The number of iterations in the third step is half of that in the first step.

It is evident that the iterations behave very sequentially, that is, the output $A^{i-1}v$ of one iteration should be fully ready for feeding to the next one for computing $A^iv = A \times (A^{i-1}v)$. The only way to optimize an iteration of the method is to parallelize the basic arithmetic operations like sparse matrix-vector multiplications. Such methods are not suitable for massive multi-core parallelization. Moreover, it is essential to ensure that every core (or processor) shares more or less the same amount of computational load. Otherwise, the synchronization step at the end of every iteration may keep many nodes waiting, thereby wasting significant (parallel) running time. To sum up, we face two challenges. First, the parallelization is rather fine-grained, and second, benefits of parallelization are lost in absence of effective load balancing.

A third problem associated with matrices available from GNFSM sieves is that the sparse matrix entries are not non-zero with a uniform probability. Instead, such matrices have some distinct distributions of non-zero entries. In order to work around these problems, an effective load-balancing or loop-scheduling strategy is needed to compute sparse matrix-vector multiplications. The load-balancing constructs of OpenMP (OpenMP, 2016) are exploited together with additional strategies derived from our mathematical analysis. During the sparse matrix-vector multiplication $mul1 : v' = A^iv$, we use the CRS representation of the matrix A. Since the rows follow the same statistical distribution, we evenly distributed them among all the available P processors or cores. Initially, no scheduling constructs are used. For the second sparse matrix-vector multiplication $(mul2 : A^tv')$ (where $v' = Av$) in each Wiedemann iteration, we use the CCS representation of A (which is the CRS representation of the transpose A^t). We start with no load-balancing or loop-scheduling strategies. As already mentioned, the columns of A are not identically distributed in terms of counts of non-zero entries. We therefore need to devise custom-made load-balancing strategies alongside the constructs provided by OpenMP. One such strategy is called *controlled dynamic load balancing*. Here, the matrix is initially analyzed to extract information like the maximum and the minimum numbers of non-zero entries in a column, the indices of the denser columns, the indices of the columns with all zero entries, and so on. Based on these information, we implement the concept of *sharing and stealing* which work as follows. The available cores are divided into smaller groups, and the range of column indices are also divided. This groupings are based on manual calculations based on column-distribution data. The subgroups are assigned various chunks or ranges of the columns, and then we use dynamic scheduling to process the chunks of columns. The subgroup sizes and the column ranges are ex-

perimentally optimized by trying several possibilities. In the *controlled static load balancing* scheme, the static construct (circular scheduling) is used in conjunction with the pre-analyzed grouping of the cores and chunking of columns.

3.2.1 Distributed Implementation of the Wiedemann Method

The implementations reported in the literature use multi-core (more specifically, shared-memory) architecture. In addition to single-node implementations, we also carry out multi-node implementations. We need to handle inter-node communications efficiently during the synchronization after every iteration of the Wiedemann loop.

We investigate two ways of distributing the computational load to multiple nodes. First, we note that the Wiedemann method falls under the category of Las Vegas algorithms that may report failure, and we need multiple solutions in the null space of the given matrix with the hope that at least one of these leads to a non-trivial factoring of the input integer. So we can let every node run the same algorithm with a different random vector v. Here, the parallel running time remains the same as that of a single execution of a single-node multi-core implementation. It only helps in getting multiple non-zero solutions, thereby increasing the probability of successful factoring. In this approach, the communication overhead is essentially absent, as every node runs independently of one another. However, the degree of parallelization is restricted by the number of independent runs required to obtain a successful factoring (with high probability). More precisely, each random vector can factor with a probability of $1/2$, so only a few independent trials are needed for achieving high success probability. Having a larger number of nodes than this requirement cannot be meaningfully exploited using this idea.

A better option is to run the matrix-vector multiplications in the distributed setting. We need to distribute the rows judiciously to the collaborating nodes. For the matrix A, this can be done by equally dividing the rows. However, during the multiplication by the transpose A^t, we have an uneven distribution of non-zero entries in different columns of A. A careful pre-planning is necessary before taking distribution decisions. More processors need to be used for handling the denser columns than those handling the sparser columns. We use the same trial-and-optimize strategy as explained in connection with our controlled load-balancing policy for single-node implementations. This distribution idea is not arbitrarily scalable, because each Wiedemann iteration requires the full vectors v and v' for synchronization. These vectors need to be communicated to all the participating nodes.

3.3 Implementational Details of the Block-Wiedemann Method

This section explains our implementation and practical optimization details for the block-Wiedemann method. We start with a single-node multi-core implementation, and then port it to a distributed implementation. As before, we represent the matrices in the CRS and CCS formats. A new load-balancing technique is developed and reported for the block-Wiedemann method.

