Parallelization of the Wiedemann Large Sparse System Solver over Large Prime Fields

For the partial fulfilment of the degree of Master of Technology

Pratyay Mukherjee (09CS6001)

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Under the guidance of

Dr. Abhijit Das

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur,

WB 721302, India.

Certificate

This is to certify that the thesis entitled Parallelization of the Wiedemann large sparse system solver over large prime fields, submitted by Pratyay Mukherjee, roll no: 09CS6001, in the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India, for the award of the degree of Master Of Technology, is a record of an original research work carried out by him under my supervision and guidance. The thesis fulfills all requirements as per the regulations of this institute. Neither this report nor any part of it has been submitted for any degree or academic award elsewhere.

Abhijit Das

This is a draft version of the original thesis with some minor changes

Abstract

The discrete logarithm problem over finite fields serves as the source of security for several cryptographic primitives. The fastest known algorithms for solving the discrete logarithm problem require solutions of large sparse linear systems over large prime fields, and employ iterative solvers for this purpose. The published results on this topic are mainly focused on systems over binary fields, that is, systems coming from integer-factoring algorithms. Solving systems over large prime fields has not yet received much research attention. In this thesis, our main goal is to efficiently implement the Wiedemann method to solve large sparse linear systems of equations over large prime fields. The second phase of the Wiedemann method (computation of the minimal polynomial of a linear sequence) offers several choices including the Berlekamp-Massey and the Levinson-Durbin algorithms. Assessing the relative performance of the above two variants of the second phase is another important goal of this work. We first detail our optimized sequential implementation of the Wiedemann method. Subsequently, we deal with shared-memory parallel implementations of the Wiedemann method using a small number of cores. We have been able to achieve a speedup of about four using eight cores. Our experiments also suggest that the Levinson-Durbin algorithm in the second stage is more suitable to parallelization than the Berlekamp-Massey algorithm.

Contents

List of Figures

Chapter 1

Introduction

The Discrete Logarithm Problem (DLP) over finite fields serves as the basis of several cryptographic primitives. For example, the Diffie-Hellman key-agreement protocol [10], the ElGamal public-key crypto-system [11] and the Digital Signature Algorithm (DSA) [20] rely on the difficulty of solving the DLP, for their security to remain intact. Similarly, the Integer Factorization Problem(IFP) serves as the basis of various important public-key cryptosystems e.g. RSA [31]. Since most of the public-key crypto-systems are based on these two computationally hard problems, these problems attract significant research attention. Because, solving these problems in a feasible time eventually leads to breaking those crypto-systems. Currently, the best algorithms known to solve these problems are certain sub-exponential-time sieving algorithms. If x is the positive integer to be factored, then the time complexity of a sub-exponential algorithm can be expressed as,

$$L(x, w, c) = exp[(c + o(1)(lnx)^{\omega}(ln(lnx))^{1-\omega}] \tag{1.1}$$

for some positive constant c and for some real number ω $(0 < \omega < 1)$. Evidently, smaller the value of ω, the expression gets closer to the polynomial. In this context, The Number Field Sieve [22, 14] and The Function Field Sieve [2] are the two fastest known methods and in these methods the value of ω is close to 1/3. Before the proposal of these two algorithms, the Quadratic Sieve, proposed by Carl Pomerance [29], was the best known algorithm for factoring integers. In fact, it is still used for factoring integers of size ≤ 100 digits. For solving the DLP, these sub-exponential algorithms can be easily adapted to prime fields and extension fields of small characteristics. Collectively, these methods are known as Index Calculus Methods.

The algorithms to solve the DLP consist of two phases. The first phase is sieving which produces linear systems of equations of the form $A\mathbf{x} = \mathbf{b}$. These systems posses certain characteristics like sparsity. Again, for integer factoring, the equations are modulo 2 whereas, in case of Discrete Logarithm, these equations are modulo p where p is a very large prime (may be as big as 512 bits).

Essentially, the fastest known algorithms for solving the DLP require the solution of large sparse linear systems over finite fields. As the size of the system of equations increases, standard Gaussian elimination becomes impractical($O(n^3)$). However, some alternative iterative methods prove to be computationally more efficient than Gaussian elimination, particularly for large and sparse linear systems.

Efficient implementations of these iterative system solvers are quite challenging as the linear-algebra phase often turns out to be the practical bottleneck in the context of solving the DLP. The Lanczos method [6] and the Wiedemann method [32] are two iterative system solvers that outperform Gaussian elimination for Large Sparse Linear Systems. Now, our current objective is to efficiently implement the Wiedemann method to solve a large sparse linear system in large prime field. While talking of efficient implementations, the notion of parallelization comes consequently. But in reality, the sieving part turns out to be massively parallelizable, whereas, the linear system-solving part provides some resistance to massive parallelization. The main focus of the thesis is efficient implementation of the Wiedemann's system solver first in a sequential platform and then in a multi-core scenario, at least for a limited number of cores.

1.1 Motivation

The Gaussian Elimination Method is the most popular algorithm which is used to solve linear systems. But when the size of the system becomes larger, it becomes infeasible to implement this algorithm. Nonetheless, for comparatively smaller systems it can be used successfully. But it has been discovered that, some other iterative methods are computationally more attractive than Gaussian elimination for large and sparse linear systems, because the fillin of Gaussian elimination is too large to handle for these systems. The Lanczos [6] and the Wiedemann [32] methods are two iterative methods that can solve large sparse linear systems much more efficiently than the Gaussian elimination. In order to speed up the entire algorithm to solve the DLP, efficient implementation and consequently parallelization of the linear algebra step is an absolute necessity. However, these iterative methods are inherently sequential, mainly because no iteration in these algorithms can start before the previous iteration completes. Therefore, the only option left is to parallelize each iteration of the algorithm. But in that scenario, it essentially implies parallelization at a relatively fine level. So, scalability or load-balancing issues (in terms of the number of concurrent processors) become important.

Most of the research works done in this field are either abstract in nature [33] or focused towards systems over $GF(2)$ [12, 16, 8]. As mentioned earlier in this chapter, these kind of systems are generally generated from the sieving when solving Integer Factorization. There are obviously a lot of advantages of the systems in $GF(2)$ compare to those in $GF(p)$. The storing of values can be done in a single bit for the former case but it requires multi-precision storage in the later. Again, the arithmetics involved in $GF(p)$ must be multi-precision arithmetics which is too much expensive. Whereas, the arithmetics involved in $GF(2)$ can be done in bitwise operations which is obviously much efficient than the earlier one. These difficulties set the challenges to implement $GF(p)$ case separately.

The Lanczos Algorithm mentioned earlier, is the most important algorithm as a competing algorithm to the Wiedemann's. The Lanczos Algorithm has been implemented [5] in the exactly same platform, which is being used here to implement the later one. And the parallel implementation of the former has shown a speed up of 6.57 (using different library, 4.51 using the library used in our case) over eight cores. Success in implementing Lanczos motivates us to explore several implementation issues of the Wiedemann Algorithm.

So, in this project our initial objective was to implement Wiedemann Algorithm sequentially with its two variants (to be discussed in the next chapter). After naive sequential implementation we tried to optimize the implementations to increase the performances considering several practical situations with a matrix having dimension $2.2mil \times 1.6mil$ and possessing special characteristics similar to those coming from sieving step while solving DLP. Then we tried to parallelize the implementations over an 8-core platform. We compared the performances of the two variants in both sequential and multi-core scenario.

1.2 Contribution

The primary object of this thesis is to exploit the implementation issues of the Wiedemann Algorithm and to find out how this algorithm responds in parallel scenario. Since this algorithm uses another algorithm as a subroutine (to be clarified later) to find minimal polynomial [32], we have the flexibility to use different algorithms to serve that purpose. In this thesis we will explore two different algorithms (viz. Berlekamp-Massey [4] and Levinson-Durbin [19]) from implementation perspective. Comparing these algorithms with respect to different parameters both in sequential and in multi-core scenario is one of the main theme of the thesis.

First, the Wiedemann Algorithm has been implemented naively using both the sub-algorithms. Of course, we are dealing with some special systems which are very large and sparse. So, considering these issues the implementations were modified with special data structures. Also, corresponding changes in the subroutines were made. Experiments were done with random sparse systems in prime fields of different size.

Since, our ultimate goal was to solve the Discrete Log Problem, the properties of the systems of equations generated after the sieving part were taken into consideration. A matrix having similar characteristics to systems generated by the sieving method having the dimension of $2.2m \times 1.6m$ was taken for the experiment and the sequential implementation has been optimized using various technique.

Finally, both the implementations has been tried to parallelize over a standard 8-core platform, and the results obtained from two different subroutines were compared both in parallel and in sequential scenario. Also, the important characteristics of these two different algorithms were explored.

1.3 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 provides a detailed literature survey and the theoretical background. In this chapter, all the competing algorithms are described briefly along with a detailed theoretical foundation for the Wiedemann Algorithm. In chapter 3, the sequential implementation is described thoroughly. We mention the difficulties we faced and how they were overcome. Also, a comparative study of two different implementations is reported with two different subroutines and the results are analyzed thoroughly. In chapter 4, the parallelization of both the sequential implementations is described in details and similar to the sequential case both the implementations are compared and analyzed. Finally, chapter 5 provides the possible future works related and the conclusion, at which we arrives through the project.

Chapter 2

Related Work and Background

2.1 Introduction

When dealing with large systems of equations, standard Gaussian elimination turns out to be too costly in terms of time and space. So, some alternative approach was a necessity, especially for the sparse systems. Because, using the sparsity property might reduce the complexity of time and space. In this chapter, we are going to discuss one of the special algorithm viz. Wiedemann Algorithm which iteratively solves very large systems of equations in feasible time and space. The Lanczos and the Conjugate Gradient method [30] are two competing iterative solvers for large sparse linear systems over prime fields. In this chapter, we provide a brief description of these iterative solvers, collectively known as Krylov space methods. An adaptation of the standard Gaussian Elimination to reduce the size of the linear systems is also described.

2.2 Gaussian Elimination

Gaussian Elimination is a standard method for solving a linear systems of equations. This method also finds use in computing the rank of a matrix and in finding the inverse of an invertible square matrix. Gaussian Elimination consists of two phases. In the first phase, it reduces a given system to a triangular or echelon form. In the second phase, it uses back-substitution to find the solution to the systems of equations provided. This method has a time-complexity of $O(n^3)$, where n is the dimension of the input matrix. Further, as the size of the prime modulus increases, the system size increases proportionately. During the first phase of Gaussian Elimination, a large percentage of matrix elements get modified. As a result, even if we start with a sparse matrix, we eventually end up getting a dense one after the first phase. So the space requirement of Gaussian Elimination becomes proportional to $O(n^3)$. Henceforth, Standard Gaussian Elimination can not be a suitable candidate for solving large sparse linear systems. On the other hand, the alternative iterative methods converge in $\leq n$ iterations, n being the number of variables in the concerned system. Each iteration can be completed in $O(nk)$ times, where k denotes the maximum number of non-zero entries in a row of a sparse matrix. While solving the DLP, we surely are only interested in the special matrices which are generated by sieving. For these matrices, we have $k = O(\log n)$, which leads to the fact that in total, the iterative methods run in $O(n^2 \log n)$ time. Also, the space requirement becomes $O(n \log n)$. A partial application of the Gaussian Elimination method, known as the Structured Gaussian Elimination (first proposed by Odlyzko [25], and later improved by himself and Lamacchia [21]) can often prove to be quite effective even in the context of sparse matrices. It was designed with an intention to reduce the size of the system while simultaneously preserving its sparsity. The method starts by declaring some columns (those with the largest number of non-zero elements) as heavy and eliminates rows and columns without increasing the sparsity of the remaining light columns. It is theoretically not very clear about the extent to which Structured Gaussian Elimination needs to be performed before the matrix is passed to a sparse system solver. Because, if the elimination is carried out beyond a certain limit, it results in an unwanted problem called the avalanche. The limit up to which Structured Gaussian Elimination should be continued depends on a number of parameters and is determined only empirically. Structured Gaussian Elimination is usually expected to reduce the number of columns to one-third. We will not study this algorithm in detail in this thesis.

2.3 Introduction to Iterative Solvers

Iterative methods like the Lanczos and the Conjugate Gradient methods attempt to reach a solution by computing successive approximations to the solution starting with an initial guess. A somewhat different iterative method was proposed by Wiedemann in 1986. Apart from the vital problem of breakdown associated with some of these methods, these methods are inefficient when used over $GF(2)$. The reason is that, these methods over $GF(2)$ require manipulations with single bits, whereas most computers handle chunks of 32 or 64 bits simultaneously. Further, if the bit vectors are compressed for efficiency of storage, the single bits need to be uncompressed before use. These algorithms should be modified to work with blocks of vectors instead of single vectors. Block versions of Lanczos [7] and Wiedemann [8] were first proposed by Coppersmith. He modified these two algorithms so as to run with 32 vectors at a time. After that, Montgomery [24] came up with a new version of the block Lanczos algorithm. This method runs faster and is less complicated than the Coppersmith's one. That is why, Montgomery's block Lanczos algorithm is used extensively for Integer Factorization.

After the works of Coppersmith and Montgomery no fundamentally new sparse solvers are proposed in literature. Instead, the focus gradually shifted from efficient algorithms to efficient implementations. The efforts started with Montgomery implementing his own version of block Lanczos [24] and parallely Lobo [23] implementing the block Wiedemann method of Coppersmith. In 1998, Penninga [27] implemented a variant of the block Wiedemann method suggested to him by Villard, which is based on algorithm by Beckermann and Labahn [3]. Penninga also compares his results with Lobo's [23] and Montgomery's [24] and concludes that the block Lanczos method by Montgomery is much faster than both variants of the block Wiedemann method. In 2001, Yang and Brent [33] proposed an improved version of the Lanczos method capable of running on parallel architectures. In addition, they present a theoretical model of computation and communication phases, which provides a quantitative analysis of the parallel performance of the algorithm. They selected Lanczos over $GF(2)$. This marked the beginning of an era of implementing sparse system solvers on parallel platforms. The implementations reported in [28, 18, 16, 12] are focused towards systems over $GF(2)$. In 2004, Dan Page [26] implemented Lanczos algorithm over $GF(p)$ in a distributed platform and suggested some useful technique to handle the large integers. In the following section of this chapter we will describe thoroughly the working principle of Wiedemann Algorithm along with its two different variants mentioned earlier and also we will provide a brief description of Lanczos and the Conjugate Gradient methods.

2.4 The Wiedemann Algorithm

2.4.1 Introduction

Douglas Wiedemann's [32] landmark approach to solve sparse linear systems over finite fields provides the symbolic counterpart to non-combinatorial numerical methods for solving sparse linear systems, such as the Lanczos method [6]. The problem is to solve a sparse linear system, when the individual entries lie in a generic field, and the only operations possible are field arithmetic; the solution is to be exact. This situation is particularly possible while one is operating in a finite field only. Wiedemann presents a randomized Las Vegas algorithm for solving a sparse linear system over a finite field. Classically, he made a link to the Berlekamp-Massey problem [4] from coding theory while using randomization at the same time. On input of an $n \times n$ matrix A given by a so-called black box, which is a program that can multiply the matrix by a vector, and of a vector b, the algorithm finds, with high probability in case the system is solvable, a solution vector x with Ax = b. It is assumed that the field has sufficiently many elements (say no less than $50n^2log(n)$). Otherwise one goes for a finite algebraic extension. So, we are assuming here, the field is $GF(p)$, where p is a large prime (it may be as big as

512 bits). This assumption will ensure the sufficiency of elements in that field.

2.4.2 The Algorithm

We are given an $m \times n$ matrix B over prime field $GF(p)$ with $m > n$ to represent the linear system:

$$B\mathrm{x} \equiv \mathrm{u}(mod p) \tag{2.1}$$

We assume that the equations are consistent and u is in the column space of B. The computation of DLP demands the solution i.e. x to be unique, that is the matrix B must be of full column rank. Here this requirement is ensured with a high probability since the system is overdetermined [19]. Now, the Wiedemann algorithm is classically applicable to systems of the following form:

$$A\mathrm{x} = \mathrm{b} \tag{2.2}$$

where A is a square matrix of dimension $n \times n$. In order to adapt this algorithm to the case of finite fields, we transform Equation(2.1) to Equation(2.2) by letting

$$A = B^t B, \tag{2.3}$$

$$\mathrm{b} = B^t \mathrm{u} \tag{2.4}$$

where B^t denotes the transpose of B.

Now we can apply Wiedemann's method on the matrix A where A is an $n \times n$ square matrix. One important point is that, unlike Lanczos here A need not to be symmetric or positive definite. This is an advantage of the concerned algorithm over Lanczos' one. Now, the characteristic equation of A is

$$\chi_A(x) = det(xI - A), \tag{2.5}$$

where I is the $n \times n$ identity matrix. Calley-Hamilton theorem states that A satisfies $\chi_A(x)$, that is , $\chi_A(A) = 0$. Now, the set of all polynomials in $K[x]$ satisfied by A forms an ideal of $K[x]$. The monic generator of this ideal, that is, the monic non-zero polynomial of the smallest degree, which A satisfies, is called the minimal polynomial of A and denoted as $\mu_A(x)$. Clearly, $\mu_A(x)|\chi_A(x)$ in $K[x]$. Wiedemann Algorithm starts by probabilistically determining $\mu_A(x)$. Let

$$\mu_A(x) = x^d - c_{d-1}x^{d-1} - c_{d-2}x^{d-2} - ..... - c_1x - c_0 \in K[x] \tag{2.6}$$

with $d = deg(\mu_A(x)) \leq n$. Since $\mu_A(A) = 0$, for any $n \times 1$ non-zero vector v and for any integer $k \geq d$, we have :

$$A^k\text{v} - c_{d-1}A^{k-1}\text{v} - .....c_1A^{k-d+1}\text{v} - c_0A^{k-d}\text{v} = 0. \tag{2.7}$$

Let v_k be the element of A^kv at some particular position. The sequence v_k for $k \geq 0$, satisfies the recurrence relation:

$$v_k = c_{d-1}v_{k-1} + c_{d-2}v_{k-2} + ... + c_1v_1 + c_0v_0 \tag{2.8}$$

for all $k \geq d$. Using the Berlekamp-Massey algorithm classically or by some other appropriate algorithm we compute the polynomial $C(x)$ whose degree is $d' \leq d$. But then $x^{d'}C(1/x)|\mu_A(x) \in K[x]$. After trying several such sequences (corresponding to different position in v), we obtain various polynomials $x^{d'}C(1/x)$ whose L.C.M. is expected to give the minimal polynomial $\mu_A(x)$.

In order that the Berlekamp-Massey algorithm works correctly in each case, we take the obvious upper bound n for d, and supply $2n-1$ vector elements $v_0, v_1, ..., v_{2n-1}$. This means that we need to compute the $2n$ matrix-vector products A^iv for $i = 0, 1, 2, ...2n-1.$. Since A is a sparse matrix with $\tilde{O(n)}$ non-zero entries, the determination of $\mu_A(x)$ can be computed in $\tilde{O(n^2)}$ time.(It is easy to argue that Berlekamp-Massey algorithm performs a total of only $\tilde{O(n^2)}$ basic field arithmetic in a field K)

For computing a solution of Ax = b, we use $\mu_A(x)$ as follows. Putting $k = d$ and v = b in Eqn (2.7) gives:

$$A(A^{d-1}\text{b} - c_{d-1}A^{d-2}\text{b} - c_{d-2}A^{d-3}\text{b} - ...... - c_1A\text{b}) = c_0\text{b}, \tag{2.9}$$

that is, if $c_0 \neq 0$,it becomes:

$$\text{x} = c_0^{-1}(A^{d-1}\text{b} - c_{d-1}A^{d-2}\text{b} - c_{d-2}A^{d-3}\text{b} - ...... - c_1A\text{b}) \tag{2.10}$$

which is a solution of $A\mathrm{x} = \mathrm{b}$.The basic time-consuming task here is the computation of the $d \leq n$ matrix-vector products $A^i\mathrm{b}$ for $i = 0, 1, 2, ...d - 1$ which can be computed in $\tilde{O(}n^2)$ time.

2.4.3 Finding Minimal Polynomial

Wiedemann's method for solving a linear system is based upon linear recurrent sequences. Let $a_0, a_1, a_2, ...$ be an infinite sequences of the elements which belong to a field K and the first d terms are supplied as initial conditions. So, for all $k \geq d$ we have:

$$a_k = c_{d-1}a_{k-1} + c_{d-2}a_{k-2} + ... + c_0 a_{k-d} \tag{2.11}$$

for some constant elements $c_0, c_1, ..., c_{d-1} \in K$.

Wiedemann used the Berlekamp-Massey algorithm classically to find the minimal polynomial which is essentially finding the co-efficients $c_0, c_1, ...c_{d-1}$. This algorithm uses basic polynomial arithmetics. Here we describe briefly, how that algorithm works.

Consider the generating function of the sequence $a_0, a_1, a_2, ...$:

$$\begin{aligned} G(x) &= a_0 + a_1 x + a_2 x^2 + ... + a_{d-1}x^{d-1} + a_d x^d + a_{d+1}x^{d+1} + ... && (2.12) \\ &= (a_0 + a_1 x + a_2 x^2 + ... + a_{d-1}x^{d-1} + a_d x^d) + \sum_{k \geq d} a_k x^k. && (2.13) \\ &= (a_0 + a_1 x + a_2 x^2 + ... + a_{d-1}x^{d-1} + a_d x^d) + \\ &\sum_{k \geq d} (c_{d-1}a_{k-1} + c_{d-2}a_{k-2} + ... + c_0 a_{k-d})x^k && (2.14) \end{aligned}$$

that is,

$$C(x)G(x) = R(x) \tag{2.15}$$

where $R(x)$ is a polynomial of degree $\leq d-1$, and

$$C(x) = 1 - c_{d-1}x - c_{d-2}x^2 - .... - c_0x^d. \tag{2.16}$$

In order to compute $c_0, c_1, ...c_{d-1}$, it suffices to compute $C(x)$. Clearly, we can use extended GCD algorithm as follows. Let us assume,

$$A(x) = a_0 + a_1x + a_2x^2 + ... + a_{2d-1}x^{2d-1}. \tag{2.17}$$

Then, equation(2.15) can be re-written as:

$$C(x)A(x) + B(x)x^{2d} = R(x) \tag{2.18}$$

for some polynomial $B(x)$. These observations lead to the algorithm we are going to describe next for computing the co-efficients $c_0, c_1, ...c_{d-1}$ in equation(2.11), given the first $2d$ terms $a_0, a_1, a_2, , ..., a_{2d-1}$ in the sequence. The extended GCD computations maintains an invariance of the form $B_1(x)x^{2d} + C_1(x)A(x) = R_1(x)$. Since the multiplier $B_1(x)$ is not needed, it is not explicitly computed.

The Iteration:(Berlekamp-Massey)

```
Let $R_0(x) = x^{2d}$ and $R_1(x) = a_0 + a_1x + a_2x^2 + ..... + a_{2d-1}x^{2d-1}$
Initialize $C_0(x) = 0$ and $C_1(x) = 1$.
While($deg(R_1(x)) \geq d$){
        Let $Q(x) = R_0(x)\text{quot}R_1(x)$ and $R(x) = R_0(x)\text{rem}R_1(x)$
        Update $C(x) = C_0(x) - Q(x)C_1(x)$.
        Prepare for the next iteration:
          $R_0(x) = R_1(x)$,$R_1(x) = R(x)$,$C_0(x) = C_1(x)$ and $C_1(x) = C(x)$.
}
```

Divide $C_1(x)$ by its constant term.
Recover the coefficients $c_0, c_1, ..., c_{d-1}$ from $C_1(x)$

Now, while analyzing Coppersmith's block Wiedemann Algorithm [17], Kaltofen proposed an algorithm which can be used to compute the minimal polynomial. This algorithm can be used as a subroutine within the Wiedemann algorithm instead of the earlier mentioned Berlekamp-Massey's algorithm because essentially both of them are doing the same thing in same running time $O(d^2)$.(if d iterations are involved)

As stated earlier, Wiedemann's method is based upon linear recurrent sequences. Now, we recall the equation(2.11) and re-write the equation as:

$$\begin{bmatrix} a_{k-1} & a_{k-2} & \dots & a_{k-d} \end{bmatrix} \begin{bmatrix} c_{d-1} \\ c_{d-2} \\ \vdots \\ c_1 \\ c_0 \end{bmatrix} = a_k \tag{2.19}$$

Using this relation for $k = d, d+1, d+2, ..., 2d-1$, we write:

$$\begin{bmatrix} a_{d-1} & a_{d-2} & \dots & a_1 & a_0 \\ a_d & a_{d-1} & \dots & a_2 & a_1 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{2d-2} & a_{2d-3} & \dots & a_d & a_{d-1} \end{bmatrix} \begin{bmatrix} c_{d-1} \\ c_{d-2} \\ \vdots \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_d \\ a_{d+1} \\ \vdots \\ a_{2d-2} \\ a_{2d-1} \end{bmatrix} \tag{2.20}$$

Clearly, since the first $2d$ terms that is $a_0, a_1, a_2, \dots, a_{2d-1}$ are known to us, solving the above system will yield the solution as the values of the co-efficients $c_0, c_1, ..., c_{d-1}$. Now, to solve our system $A\mathbf{x} = \mathbf{b}$, we need to solve the above system. But, one very important thing is to notice the special structure the above matrix has. Every row is shifted one position towards the right with a introduction of a new element in its leftmost position. This kind of matrix is called Toeplitz matrix. A Toeplitz matrix system can be solved as efficiently as $O(d^2)$ running time. In his modified scheme, Kaltofen used this technique to find the minimal polynomial of the matrix given as an input to the Wiedemann's method. We are going to explain the iterative scheme mainly proposed by Levinson and modified by Durbin and Zehar.