3.3.1 Saving Operations

First suggested by (Thomé, 2002), many operations are repeated in the block Wiedemann method. Some of the results can be saved to be reused when needed. The saved computations particularly boost the performance of BW3. In the steps BW1 and BW3, the products of the input sparse matrix A and the already computed blocks A^kV are required twice. These results are saved during step BW1 for use in BW3. This reduces the number of matrix-block computations considerably in BW3.

3.3.2 Choice of Parameters

To start with, we need to choose the block size $m \times n$. The number of useful coefficients in the sequence A_L decreases with m and n, and the complexity of the block operations increases with m and n. After many experimental trials, we fix these parameters as $m = 512$ and $n = 512$. In the step BW2, two additional parameters s and d need to be defined. We notice that fixing $s = 1$ as it is suggested ($s = \frac{m}{n}$ in (Thomé, 2002) or s = 1 in (Anand and II, 2007)) does not provide a high probability of success. The assumption that the coefficient of degree zero in the sequence A_L is of full rank also often appears to be wrong. Therefore, we decide to find at each execution an appropriate value of s, that is, the one for which the columns of the matrices $A_L[0], \ldots, A_L[s-1]$ form a basis of $GF(2)^m$. Moreover, we fix $d = \left\lceil \frac{N}{m} \right\rceil$.

3.3.3 An Adaptive Load-balancing Technique

A scheduling approach for a multi-node cluster proposed by (Bhattacherjee and Das, 2010) computes the time for each unit operation. A runtime adaptive load-balancing algorithm proposed by (Lee and Eigenmann, 2008) uses the cumulative time per core. Every

core loads the entire sparse representation of the matrix under this approach. Following this approach, we extend the controlled load-balanced scheduling, and develop an adaptive load-balancing algorithm for distributing the load across multiple nodes. In this technique, we try to normalize the execution capacities of the cores by ensuring that the average execution time per core remains the same. Let N be the numbers of rows/columns that need to be processed, and M the number of available cores. Initially, $\frac{N}{M}$ numbers of rows/columns are distributed to each of the M cores. The matrix-vector (or block vector) product operation is executed, and the execution time for each row/column is stored in an array/list. The average execution time is calculated as $\frac{\text{total execution time}}{M}$, where the total execution time is the sum of the execution times of all rows/columns. Finally, the rows/columns are redistributed to the cores in following manner.

- Compute the sum of the execution times for all rows/columns in the range 0 to $N-1$.
- Assign the row/column range i to j to the k-th core such that the total execution time for the range i to j is less than the average execution time.
- Record the starting and ending indices of rows/columns for each core.

4 EXPERIMENTAL RESULTS

Our computations are carried out in Intel Xeon E5-2683v4 16C/32T Linux machines. Each node has 32 cores clocked at 2.1 GHz, and there are eight worker nodes, each having 128 GB RAM. These nodes are connected via an InfiniBand switch. The gcc compiler version 9.2.0 is used, OpenMP version 4.5 is used for thread-level parallelism, and OpenMPI 3.1 is used for the distributed implementations. Programs are compiled with the –O2 optimization flag. We have considered two matrices A_1 and A_2 of sizes 3698651×2934621 and 2100000×1500000, respectively. The speedup figures achieved by the single-node implementations are listed in Table 1. Here, DS and SS refer to dynamic and static scheduling without any explicit load-balancing policy, whereas CDS and CSS refer to dynamic and static scheduling with our controlled load-balancing policy. Since the rows of A are statistically identical, we do not apply our controlled load-balancing strategy for the multiplication *mul*1 (computation of Av). This is applied only for the multiplication *mul*2 (computation of Av'). Static scheduling (without load balancing) performs very poorly for *mul*2 and is not listed in the table.

Table 1: Speed-up achieved by several single-node multi-core implementations.

Number of cores	*mul*1		*mul*2		
	DS	SS	DS	CDS	CSS
A_1					
4	1.64	1.07	1.67	1.66	0.96
8	2.84	1.79	2.72	3.00	1.18
16	4.52	2.41	4.08	3.81	1.55
32	4.83	4.65	5.06	6.13	3.86
A_2					
4	2.21	1.24	2.38	2.81	2.11
8	3.64	1.71	3.50	5.29	2.65
16	5.46	3.48	6.78	10.85	4.43
32	6.56	6.30	7.48	19.72	8.34

The effectiveness of our load-balancing strategy is evident from the table. In general, the strategy performs the best in tandem with dynamic scheduling.