We have a system as follows:

$$T\mathbf{c} = \mathbf{w}. \tag{2.21}$$

It is an $n \times n$ linear system with the i, j-th entry of T is given by $t_{i-j} \in K$(a function of $i - j$). Now, let us assume that the $i \times i$ sub-matrix sitting at the top left corner of T is denoted by $T^{(i)}$, that is,

$$T^{(i)} = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & \dots & t_{-i+1} \\ t_1 & t_0 & t_{-1} & \dots & t_{-i+2} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ t_{i-1} & t_{i-2} & t_{i-3} & \dots & t_0 \end{bmatrix} \tag{2.22}$$

Clearly $T = T^{(n)}$. Similarly, let $\mathbf{w}^{(\mathbf{i})}$ denote the $i \times 1$ vector obtained by the top i elements of $\mathbf{w}$ (we have $\mathbf{w} = \mathbf{w}^{(n)}$). We iteratively solve the system:

$$T^{(i)}\mathbf{c}^{(i)} = \mathbf{w}^{(i)}. \tag{2.23}$$

for $i = 1, 2, 3..., n$. We also keep on computing and using two auxiliary vectors y^i and z^i satisfying :

$$T^{(i)}\mathbf{y}^{(i)} = \begin{bmatrix} \epsilon^{(i)} \\ \mathbf{0}_{i-1} \end{bmatrix} \qquad \textit{and} \qquad T^{(i)}\mathbf{z}^{(i)} = \begin{bmatrix} \mathbf{0}_{i-1} \\ \epsilon^{(i)} \end{bmatrix} \tag{2.24}$$

for a suitable choice of the scalar $\epsilon^{(i)}$ to be specified later. Here the superscript $^{(i)}$ is used to denote the i-the iteration. In an actual implementation, remembering the values form only the previous iteration will suffice.

For $i = 1$, we have $t_0 c_1 = w_1$ which gives $\mathbf{c}^{(1)} = (t_0^{-1} w_1)$, provided that $t_0 \neq 0$. We also take $\mathbf{y}^{(1)} = \mathbf{z}^{(1)} = (1)$ which implies that $\epsilon^{(1)} = t_0$.

Iteration(Levinson-Durbin's algorithm)

Suppose that, we have already solved $T^{(i)}\mathbf{c}^{(i)} = \mathbf{w}^{(i)}$, and from this we want to find the solution for $T^{(i+1)}\mathbf{c}^{(i+1)} = \mathbf{w}^{(i+1)}$. At this stage, the vectors $\mathbf{y}^{(i)}$ and $\mathbf{z}^{(i)}$, and the scalar $\epsilon^{(i)}$ is known. We can write:

$$T^{(i+1)} = \begin{bmatrix} & & & t_{-i} \\ & T^{(i)} & & \vdots \\ t_i & t_{i-1} & \dots & t_0 \end{bmatrix} = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{-i} \\ \vdots & T^{(i)} & & \\ t_i & & & \end{bmatrix} \tag{2.25}$$

This implies that the following equalities hold:

$$T^{(i+1)} \begin{bmatrix} \mathbf{y}^i \\ 0 \end{bmatrix} = \begin{bmatrix} \epsilon^{(i)} \\ \mathbf{0}_{i-1} \\ -\epsilon^{(i)}\xi^{(i+1)} \end{bmatrix} \quad \textit{and} \quad T^{(i+1)} \begin{bmatrix} 0 \\ \mathbf{z}^i \end{bmatrix} = \begin{bmatrix} -\epsilon^{(i)}\zeta^{(i+1)} \\ \mathbf{0}_{i-1} \\ \epsilon^{(i)} \end{bmatrix} \tag{2.26}$$

where,

$$\xi^{(i+1)} = -\frac{1}{\epsilon^{(i)}} \begin{bmatrix} t_i & t_{i-1} & \dots & t_1 \end{bmatrix} \mathbf{y}^{(i)}. \tag{2.27}$$

and,

$$\zeta^{(i+1)} = -\frac{1}{\epsilon^{(i)}} \begin{bmatrix} t_{-1} & t_{-2} & \dots & t_{-i} \end{bmatrix} \mathbf{z}^{(i)}. \tag{2.28}$$

we compute $\mathbf{y}^{i+1}$ and $\mathbf{z}^{i+1}$ as linear combinations of $\begin{bmatrix} \mathbf{y}^i \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ \mathbf{z}^i \end{bmatrix}$ as :

$$\mathbf{y}^{(i+1)} = \begin{bmatrix} \mathbf{y}^{(i)} \\ 0 \end{bmatrix} + \xi^{(i+1)} \begin{bmatrix} 0 \\ (\mathbf{z})^i \end{bmatrix} \tag{2.29}$$

and,

$$\mathbf{z}^{(i+1)} = \begin{bmatrix} 0 \\ \mathbf{z}^{(i)} \end{bmatrix} + \zeta^{(i+1)} \begin{bmatrix} \mathbf{y}^{(i)} \\ 0 \end{bmatrix} \tag{2.30}$$

and this requires us to take

$$\epsilon^{(i+1)} = \epsilon^{(i)}(1 - \xi^{(i+1)}\zeta^{(i+1)}). \tag{2.31}$$

Finally we update the solution $c^{(i)}$ to $c^{(i+1)}$ by noting that :

$$T^{(i+1)} \begin{bmatrix} \mathbf{c}^{(i)} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{w}^{(i)} \\ \eta^{(i+1)} \end{bmatrix} = \mathbf{w}^{(i+1)} + (\frac{\eta^{(i+1)} - w_{i+1}}{\epsilon^{(i+1)}})T^{(i+1)}\mathbf{z}^{(i+1)}, \tag{2.32}$$

where,

$$\eta^{(i+1)} = \begin{bmatrix} t_i & t_{i-1} & \dots & t_1 \end{bmatrix} \mathbf{c}^{(i)}. \tag{2.33}$$

that is,

$$\mathbf{c}^{(i+1)} = \begin{bmatrix} \mathbf{c}^{(i)} \\ 0 \end{bmatrix} + (\frac{w_{i+1} - \eta^{(i+1)}}{\epsilon^{(i+1)}})\mathbf{z}^{(i+1)} \tag{2.34}$$

This solution vector **c** contains the co-efficients from the equation 2.9. After recovering the co-efficients, the solution to the original system is extracted similarly.

2.5 Lanczos Algorithnm

Like Wiedemann the Lanczos Algorithm is classically applicable to the linear system

$$A\mathbf{x} = \mathbf{b} \tag{2.35}$$

but unlike Wiedemann, here A must be symmetric and positive definite.The standard Lanczos algorithm solves equation (2.35) by starting with $\mathbf{w}_0 = \mathbf{b}$ and iterating as follows:

$$\mathbf{w}_i = A\mathbf{w}_{i-1} - \sum_{j=0}^{i-1} c_{ij}\mathbf{w}_j \qquad (i > 0) \tag{2.36}$$

where

$$c_{ij} = \frac{\mathbf{w}_j^t A^2 \mathbf{w}_{i-1}}{\mathbf{w}_j^t A \mathbf{w}_j} \tag{2.37}$$

until $\mathbf{w}_i = 0$ is obtained. It can be proved that,

$$\mathbf{w}_j^t A\mathbf{w}_i = 0 \qquad (i \neq j) \tag{2.38}$$

Therefore, the vectors $\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_i$ are linearly dependent, viz. $\sum_{j=0}^{i} a_j\mathbf{w}_j = 0$ where, $a_i \neq 0$. So, from equation (2.38) left-multiplying both sides yield $a_i\mathbf{w}_i^t A\mathbf{w}_i = 0$. Since $a_i \neq 0$, we have $\mathbf{w}_i = 0$ by positive definiteness of A. Now, let us assume k is the first value of i for which $\mathbf{w}_i = 0$. We define

$$\mathbf{x} = \sum_{j=0}^{k-1} \frac{\mathbf{w}_j^t\mathbf{b}}{\mathbf{w}_j^t A\mathbf{w}_j}\mathbf{w}_j \tag{2.39}$$

then, from equation(2.36) and equation(2.37) we get,

$$A\mathbf{x} - \mathbf{b} \in \{A\mathbf{w}_0, A\mathbf{w}_1, \ldots, A\mathbf{w}_{k-1}, \mathbf{b}\} \subseteq \{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_k\}. \tag{2.40}$$

By construction, we have $\mathbf{w}_j^t A\mathbf{x} = \mathbf{w}_j^t\mathbf{b}$ for $0 \leq j \leq k-1$. Therefore, we get, $(A\mathbf{x} - \mathbf{b})^t(A\mathbf{x} - \mathbf{b}) = 0$, that is $A\mathbf{x} = \mathbf{b}$. It is to be observed that the equations (2.36) and (2.37) require adding suitable multiples of earlier $\mathbf{w}_j$ for computing $\mathbf{w}_i$. The terms vanish when $j < i - 2$, because,

$$\begin{aligned}
\mathbf{w}_j^t A^2\mathbf{w}_{i-1} &= (A\mathbf{w}_j)^t(A\mathbf{w}_{i-1}) \\
&= (\mathbf{w}_{j+1} + \sum_{j=0}^{k} c_{j+1,k}\mathbf{w}_k)^t(A\mathbf{w}_{i-1}) \\
&= 0 \qquad (j < i-2)
\end{aligned} \tag{2.41}$$

Therefore, by equation (2.38), equation (2.36) simplifies to,

$$\mathbf{w}_{i+1} = A\mathbf{w}_i - c_{i+1,i}\mathbf{w}_i - c_{i+1,i-1}\mathbf{w}_{i-1} \qquad (i \geq 1) \tag{2.42}$$

where the co-efficients $c_{i+1,i}$ and $c_{i+1,i-1}$ can be computed as follows

$$c_{i+1,i} = \frac{(A\mathbf{w}_i)^t(A\mathbf{w}_i)}{\mathbf{w}_i^t(A\mathbf{w}_i)} \tag{2.43}$$

and,

$$c_{i+1,i-1} = \frac{(A\mathbf{w}_{i-1})^t(A\mathbf{w}_i)}{\mathbf{w}_{i-1}^t(A\mathbf{w}_{i-1})} \tag{2.44}$$

2.6 Conjugate Gradient Algorithm

The Conjugate Gradient method was proposed by Hestenes and Stiefel[30]. This method finds a solution to the equation $A\mathbf{x} = \mathbf{b}$ by updating $\mathbf{x}$ along conjugate direction vectors $\mathbf{p}_i$, where A is a real, symmetric and positive-definite matrix. In this method, $\mathbf{x}_0$ is initialized with some random vector and also $\mathbf{p}_0$ and $\mathbf{r}_0$ are set as follows,

$$\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{y} - A\mathbf{x}_0$$

Then this method iterates as follows,

$$\begin{aligned}
\mathbf{s}_i &= A\mathbf{p}_i \\
a_i &= \frac{\mathbf{r}_i^t \mathbf{r}_i}{\mathbf{p}_i^t \mathbf{s}_i} \\
\mathbf{x}_{i+1} &= \mathbf{x}_i + a_i \mathbf{p}_i \\
\mathbf{r}_{i+1} &= \mathbf{r}_i - a_i \mathbf{s}_i \\
\mathbf{b}_i &= \frac{\mathbf{r}_{i+1}^t \mathbf{r}_{i+1}}{\mathbf{r}_i^t \mathbf{r}_i} \\
\mathbf{p}_{i+1} &= \mathbf{r}_{i+1} + b_i \mathbf{p}_i
\end{aligned}$$

In these equations, $\mathbf{r}_i$ is the residual vector, $\mathbf{p}_i$ serves as the conjugate vector, and $\mathbf{s}_i$ represents the matrix-scaled direction vector. As in the Lanczos method, the solution vector $\mathbf{x}$ is obtained when the iteration terminates. Termination occurs when $\mathbf{r}_i = 0$. This requires $\leq n$ iterations, where n is the number of variables in the given system of equations.