A comparison with the reported implementation of (Dumas and Villard, 2002) is shown in Table 2. For meaningful comparisons, we use a 423360×423360 matrix with 23 non-zero entries per row, similar to as reported in (Dumas and Villard, 2002). Our timing figures pertain to the CDS strategy on 4 and 32 cores, whereas Dumas et al. use only 4 cores. The times are in hours.

Table 2: Comparison of load-balanced parallel Wiedemann method.

Our time (4 cores)	Our time (32 cores)	Time by Dumas et al.
23.96	3.92	34.00

Table 3 compares the running times and speedup achieved by distributed implementations with blocking communications (DB) and non-blocking communications (DNB) over our single-node implementation (SN). 32 cores are used in each node. The distributed implementations use eight identical nodes (with 32 cores per node). In all these experiments, controlled load-balancing with dynamic scheduling is used. All times in the table are in seconds.

The table indicates that the blocking communication mode gives very little boost to the distributed implementation. On the other hand, we are able to get some decent speedup with the non-blocking communication mode.

For the block-Wiedemann method, we use a matrix A_3 of size 6699191×6699181 with an average of 63 non-zero entries per row, a maximum of 101 non-zero entries, and a minimum of 52 non-zero entries. Table 4 demonstrates the benefits of using the adaptive load-balancing (ALB) scheme over the controlled load-balancing (CLB) scheme. The times are in seconds. The ALB scheme applies only to the steps BW1 and BW3. For the step BW2, we only use dynamic scheduling. All the distributed implementations run

Table 3: Comparison of single-node and distributed implementations.

*mul*1					*mul*2				
SN	DB		DNB		SN	DB		DNB	
Time	Time	Speedup	Time	Speedup	Time	Time	Speedup	Time	Speedup
A_1									
0.43	0.32	1.34	0.13	3.30	0.48	0.27	1.77	0.11	4.36
A_2									
0.25	0.21	1.19	0.11	2.27	0.11	0.13	0.84	0.07	1.57

Table 4: Distributed running times of block Wiedemann with controlled and adaptive load balancing.

CLB				ALB				
BW1	BW2	BW3	Total	BW1	BW2	BW3	Total	Speed-up
54510	27975	14849	83974	33581	27975	11777	73333	1.14

on eight nodes, each utilizing 32 cores. The non-blocking communication mode of MPI is used in all the implementations reported in the table.

A comparison with the times reported in (Cavallar et al., 2000) and (Chen et al., 2008) is made in Table 5. Although both these papers use the block-Lanczos implementation, these target solving a linear system of dimensions similar to A_3, generated from the sieving stage of RSA-512 factorization. The times reported are in hours.

Table 5: Comparison with (Cavallar et al., 2000) and (Chen et al., 2008).

CLB Strategy	ALB Strategy	Cavallar et al.	Chen et al.
23.33	20.37	224	37.51

5 CONCLUSIONS

This paper deals with the parallel and distributed implementations of the large sparse linear system solver called Wiedemann's method. The focus is on solving systems over GF(2) available from the sieving stage of the general number field sieve method for factoring RSA moduli. In this paper, the Wiedemann method is implemented in three settings: sequential (single-core), parallel (single-node multi-core), and distributed (multi-node multi-core). Effective load-balancing ideas are proposed for the parallel and distributed implementations. The block Wiedemann method is also implemented, parallelized, and distributed. An adaptive load-balancing strategy is designed for the block Wiedemann implementations. Experimental results and speed-up figures are reported extensively to illustrate the effectiveness of our optimization steps.

ACKNOWLEDGEMENTS

This work is funded by Ministry of Electronics and Information Technology, India. Project: Cryptanalysis of cryptographic ciphers with an emphasis on AES and RSA (CER).

REFERENCES

Anand, C. and II, S. (2007). Factoring of large numbers using number field sieve-the matrix step.

Bai, S., Gaudry, P., Kruppa, A., Thomé, E., and Zimmermann, P. (2016). Factorisation of rsa-220 with cado-nfs.

Bai, S., Thomé, E., and Zimmermann, P. (2012). Factorisation of rsa-704 with cado-nfs. *Cryptology ePrint Archive*.

Barbulescu, R., Bouvier, C., Detrey, J., Gaudry, P., Jeljeli, H., Thomé, E., Videau, M., and Zimmermann, P. (2014). Discrete logarithm in gf (2 809) with ffs. In *International Workshop on Public Key Cryptography*, pages 221–238. Springer.

Beckermann, B. and Labahn, G. (1994). A uniform approach for the fast computation of matrix-type padé approximants. *SIAM Journal on Matrix Analysis and Applications*, 15(3):804–823.