2.7 Comparison among iterative methods

All the iterative solvers described in this section runs in $\tilde{O(n^2)}$ time. It is the constant term associated which distinguishes them. However, from above discussion, if we observe them from a high level, it can be noted that the number of iterations in the Wiedemann algorithm is almost double than that of Lanczos algorithm. Also, the space requirement is almost two times for the former case in comparison with the later one. But, the combinatorial approach and flexibility which Wiedemann's algorithm offers, draw our attention to explore it from an implementation point of view. In the next few chapters we will discuss the implementation issues of the algorithm along with the experimental results both in sequential and parallel scenario.

Chapter 3

Sequential Implementation

3.1 Introduction

In this chapter we discuss the sequential implementation of Wiedemann Algorihm over large prime field $GF(p)$ with its two variants which use Berlekamp-Massey and Levinson-Durbin methods to find the minimal polynomial. We mention difficulties arouse while implementing the algorithm and the technique by which those were fixed. A few modification techniques which were used in the implementation of Lanczos Algorithm[5] has been implemented here too. But, the main theme of this chapter is to explore several characteristics of two variants of the Wiedemann Algorithm from implementation perspective. In the conclusion of this chapter we provide a comparative study of two different variants from the experimental results with analysis.

3.2 Storing the matrix

As a beginning step, the algorithm was implemented naively. That is, the matrix was represented in normal two-dimensional array and vectors or polynomials were in one-dimensional array. But, this kind of storage was found to be very inefficient when the matrix is very sparse and has a large dimension(e.g. $10^6 \times 10^6$). So, we needed some data structure which would be advantageous in case of sparse matrix.

There are several techniques to store the sparse matrix efficiently. One of them is to store the matrix as an array of triplets where each such triplet contains $(data, rowno, colno)$. Another technique is to represent the matrix in Compressed Row Format. In this technique three arrays are maintained. One of them contains data which are nothing but the non-zero entries and these are stored row-wise. That is, in this format, storing is done by moving in a row first. Whenever, the last non-zero element of a row has been entered, the next row is processed. Another array is maintained to store the column number corresponding to the non-zero entries. Another very important array is also maintained, which is used to store the number of non-zero elements in a row cumulatively. So, the last array has smaller size compare to the first two as its size is only equal to the number of row whereas size of the first two arrays is equal to the number of elements. We used this scheme to implement Wiedemann algorithm. Because, we noticed that, in this scheme the number of row elements is stored cumulatively which is an important information while doing arithmetics over matrices. Also, storing in this scheme requires lesser space as the size of the third array is smaller. There is another format called Compressed Column Format which is very similar to the previous one. The only difference is, while storing in this format we are moving across a column first, storing the row number in the second array and the cumulative number of elements in a column in the final array. Here we are providing an example how a matrix is stored in CRS format and CCS format in figures 3.1, 3.2 and 3.3.

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

Figure 3.1: The Example Matrix B

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

Figure 3.2: CRS format of the matrix B

val	10	3	3	9	7	8	4	8	8	7	9	-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

Figure 3.3: CCS format of the matrix B(same as CRS of B^t)

Also, we have considered the fact that the entries of the vectors in this algorithm belong to a generic field $GF(p)$ characterized by a prime p which may be as large as 512 bits. So, the normal integer arithmetic would not support that. That is why we used multiple-precision arithmetic using a library called GNU/MP [13].

3.3 Steps for implementation

The Wiedemann Algorithm consists of three major steps. It executes three different iterations in those steps. After completing one step only the next step can be executed. This idea is quite different from its competing algorithms e.g. Lanczos or Conjugate Gradient method, in which everything takes place within one iteration. Now, we will briefly describe those implementation steps.

3.3.1 Matrix-vector Multiplication

As a first step, we have to take a random vector $\mathbf{v}$ and pre-multiply it by the input matrix A (recall from equation(2.3) $A = B^tB$) to generate $A^i\mathbf{v}$ iteratively for $i = 0, 1, ..., 2n-1$. To avoid matrix multiplications we compute $A^i\mathbf{v}$ in the i-th iteration by multiplying the matrix A and vector $A^{i-1}\mathbf{v}$(which has already been computed in the previous iteration). We maintain an array of vectors called A_V to store these products. $A\_V[i]$ denotes the product $A^i\mathbf{v}$. Since Wiedemann algorithm converges with a very high probability, storing only first few (say 5) elements of $A\_V[i]$ instead of the whole vector $A\_V[i]$ should suffice. This optimization certainly saves a lot of space especially when the dimension of the matrix is of the order of 10^6. This step involves the costliest matrix-vector multiplication. Also, here the range of the loop variable i is twice the dimension of matrix A where in case of Lanczos it is equal to the dimension.

3.3.2 Finding the Minimal Polynomial

In the second step, Wiedemann algorithm finds the minimal polynomial by another algorithm (two different algorithms has been used to do this task as described earlier in section 2.3.3). This algorithm takes the first elements of the products computed in the previous step and tries to find out the minimal polynomial. That is, it takes the first elements of vectors $A\_V[i]$ (where $i = 0, 1, \ldots, 2n - 1$) in order as the coefficients of the input polynomial. Since, Wiedemann is a probabilistic Las Vegas algorithm [19], it may be the case that the algorithm fails to find the minimal polynomial in the first chance. If it is not found, it tries with the next elements of the vectors $A\_V[i]$ (where $i = 0, 1, \ldots, 2n - 1$). Since, it is a randomized algorithm there is a probability that it is not found even after all the elements are given as input. Then we go back to the first step and try with another random vector. But, for very large system the chance of not founding is too small to consider. So, we can assume that it succeeds within the first few trials.

3.3.3 Evaluating the solution vector

In the final step, after computing the minimal polynomial we evaluate the solution vector by earlier mentioned formula (equation 2.10). This computation again involves the costly matrix-vector multiplication accompanied by scalar-vector multiplication, vector addition and subtraction. If the total execution time of the algorithm is considered, this step is expected to be less costlier compare to the first step as the range of the loop variable i is equal to the dimension of matrix A (viz. $i = 0, 1, \ldots, n - 1$) that is half of the range in the first step, though it is as costly as the first step considering a single iteration.

3.4 Special structure of matrix

Since the input matrix we are considering here comes from sieving step to solve DLP, a few important characteristics can be observed in this type of matrices what we must take into account. We have to consider the special structures of the input matrix. Classically the Wiedemann Algorithm takes a square matrix as input. Since, the matrix B, which is the output of the sieving step, is overdetermined, we have to pre-multiply B with B^t to make it a square matrix A (as explained in section 2.3.2). But, the problem with A is that, it is not as sparse as B and so we lose the advantage of sparsity here. Further, the time consumed by the matrix multiplication (B^tB) is very large and therefore we always want to avoid this costly operation here. To fix this, we rather keep the working matrix as (B^tB) instead of computing the product A explicitly, and store B^t and B separately. It is easy to get CRS (explained in the section 3.2) of B^t as it is nothing but the CCS of B.

First, we assume that the matrix B has dimension $m \times n$. Now, the n columns of B can be divided into two separate blocks considering the non-zero elements and density of a column. The first t columns are corresponding to the small primes in the factor base (detail is available in the discussion of Linear Sieve Algorithm in [9]). The remaining columns of B correspond to the $2M + 1$ variables coming from the sieving interval. For $1 \leq i \leq t$, the i-th column empirically contains about $\frac{m}{p_i}$ 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 coincide resulting in a single non-zero entry of -2, but probability of this event is insignificant and henceforth it is ignored in our subsequent discussions. Each of the last $2M + 1$ columns contains $2m/n$ non-zero entries on an average. For $m \leq 2n$, this value is ≤ 4.

From our observation it has been noted that almost three-fourth of the non-zero entries of B are $+1$. Also, most of the remaining non-zero entries are found to be -1. While multiplying a vector by B or B^t ,the matrix entries ± 1 draw special attention. For example, if we consider a typical sum of the form $\sum_r b_r v_r$ where b_r denotes the non-zero entries of B and v_r are the entries of a vector, the addition of the product $b_r v_r$ can be replaced by the addition of v_r if $b_r = 1$ and by the subtraction of v_r if $b_r = -1$. Finally as mentioned in [26] , an excellent strategy to speed up the matrix-vector multiplication is to perform the modulo prime reduction after the entire expression $\sum_r b_r v_r$ is evaluated. Since b_r are single-precision integers, therefore, although, v_r are general elements of $GF(p)$ and that is why are multi-precision integers, the word size of $\sum_r b_r v_r$ is only slightly larger than that of the prime p, even when there are many terms in the sum (like during multiplication by the first row of B^t, number of non-zero terms is half of the size of the row).

Again, it must be mentioned here, although the characteristics of the systems coming from only the linear sieve has been considered here, these hold almost identically for the systems generated from other sieving. If we observe the matrices generated from Cubic Sieve [9], there will be exactly three -1 instead of two in the last $2M + 1$ columns in each row. In case of Number Field Sieve [22, 14], the matrices generated do not contain blocks of -1. 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. However, it is interesting that, in any case, most of the non-zero entries of the matrices are ± 1. Therefore, the strategy we described in the previous paragraph will be fruitful in those cases too.

3.5 Experiments and Results

In section 3.3 we have described the three steps in high-level for sequential implementation. Now, since we had estimated that, for the matrix having dimension of $2.2milion \times 1.6milion$, it will take more than one month for total computation, we decided to measure time for one iteration in each step. Again, we have mentioned in section 3.3 that the next step can not start until the previous step is complete. Therefore, we generated some random data to supply as input to the next step. Mainly, the first step and the third step are dominated by costly matrix-vector multiplication and these steps are common for both the implementations. So, we report the time taken for this operation solely. But in the second step there are two different implementations, so we report the total time taken in one iteration of second step separately. The primary objective of this chapter is to analyze the results obtained from these two sub-algorithms and compare their performances varying several parameters.

3.5.1 Modifications

First of all our target was to achieve the correctness. After achieving correctness for comparatively smaller matrices we executed the experiment for a standard matrix of dimension $2.2m \times 1.6m$. While handling this massive amount data, we faced a lot of problem in managing memory and time simultaneously. It is important to mention here that in the following modifications only the execution time of matrix-vector multiplication has been measured. Since it is the costliest operation and consumes most of the total execution time, it is a good candidate for measuring time after implementing various modifications.

Representing B^t in CRS

At first, we represented B in CRS form and B^t in CCS form. But, then the matrix-vector multiplication $B^t(B\mathbf{v})$ takes enormous amount of time which was making the computation almost infeasible for large systems. Because, this operation normally takes each row of matrix at a time whereas CCS stores the matrix column wise. So, we made a modification that is we stored both B and B^t in CRS form by generating the transpose of B. Although the transpose operation is costly one, but since it works outside any iteration, it does not affect the total running time significantly.

Reducing Cache Miss

After the modification mentioned above, the execution time of one iteration of the matrix-vector multiplication was taking approximately 22 second. But, there were a lot of scope for improvement. Another very important improvement was to merge the first two arrays viz. value and col_ind of matrix and to represent them in one single structure. This improved the running time a lot. The execution time of one iteration of the matrix-vector multiplication came down to 12.5 sec. Actually, by representing within one structure we reduce the probability of cache miss. Basically, our observations suggest that whenever we are accessing any value of the matrix, immediately, the column index of that value is being accessed. That is why, this kind of representation helps a lot in our implementation.

Using faster functions

In the matrix-vector multiplication, initially we used addition/subtraction and multiplication routines separately in GNU/MP. Later we replaced them with single add_-mul/sub_mul functions. These functions are lower level functions and work faster than the previous functions. Also, this technique reduces number of functions in the loop. This modification brings down the execution time of the first step from 12.5 sec to 10.82 sec

3.5.2 Results

After all these modifications, we did a few experiments taking a matrix of dimension $2.2mil \times 1.6mil$. First of all, we concentrated into the first part which involves mainly a matrix-vector multiplication. We have taken four different primes of different size which are given below. We report the execution time of one iteration of the first step with respect to arithmetics in different fields characterized by these four primes. The results are shown in fig. 3.4.