Bender, E. A. and Canfield, E. R. (1999). An approximate probabilistic model for structured gaussian elimination. *Journal of Algorithms*, 31(2):271–290.

Berlekamp, E. (1968). Binary bch codes for correcting multiple errors. *Algebraic Coding Theory*.

Bhateja, A. and Kannan, V. (2017). Cache optimized solution for sparse linear system over large order finite field. In *International Conference on Mathematics and Computing*, pages 84–95. Springer.

Bhattacherjee, S. and Das, A. (2010). Parallelization of the lanczos algorithm on multi-core platforms. In *International Conference on Distributed Computing and Networking*, pages 231–241. Springer.

Case, M. (2003). A beginner's guide to the general number field sieve. *Oregon State University, ECE575 Data Security and Cryptography Project*.

Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W., Montgomery, P. L., Murphy, B., Te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., et al. (2000). Factorization of a 512-bit rsa modulus. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–18. Springer.

Chen, J.-M., Yu, S.-I., Ou-Yang, Y., Wang, P.-H., Lin, C.-H., Huang, P.-Y., Yang, B.-Y., and Laih, C.-S. (2008). Improved factoring of rsa modulus. In *Proceedings of the 25th Workshop on Combinatorial Mathematics and Computation Theory*. Citeseer.

Coppersmith, D. (1994). Solving homogeneous linear equations over gf(2) via block wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350.

Cowie, J., Dodson, B., Elkenbracht-Huizing, R., Lenstra, A., Montgomery, P., and Zayer, J. (1996). A world wide number field sieve factoring record: On to 512 bits. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 382–394. Springer.

Dumas, J.-G. and Villard, G. (2002). Computing the rank of large sparse matrices over finite fields. In *Computer Algebra in Scientific Computing (CASC) 2002*, pages 47–62.

Kaltofen, E. (1995). Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806.

Kaltofen, E. and Lobo, A. (1999). Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, 24(3):331–348.

Kaltofen, E. and Saunders, B. D. (1991). On wiedemann's method of solving sparse linear systems. In *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, pages 29–38. Springer.

Kaltofen, E. and Yuhasz, G. (2013). On the matrix berlekamp-massey algorithm. *ACM Transactions on Algorithms (TALG)*, 9(4):1–24.

Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., et al. (2010). Factorization of a 768-bit rsa modulus. In *Annual Cryptology Conference*, pages 333–350. Springer.

Krylov, A. N. (1931). On the numerical solution of the equation by which the frequency of small oscillations is determined in technical problems. *Izv. Akad. Nauk SSSR Ser. Fiz.-Mat*, 4:491–539.

Lanczos, C. (1952). Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Standards*, 49(1):33–53.

Lee, S. and Eigenmann, R. (2008). Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 195–204.

Lenstra, A. K., Lenstra, H. W., Manasse, M. S., and Pollard, J. M. (1993). The number field sieve. In *The development of the number field sieve*, pages 11–42. Springer.

Lin, C.-Y., Chung, Y.-C., and Liu, J.-S. (2003). Efficient data compression methods for multidimensional sparse array operations based on the ekmr scheme. *IEEE Transactions on Computers*, 52(12):1640–1646.

Massey, J. (1969). Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15(1):122–127.

Montgomery, P., Cavallar, S., and Te Riele, H. (1997). A new world record for the special number field sieve factoring method. *CWI Quaterly*, 10(2):105–107.

Montgomery, P. L. (1995). A block lanczos algorithm for finding dependencies over gf (2). In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 106–120. Springer.

OpenMP, A. (2016). The openmp api specification for parallel programming, openmp arb.

Penninga, O. (1998). Finding column depedencies in sparse matrices over f_2 by block wiedemann. *Modelling, Analysis and Simulation [MAS]*.

Pomerance, C. and Smith, J. W. (1992). Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experimental Mathematics*, 1(2):89–94.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Thomé, E. (2002). Subquadratic computation of vector generating polynomials and improvement of the block wiedemann algorithm. *Journal of symbolic computation*, 33(5):757–775.

Villard, G. (1997). Further analysis of coppersmith's block wiedemann algorithm for the solution of sparse linear systems. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 32–39.

Wiedemann, D. (1986). Solving sparse linear equations over finite fields. *IEEE transactions on information theory*, 32(1):54–62.

Yang, L. T., Huang, G., Feng, J., and Xu, L. (2017). Parallel gnfs algorithm integrated with parallel block wiedemann algorithm for rsa security in cloud computing. *Information Sciences*, 387:254–265.