Prime$_0$

660122737328154591756994364686791621421435841
(45 digit)

Prime$_1$

66012273732815459175699436468679162142143584200000000000000000000019
(68 digit)

Prime$_2$

12897917477709474701209401271092740912709174170947494717047104714714
9170947094770941704704974714971709417094701470821
(117 digit)

Prime$_3$

925842416918996157077437639542304415337233709452846213414260851731447
0534489119835183448328128741818048424454918349563314641020251485996940
365595422378597
(154 digit)

After this experiment we look into the second step. Now, this step is different in two different implementations. Since, the comparison of these two implementations is the main theme of the thesis, we thoroughly investigate the algorithms and then implement both of them. Due to unavailability of the whole data from the first step (as mentioned earlier that completion of one step would take more than a month), we generate the input data randomly and use the same data for both Berlekamp-Maseey and Levinson-Durbin algorithm. We execute the algorithms for different iterations. The results are graphically provided in figure 3.5. Here, only the 154-digit prime field ($Prime_3$) is considered.

3.5.3 Observations and Analysis

Now, from fig. 3.5 we can see that, while the Levinson-Durbin (abbr. LVD) implementation takes more time in the later iterations than the earlier iterations, the Berlekamp-Massey (abbr. BKM) implementation takes almost equal time in every iteration.

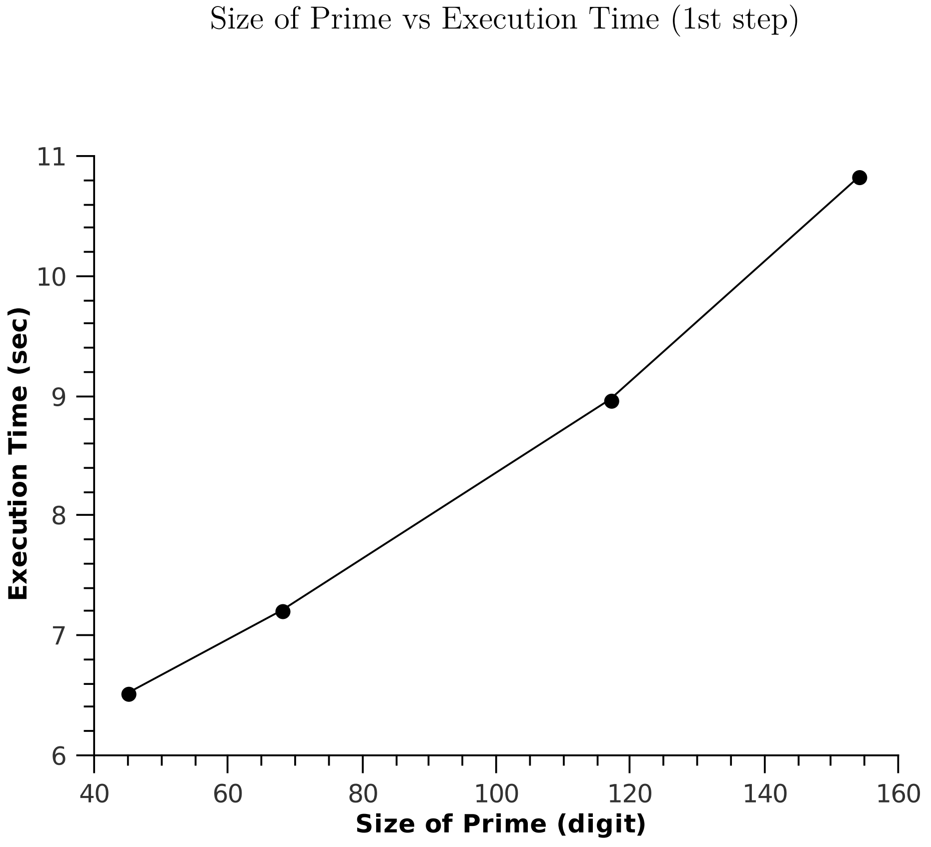

Figure 3.4: Size of Prime vs Execution Time in First Step

In case of LVD, the execution time in an iteration varies from 3.7 sec to 10.1 sec, whereas in case of BKM, the variation is from 8.4 sec to 8.9 sec, which is pretty small. It is interesting to notice that, in the earlier iterations, LVD performs much better than BKM whereas it is opposite in the later iterations. Now, considering our experiments, where five iterations has been chosen in almost equal distance, if we calculate the average execution time in an iteration, LVD (average 6.75 sec) seems to perform better than BKM (average 8.67 sec).

Analyzing the behavior, discussed above, requires a thorough observation of the algorithms. If we go back to section 2.3.3, we can clearly see that, BKM essentially deals with polynomials having degree of maximum $2n$. In each iteration, it computes four polynomials viz. R_0, R_1, C_0, C_1. Initially, the degree of R_0 is $2n$ and that of R_1 is $2(n-1)$, whereas the degree of C_0 and C_1 are initialized to 0. In each iteration, the degree of the polynomials R_0 and R_1 decreases and that of C_0 and C_1 increases. So, it is clear that in each iteration it has to deal with polynomials of large degree. That justifies the behavior we have seen in figure 3.5.

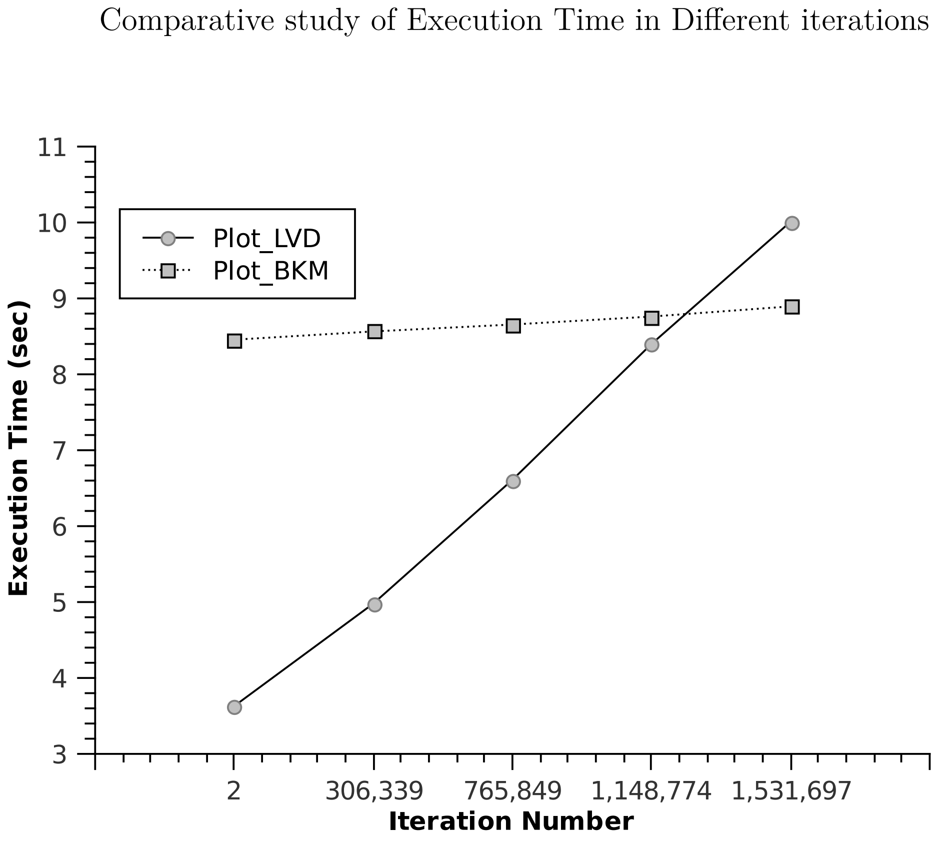

Figure 3.5: Comparative execution Time in Different iterations

Now, if we consider the LVD algorithm, we can clearly see that it deals with vectors. In each iteration it computes three vectors X, Y and Z and scalars $\xi, \zeta, \epsilon, \eta$. The fact is that, the scalars are calculated by the vector arithmetic from the X, Y, Z vectors computed in the previous iteration. And, the most important thing to notice is that, the dimension of those vectors are always equal to the present iteration number. This is the main reason for taking lesser time in the earlier iterations. Because, as in the earlier iterations the arithmetic subroutines deal with smaller vectors, they compute faster resulting in a smaller execution time.

3.6 Conclusion

In this chapter we report the sequential implementations of the two variants of Wiedemann Algorithm and compare their results. From the results and analysis we can conclude that, the LVD outperform BKM in finding the minimal polynomial when average execution time is taken into account. We admit that, there are still scope to improve the implementations. But, since, the improvement will affect both the variants simultaneously, we can speculate that, the conclusion will remain valid after those improvements. Also the analysis is not only experimental. In fact, we have given a valid theoretical reason to justify the behavior, which enforces our conclusion.

Chapter 4

Multi-Core Implementation

4.1 Introduction

From chapter 1 we remember that, our basic intention to solve a large sparse system efficiently is to solve the Discrete Logarithm Problem in a feasible time. So, if we are provided with a parallel architecture, we must exploit the parallelism of the implementation to achieve some considerable speed up over the sequential implementation. However, now-a-days, parallel systems becomes quite affordable. So, efficient implementation can not be completed without parallelization. In this chapter we are going to discuss our effort to parallelize the Wiedemann algorithm. We will explore the difficulties and various techniques to parallelize the implementation and also compare the parallelizability of two variants of this algorithm. Recently, the paradigm of parallel computation has shifted from commodity clusters to multi-core processors. A multi-core machine contains P processing elements which is computationally equivalent to a cluster consisting of P nodes (one CPU per node) with the added advantage of shared memory and at the same time it eliminates the need of message exchange across machines, which results in a significant degradation of performance. In this chapter, we discuss our implementations along with the experimental results for both the variants of Wiedemann sparse system solver on a standard eight-core platform.

4.2 Parallel Implementation

With an intention to parallelize, after thoroughly observing the section 2.3 and 3.3, we notice that, each iteration in every step is dependent upon the previous iteration and further, each step depends on the previous one. So, the only way left is to parallelize the basic arithmetic routines. The basic routines which we tried to parallelize are matrix-vector multiplication (dominant in first and third step and the costliest one), vector-vector multiplication(second step of LVD), vector addition/subtraction(second step of LVD and third step), scalar-vector multiplication(second step of LVD and third step), polynomial addition/subtraction/multiplication (second step of BKM), copy vector/polynomial (second step of both and third step).

4.3 Load Balancing

Parallel architecture is beneficial only when each core (or processor) shares more or less the same amount of computational load. Otherwise, in synchronization step some processors wait a long period for some other to complete and henceforth the benefit of parallel processing is lost. So, if an uneven distribution of load among processors are found, explicit load balancing is necessary. Here, most of the subroutines mentioned in the previous section are inherently parallelizable. That is if we equally distribute the range of loop variable among the processors they are expected to do the same amount of computations. As an example, if we apply this trivial load balancing technique to vector-vector addition operation of two vectors having dimension d, then each of the P processors handles exactly $\frac{d}{P}$ elements. Most importantly, we should keep in mind that, unlike matrices, the vectors and polynomials in this algorithm are dense. So, there is no need of explicit load-balancing for those sub-routines which handle only vectors or polynomials.

4.3.1 Problem with trivial parallelization

The need of explicit load-balancing boils down to only one non-trivial operation that is the matrix-vector multiplication. Now, we have mentioned in section 3.4, that instead of multiplying (B^tB) we keep it as it is and perform two successive matrix-vector multiplication viz. $(B)(v)$ and $B^t(Bv)$. Now, we notice that, the number of elements in each row of B does not deviate much from its average. But, in case of B^t it is not true. As described in section 3.4, the rows of B^t which are nothing but the columns of B are quite unevenly distributed. The first row is almost half-filled whereas the last $2M + 1$ rows contain a very few number of elements. So, the second multiplication is not at all parallelizable trivially. Special technique is needed to fix that. In fact, our experiment (takes 1.34 sec for the first multiplication and 6.25 sec for the second in eight-core where sequential takes 3.73 sec and 7.09 sec respectively) with trivial parallelization supports our speculation.

4.3.2 A non-trivial load-balancing

Now, since the matrix is very sparse and the non-zero entries are scattered throughout the matrix following a pattern, we decided to distribute equal number of non-zero elements to each processor. Using the Open MP library [1] which uses thread-level parallelism, we have implemented the technique and achieved a considerable amount of speed up (3.72 over 8-core). While, implementing this, we used the following technique. We divided the total number of elements by the number of cores i.e. P. Since, we wanted to give equal chunk of non-zero elements to each processor, the range of loop variable for each thread is known to us previously. We precomputed the corresponding row number for the starting value of the loop variable in each thread. While a thread starts working, it needs the row number corresponding to the non-zero entry, and it finds that row number easily from the precomputed table. Again, it is quite possible that, a particular row is computed by two different threads. That case is handled carefully for starting row and ending row of each thread. The results found are depicted in figure 4.1 and figure 4.2.

4.4 Memory Allocation

Although load balancing is the most important part, still there exist a huge scope of modifications. Here, we have implemented a few of them. Since we are handling a very large amount of data, accessing memory is a matter of concern. If the processors or threads have to wait long for accessing memory, the benefit of parallelization is lost. So, other modifications are intended to fix that problem.

4.4.1 Avoiding run-time memory allocation

The memory allocation is done using the standard malloc() command in C. But, as mentioned in [5], we experienced that, malloc() maintains everything in a single heap. Therefore, when a number of threads want to allocate memory at the same time, only one get chance. Other threads have to wait until that one is finished with allocation. This eventually degrades the performance heavily. So, we decided to allocate all the memory beforehand. That is, before the threads come into action, all the necessary memory are allocated.

4.4.2 Using lower-level functions

Initially, the implementation was done with standard GNU/MP [13] integer variable of the type mpz_t. But, the subroutines in GNU/MP for handling mpz_t integers which have prefix mpz_ allocate memory in run-time resulting degradation in performance. So, to avoid this problem, instead of using mpz_t integers, we used multi-precision integers which are handled by mpn_ subroutines. Basically, we used mp_limb_t types of variable which represents limbs rather than multi-precision integers. We used array of this limbs to store a multi-precision integer. These arrays were allocated beforehand. Since the subroutines which handle mp_limb_t variables are very low-level functions having prefix mpn_ (in fact mpz_ functions are written in GNU/MP [13] using these lower-level functions), there is no chance of run-time memory allocation. Implementing with mp_limb_t variables and mpn_ functions fixed the problem of run-time memory allocation and at the same time made the implementation much more difficult. Because, we had to handle the multi-precision integers by each limb and the interface of the low-level functions prefixed with mpn_ are not at all user-friendly.

4.4.3 Avoiding copy-vector

Copy-vector turns out to be a costly operation because it is memory intensive operation and that is why the probability of cache miss is quite high due to the large dimension of vectors (elements of which are multi-precision integer). Therefore it does not respond well in parallel scenario. So, whenever possible, the copy-vector operations are replaced by simple pointer exchanges which executes in almost no time, although, in some cases copy-vector is inevitable.

4.5 Parallelizing the second step

Now, we are going to discuss our effort to parallelize the second step of the Wiedemann Algorithm which essentially computes the minimal polynomial. Since, this step is implemented sequentially with two different algorithms as discussed in chapter 3, also in this chapter we discuss them separately. Once again, if we have a look into both these algorithms (section 2.3), we find that, the candidate for parallelization are the basic routines which has been mentioned in section 4.2. But, the costliest operation i.e. the matrix-vector multiplication is absent in this step. In the following sections, we describe the efforts to parallelize Levinson-Durbin(LVD) first and then we move ahead to Berlekamp-Massey(BKM).

4.5.1 Parallelizability of Levinson-Durbin's algorihm

This algorithm deals with vectors. Actually, it solves a special kind of matrix viz. Toeplitz Matrix system in $O(n^2)$ time. But, essentially in each iteration it computes vectors X, Y, Z and scalars $\xi, \zeta, \epsilon, \eta$ from the vectors and scalars computed in the previous iteration. It involves the basic vector arithmetics viz. vector-vector addition/subtraction, vector-vector multiplication, scalar-vector multiplication mainly. Also we have used two more routines here: copy-vector and nullify-vector. Since the vectors are dense as we have discussed in section 4.2, the trivial technique should work. Therefore we parallelize each subroutine with trivial technique. But, since the operations like vector addition/subtraction are quite memory intensive just like copy-vector, there are a lot of cache misses. So, the speed-up obtained is quite low (<3 in 8-core). Various techniques may be adapted to reduce the cache miss which will certainly increase the speed-up.

Avoiding Critical Section in Vector Multiplication

While parallelizing the vector-vector multiplication, in each iteration we have to atomically update one shared variable because the result will be a scalar which is the sum of all the products computed in each iteration. Now, to implement this, every thread has to access the critical section in each loop. This eventually degrades the speed-up because a number of times, more than one thread try to access the critical section and thus except only one, the other threads have to wait. To remove this critical section we implemented a different technique. Instead of using shared variable, we used a private variable for each thread where the sum computed in a thread accumulates. And, when the thread is finished with its computation we update the shared variable. In this technique, each thread needs to access the critical section only once after it is done with all the multiplications. This technique increases the speed-up of solely this operation from 2.13 to 3.72 which obviously enhances the overall speed-up in this phase.

4.5.2 Parallelizing Berlekemp-Massey Algorithm

While, the LVD algorithm deals with vectors, this one deals with polynomials. Although, there is not much differences in representation between polynomials and vectors, difference lies among these algorithms which has already been discussed in a sequential scenario in section 3.5.3. This algorithm uses basic operations of polynomials like addition, subtraction, copy, nullification etc. But the main difference lies in multiplication which makes it harder to parallelize. Obviously, in case of polynomial multiplication, each coefficient of one polynomial is multiplied with all the coefficients of the other. Whereas in case of vector multiplication, an element of a vector is multiplied only with the element of another vector having the same index.

Polynomial multiplication avoiding Critical section

Since, in naive polynomial multiplication algorithm a nested loop exists, in each iteration of the inner loop, critical section is accessed by the thread. So, in this case, the critical section accessing increases several times compare to the vector multiplication. In the previous section we mentioned that in each iteration critical section is accessed once, whereas in this case it is being accessed a number of times which is exactly equal to the range of inner loop variable (which is nothing but the degree of the smaller polynomial). Also, it is expected that, using critical section in the inner loop would never help to increase the speed-up. So, we tried to avoid critical section by implementing a temporary polynomial which is private to the thread. Each thread stores the output to its temporary private polynomial. So, the number of the temporary polynomial should be equal to the number of thread. After the computation, the entries of these polynomials are summed up and stored in the product polynomial. This is also a costly operation which involves a number (obviously equal to the number of threads) of polynomial addition. And, this number of addition increases with the number of processors which overshadow the benefit of parallelization and as a whole results significant degradation in performance. Therefore, removing critical section using this technique, pays off a considerable cost.

Implementing this technique, as speculated, we obtained a very little speed-up. So, although this technique seems to work for the vector-vector multiplication, it does not work good for polynomial multiplications. Therefore, we rejected this technique.

4.6 Experiments and Results

The experiments were carried out on an Intel® Xeon® E5410 dual-socket quad-core Linux server. These eight processors run at a clock-speed of 2.33 GHz and support 64-bit computations. The machine has 8 GB of main memory and a shared L2 cache of size 24 MB across 8 cores. The parallelism is achieved using free Open-MP[1] version 4.3.2. The multiple precision integer are handled using GNU/MP[13] version 4.3.1.

First, in figure 4.1, we present the execution time in first step with respect to the number of cores. Since two matrix-vector multiplications in this step has different characteristics as discussed in section 4.3, the execution time is shown separately along with the total. The experimentations are done in the parallel scenario described above only varying the number of processors involved. Then we provide the speed up we obtained compare to the sequential implementation when different number of cores are involved. The speed up is depicted graphically in figure 4.2. Here, again we present separate speed up for two multiplications along with the net speed up.

Then we present the experimental results obtained from the second step. Here, first we present the execution time of LVD algorithm varying the number of processors involved. Since, we have seen from the analysis in section 3.5.3, it takes different times in different iterations and the execution time increases with iteration number, so it becomes necessary to show the results obtained from experiments in different iterations. So, we experimented it for three iterations (first one,last one and the middle one) and the results are shown graphically in figure 4.3. Then in figure 4.4 we show the results obtained from the parallel implementation of BKM. As discussed in section 4.5.2, the technique to avoid the critical section did not work. So, we rather left the polynomial multiplication operation un-parallelized and executed our experimentations.

Again, similar to the sequential case (like figure 3.5), we provide the execution time obtained in different iterations for both LVD and BKM in 8-core platform to make a comparative study in figure 4.5, and the corresponding speed-ups are shown in fig.4.6.

4.7 Observations and Analysis

From figure 4.1 and figure 4.2 we get the performance of first-step which is mostly dominated by the matrix-vector multiplication over multi-core platform. As discussed in section 4.3, the first multiplication is easier to handle than the second one when load-balancing is concerned. Also, in case of the first multiplication, the size of vector is much smaller ($1.6mil$) compare to the second (vector size $2.2mil$). It increases the loop overhead which results in greater execution time and lesser speed-up for the second multiplication. So, implementing the load-balancing technique described in section 4.3.2, we get better speed-up i.e. 4.72 for the first-multiplication compare to the second one viz. 3.34 over 8-core.

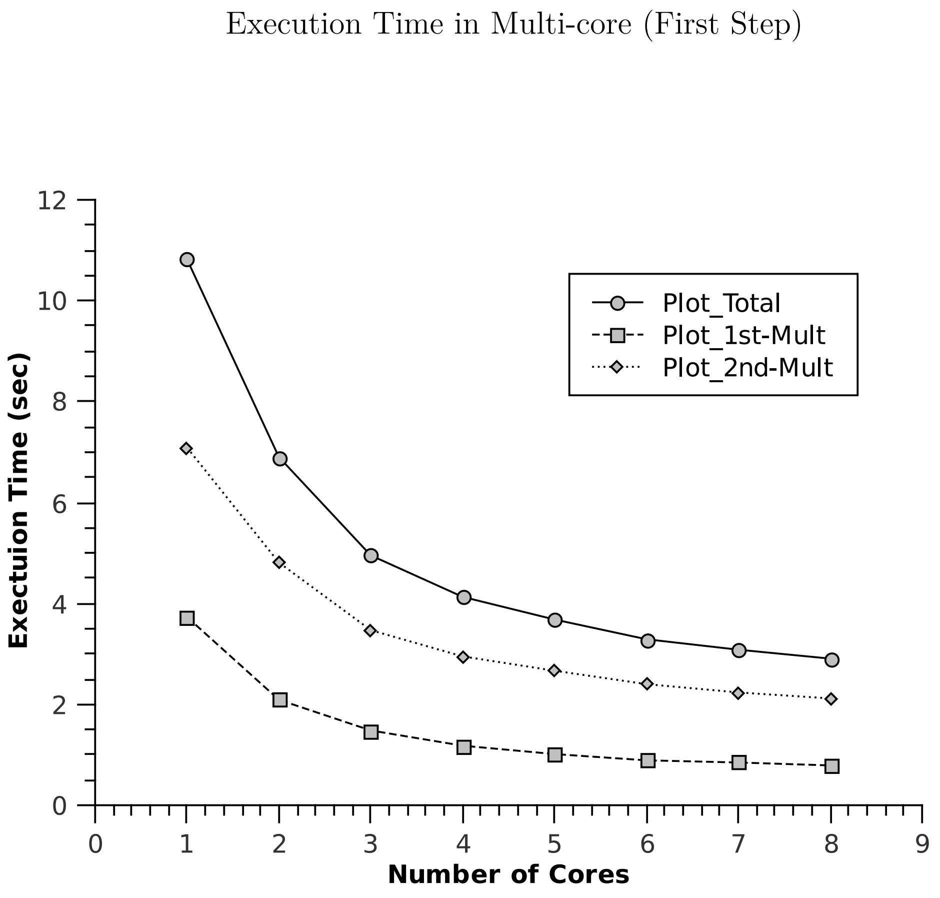

Figure 4.1: Execution Time using different numbers of Cores (1st step)

Also, from the speed-up trend, we observe that, the slope of the first one is much steeper than the second one. So, second one is expected to saturate before the first one. (According to Amdahl's Law [15], the speed up in multi-core platform is always upper-bounded by a certain limit, that is it saturates after reaching to a certain point no matter how many cores are involved. In practical situations where, there exist some portions of strictly sequential part which can not be parallelized, it always works.)

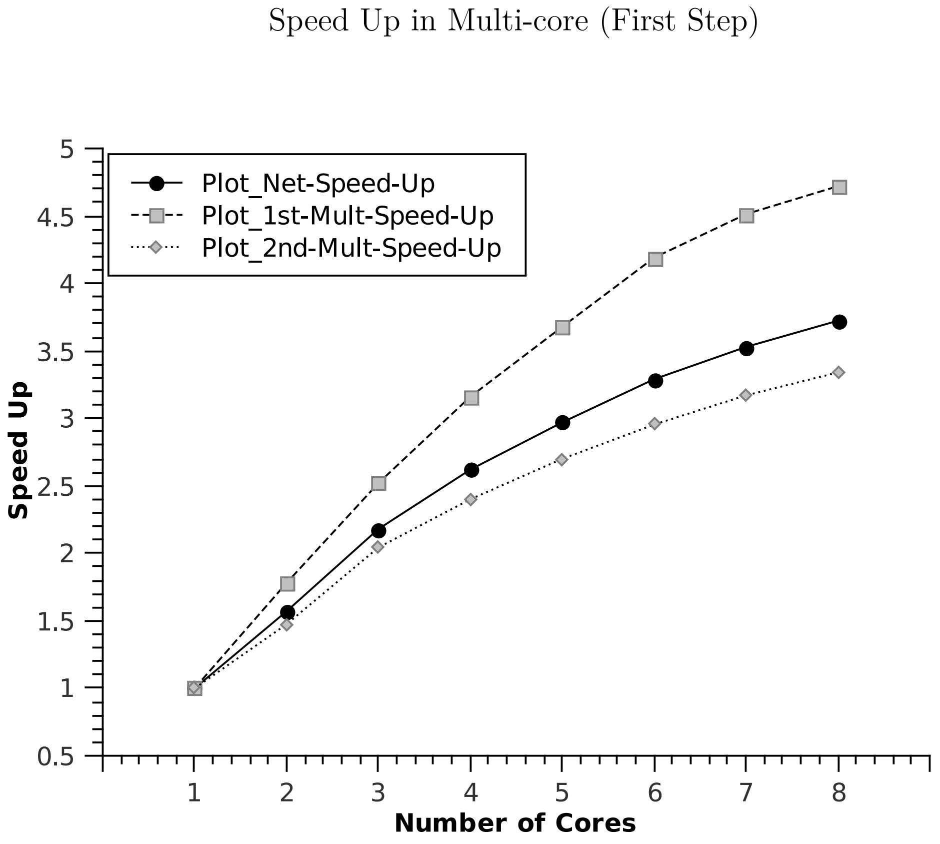

Figure 4.2: Speed up obtained using different number of Cores (1st step)

Now, since our main objective was to provide a comparative study of Levinson-Durbin's method and the Berlekamp-Massey's method, we will now try to analyze the second step implementations. From figure 4.3 we find that, the LVD algorithm is more parallelizable in the later iterations than the earlier iterations.

Because, in the earlier iterations, the chunks which are handled by each thread are of much smaller size because the vectors involved here are pretty smaller in size (discussed in detail in section 3.5.3). And that is why, the loop overhead seems to be much more significant which results in degradation in performances. However, the parallelization works better in the last iteration which takes the maximum time. In figure 4.3, the dashed curve showing the performance in the middle-most iteration can be considered as the average performance graph.

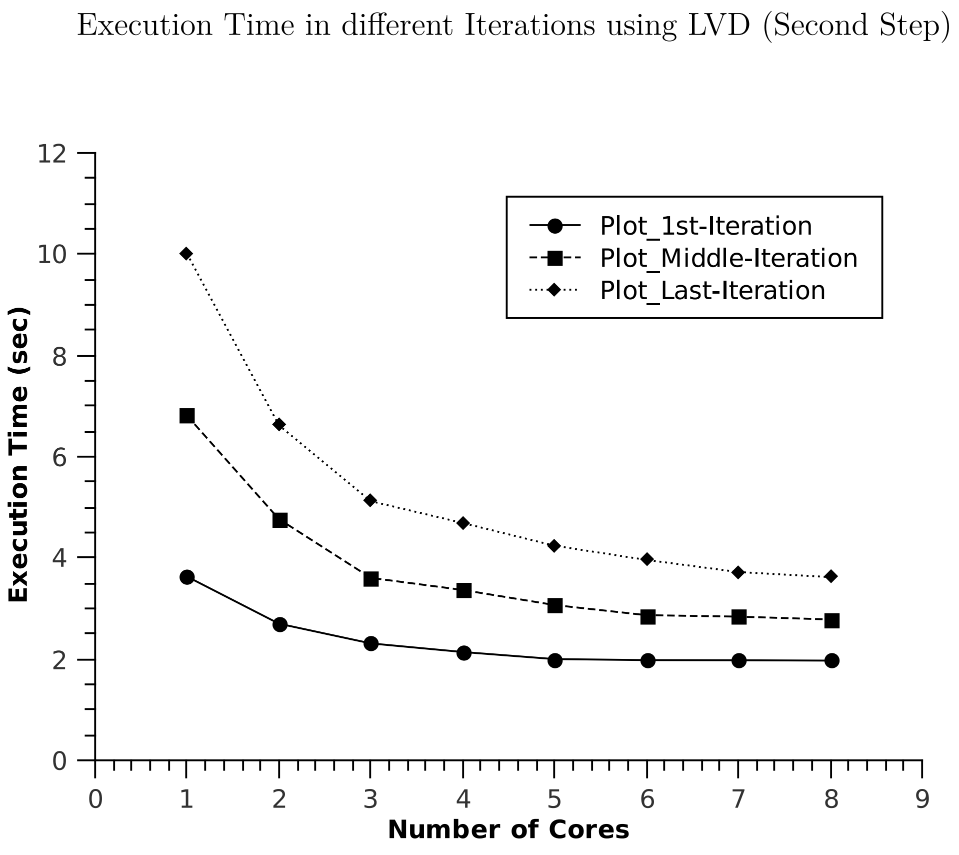

Figure 4.3: Execution Time using different numbers of Cores(LVD)

Now, analyzing figure 4.4 we can see that, BKM also shows some parallelizability even after keeping the multiplications un-parallelized. In fact, from fig. 4.4 we can observe that, in earlier iterations, the parallelization works much better than the later iterations. This is because of the fact that the polynomial multiplication takes only 5%

of total execution time of one BKM iteration in the first iteration whereas almost 20% of total in the last one. Since for the time being we keep it sequential, according to Amdahl's Law [15], the parallelization evidently works better where it consumes lesser percentage of time.

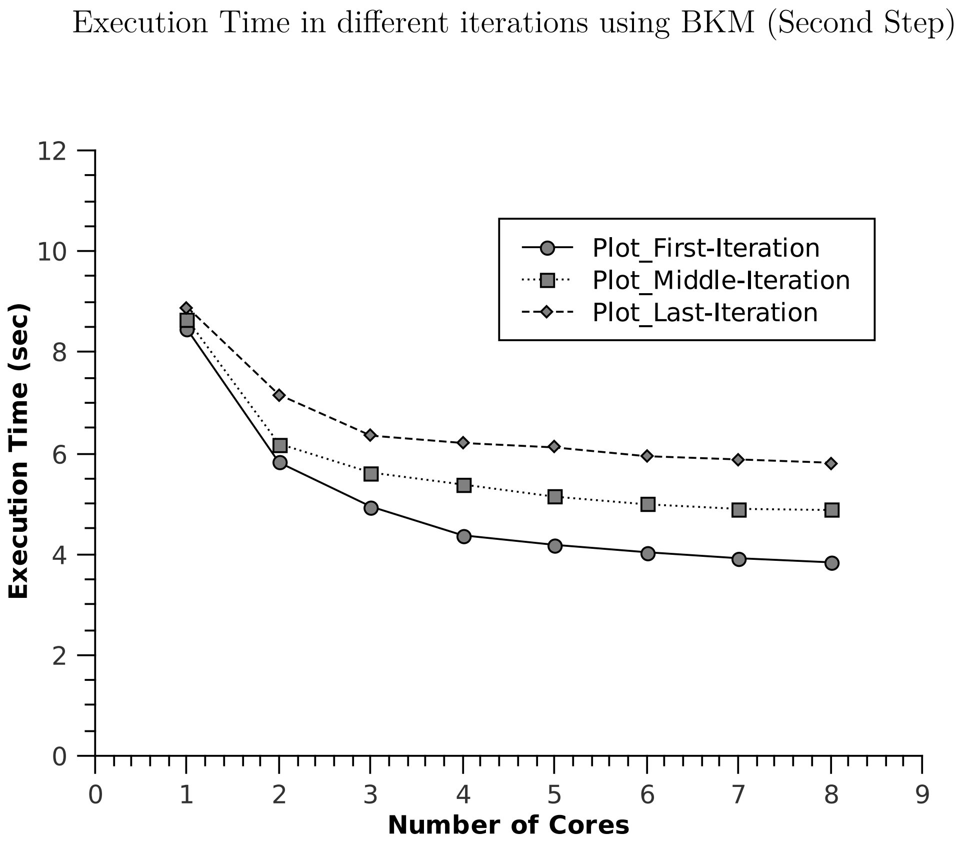

Figure 4.4: Execution Time using different numbers of Cores(BKM)

The comparison of execution time in different iterations of BKM and LVD has been depicted in fig. 4.5 over 8-core. Further, the speed-up obtained in different iterations has been shown in figure 4.6. We clearly notice that the parallelization works much better in the later iterations in LVD whereas it is the opposite in BKM. Comparing the sequential (figure 3.5) and parallel implementations (figure 4.5) we conclude that at present, in both scenario LVD seems to perform better than BKM. Also, the parallelization of LVD seems to be much easier as it responds to the techniques we used. Though, some innovative techniques to parallelize the multiplication in BKM may produce better results in the later iterations, LVD seems to keep performing better even after that improvement when averaging over all the iterations.

If we observe the speed-ups in figure 4.6, we can see that we can not achieve more than 2.81 speed up using the trivial technique. This maximum speed-up is obtained in the last iteration of LVD. For BKM, the result is even worse. The maximum speed-up 2.5 is obtained in the first iteration. A possible reason can be given from [5], which says that operations like copy-vector are not at all parallelizable.

Comparison of Execution Time in different iterations using both LVD and BKM

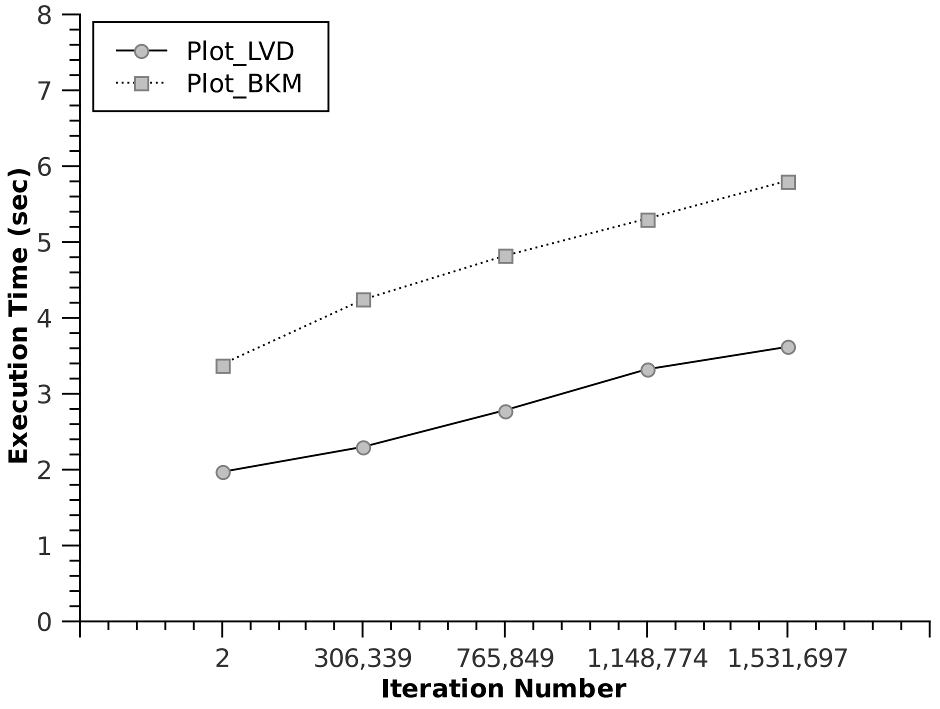

Figure 4.5: Comparative Execution Time of BKM and LVD in 8-core

Parallelizing effort is degraded by huge number of cache misses during the massive memory accessing of these operations. This kind of memory intensive operations can pull down the speed-up heavily. Now, in second step of Wiedemann Algorithm we find a lot of memory intensive operations e.g. vector addition/subtraction, which degrades the speed-up factor. The BKM is even worse in this matter. Not only a lot of inevitable copy operations are there, also the very memory intensive polynomial multiplication is also present. And, our effort (section 4.5.2) to remove the critical section from the multiplication is found to be not at all fruitful.

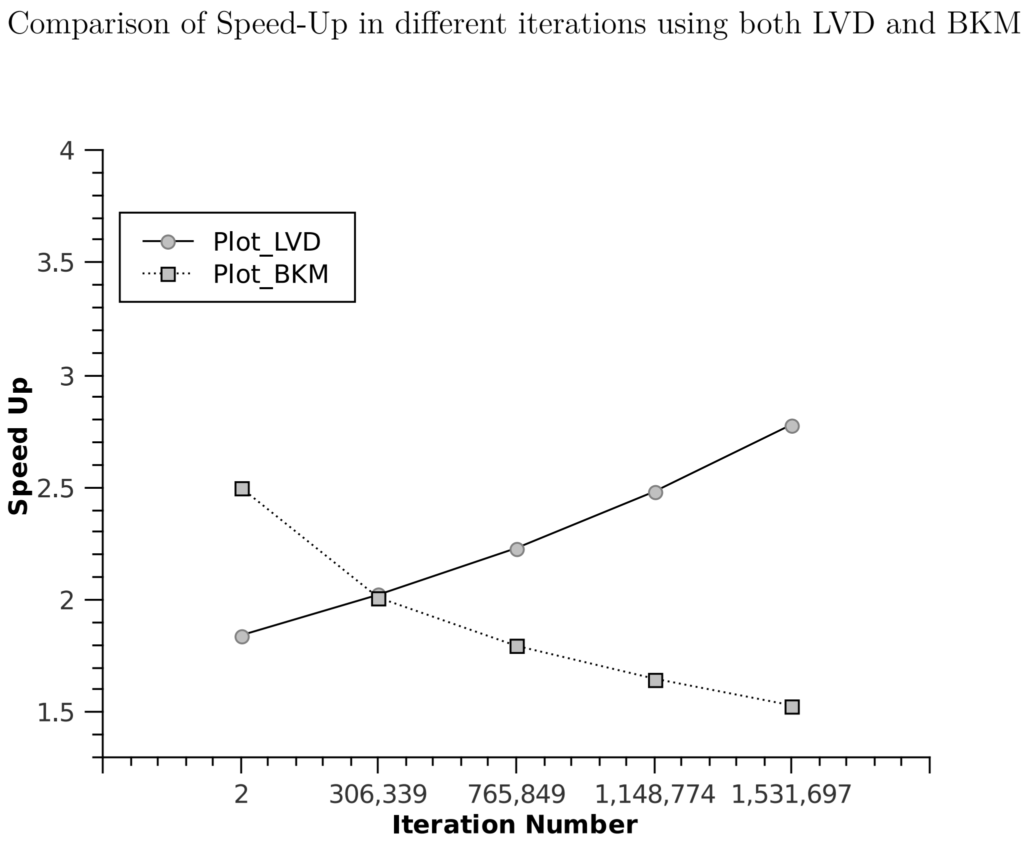

Figure 4.6: Comparative Speed-up of BKM and LVD in 8-core

4.8 Conclusion

So, as a conclusion of the multi-core implementation it can be said that, the first step, dominated by matrix-vector multiplication is found to be parallelizable using some load-balancing technique. Better load-balancing techniques will surely raise the speed-up. At least, we can say that it responds well in multi-core scenario. A better load-balancing technique can be found in the multi-core implementation of Lanczos Algorithm[5] which gives better speed-up. Since the most costly operation in Lanczos Algorithm is also the same matrix-vector multiplication, we can adapt that technique which assigns different weights to different non-zero entries according to their cost of multiplication and thus provides an excellent speed-up (4.51 for the iteration of the whole Lanczos loop where most of the time is spent computing the matrix-vector multiplication) in the same platform and using the same libraries.

But, the second step which finds minimal polynomial is found to be not quite responsive to parallelization. In fact, the LVD implementation, which is found to perform better in the sequential scenario is also found to respond better than the BKM to parallelization. Since the vectors or polynomials which are active in this part are dense, there is no non-trivial load-balancing issue. May be some good technique to make memory intensive operations faster to reduce the number of cache misses will improve the parallelization of this step. Another important point we would like to mention here. That is, here we did not experiment with the third stage. Basically, the third step is mostly dominated by the costliest matrix-vector multiplication and a few other vector addition/subtraction. All these routines are parallelized individually and nothing new is there to experiment. So, we leave the third step from all our experimentations.

Chapter 5

Conclusion and Future Direction

In this thesis, we described the efficient implementation of the Wiedemann Algorithm in sequential as well as in multi-core scenario. We investigated with two different algorithms viz. Levinson-Durbin and Berlekamp-Massey which are used to find the minimal polynomial and compared their results in sequential scenario. Also, we tried to explore the parallelizability of these two algorithms and compared their responses in multi-core platform. The main task of large sparse system solvers in $GF(p)$ is to solve the Discrete Log Problem. The most promising algorithm to do this task i.e. Lanczos Algorithm has been implemented in a multi-core scenario [5]. But, as a competing algorithm we chose Wiedemann algorithm and tried to implement it differently. In fact Wiedemann's approach is a complete different one from that of Lanczos'. And, the flexibility, this algorithm offers make it attractive to the researchers.

In this thesis, after describing the basic working principle of Wiedemann Algorithm thoroughly, we move to sequential implementation. We implemented it with its two variants mentioned earlier. In chapter 3 we described the implementation related issues and problems we faced during implementations, along with the techniques by which we overcame these. Then we provided results found from the experimentations and compared the two variants from sequential point of view. Analyzing thoroughly, we concluded with the superiority of Levinson-Durbin's approach over Berlekamp-Massey's.

In the next chapter viz. chapter 4, we described our effort to parallelize both the variants. The first step is found to have a good parallelizing potential after load-balancing non-trivially. But, in the second step, most probably due to presence of several memory intensive routines associated with a large number of cache miss, our effort was not so fruitful. Yet, the Levinson-Durbin method is found to show better parallelizing potential compare to the Berlekamp-Massey's.

There are a lot of scope to improve over our approach. First of all, the load-balancing technique we implemented here is not so good since we handle the non-zero entries differently and different non-zero entry takes different time when multiplying with large multi-precision integers. So, the technique described in [5] where each non-zero entry gets an weight according to their cost of multiplication should be implemented. Also, in [5], we can see that process-level parallelism was found to perform better compare to the thread-level which we implemented here. So, process-level parallelization technique may be tried here too.

Since we do not get good speed-up in the second step, and there is no load-balancing issue, some techniques may be implemented to improve the parallelization of memory-intensive routines. Some architecture-level analysis is needed. Also, we used the simple naive polynomial multiplication routine here to implement the Berlekamp-Massey algorithm. It can be replaced with better multiplication algorithm like Karatsuba or FFT. But, it must be mentioned that, as depicted in figure 3.5, the iterations of Berlekamp-Massey takes almost same time in each iteration. Further, the multiplication consumes less than 5% of total execution time in sequential case. So, it is quite unlikely that, even after implementing the fastest multiplication routine it can outperform Levinson-Durbin when averaged over all the iterations. Still, we can try it with FFT multiplication as it suits better for larger degree polynomials. After that, the parallelizability of that routine should be investigated.

The results we have found here are inferior than the Lanczos implementation in [5]. But since this algorithm is much flexible in a sense that it offers several choices in the second step, there are a lot of scope to improve the implementations. Also, the excellent combinatorial approach makes it elegant theoretically. But at the same time, its magnanimity in size makes it difficult for implementation. In limited time, we initiated the process providing a comparison of two different implementations. Applying more sophisticated techniques will definitely lead to better implementations in future.

Bibliography

[1] The OpenMP Application Programming Interface. http://www.openmp.org/.

[2] Leonard M. Adleman and Ming-Deh A. Huang. Function field sieve method for discrete logarithms over finite fields. Inf. Comput., 151(1-2):5–16, 1999.

[3] Bernhard Beckermann and George Labahn. A uniform approach for hermite padé and simultaneous padé approximants and their matrix-type generalizations. Numerical Algorithms, 3:45–54, 1992. 10.1007/BF02141914.

[4] Elwyn R. Berlekamp. Algebraic coding theory. McGraw-Hill Book Co., New York, 1968.

[5] Souvik Bhattacherjee and Abhijit Das. Parallelization of the lanczos algorithm on multicore platforms. In ICDCN, pages 231–241, 2010.

[6] LANCZOS C. Solution of systems of linear equations by minimized iterations. J. Res. Nat. Bur. Standards, 49:33–53, 1952.

[7] Don Coppersmith. Solving linear equations over gf(2): block lanczos algorithm. Linear Algebra and its Applications, 192:33 – 60, 1993.

[8] Don Coppersmith. Solving homogeneous linear equations over gf(2) via block wiedemann algorithm. Math. Comput., 62(205):333–350, 1994.

[9] Abhijit Das and C. E. Veni Madhavan. On the cubic sieve method for computing discrete logarithms over prime fields. Int. J. Comput. Math., 82(12):1481–1495, 2005.

[10] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, 22(6):644–654, November 1976.

[11] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Blakley and David Chaum, editors, Advances in Cryptology, volume 196 of Lecture Notes in Computer Science, pages 10–18. Springer Berlin / Heidelberg, 1985. 10.1007/3-540-39568-7_2.

[12] Ildikó Flesch. A new parallel approach to the block lanczos algorithm for finding nullspaces over GF(2). Master's thesis, Dept. of Mathematics, Utrecht University, November 2002.

[13] GNU. The GNU MP Bignum Library. http://gmplib.org/.

[14] Daniel M. Gordon. Discrete logarithms in GF(p) using the number field sieve. SIAM J. Discrete Math., 6(1):124–138, 1993.

[15] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. Computer, 41(7):33 –38, july 2008.

[16] Wontae Hwang and Dongseung Kim. Load balanced block lanczos algorithm over GF(2) for factorization of large keys. In HiPC, pages 375–386, 2006.

[17] Erich Kaltofen. Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems. In AAECC, pages 195–212, 1993.

[18] Erich Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. Algorithmica, 24(3-4):331–348, 1999.

[19] Erich Kaltofen and B. David Saunders. On wiedemann's method of solving sparse linear systems. In AAECC, pages 29–38, 1991.

[20] MD) Kravitz, David W. (Owings Mills. Digital signature algorithm. Number 5231668. July 1993.

[21] Brian A. LaMacchia and Andrew M. Odlyzko. Solving large sparse linear systems over finite fields. In Alfred Menezes and Scott A. Vanstone, editors, Advances in Cryptology-CRYPTO, volume 537 of Lecture Notes in Computer Science, pages 109–133. Springer, 1990.

[22] Arjen K. Lenstra, Hendrik W. Lenstra Jr., Mark S. Manasse, and John M. Pollard. The number field sieve. In STOC, pages 564–572, 1990.

[23] A. Lobo. Matrix free Linear System solving and applications to symbolic computation. PhD thesis, Dept of Comp. Sc., Rensselaer Polytech. Institute, Troy, New York, December 1995.

[24] Peter L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In Advances in Cryptology-EUROCRYPT, pages 106–120, 1995.

[25] Andrew M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In Advances in Cryptology-EUROCRYPT, pages 224–314, 1984.

[26] Dan Page. Parallel solution of sparse linear systems defined over gf(p). Technical Report CSTR-05-003, University of Bristol, November 2004.

[27] Olaf Penninga. Finding column dependencies in sparse systems over F_2 by block Wiedemann. Master's thesis, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, September 1998.

[28] Michael Peterson. Parallel block lanczos for solving large binary systems. Master's thesis, Dept. of Mathematics, Texas Tech University, August 2006.

[29] Carl Pomerance. The quadratic sieve factoring algorithm. In EUROCRYPT, pages 169–182, 1984.

[30] HESTENES M. R. Methods of conjugate gradients for solving linear systems. Research Jr. of the National Bureau of Standards, 49:409–436, 1952.

[31] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 21(2):120–126, 1978.

[32] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. IEEE Transactions on Information Theory, 32(1):54–62, 1986.

[33] Laurence Tianruo Yang and Richard P. Brent. The parallel improved lanczos method for integer factorization over finite fields for public key cryptosystems. In ICPP Workshops, pages 106–114, 2001.