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

Pratyay Mukherjee
Under the guidance of: Dr. Abhijit Das

Department of Computer Science & Engineering,
Indian Institute of Technology Kharagpur

May 6, 2011

Contents

Outline

Motivation

Motivation

- Security of several cryptographic schemes depend on the intractability of the **Discrete Logarithm Problem**
  - Diffie-Hellman key-agreement protocol [7].
  - ElGamal public-key cryptographic scheme [8].
  - Digital Signature Algorithm (DSA) [14].

Motivation

- Security of several cryptographic schemes depend on the intractability of the **Discrete Logarithm Problem**
  - Diffie-Hellman key-agreement protocol [7].
  - ElGamal public-key cryptographic scheme [8].
  - Digital Signature Algorithm (DSA) [14].
- What is the measurement of Intractability?

Motivation

- Security of several cryptographic schemes depend on the intractability of the **Discrete Logarithm Problem**
  - Diffie-Hellman key-agreement protocol [7].
  - ElGamal public-key cryptographic scheme [8].
  - Digital Signature Algorithm (DSA) [14].
- What is the measurement of Intractability?
  - The time taken to solve the problem.

Motivation

- Security of several cryptographic schemes depend on the intractability of the **Discrete Logarithm Problem**
  - Diffie-Hellman key-agreement protocol [7].
  - ElGamal public-key cryptographic scheme [8].
  - Digital Signature Algorithm (DSA) [14].
- What is the measurement of Intractability?
  - The time taken to solve the problem.
- Solving DLP in feasible time is one of the most important focus of modern cryptanalysis with the massive improvement in Computational Power.

Solving the Discrete Logarithm Problem

Solving the Discrete Logarithm Problem

- The fastest known algorithms for solving the discrete logarithm problem involve two stages :

Solving the Discrete Logarithm Problem

- The fastest known algorithms for solving the discrete logarithm problem involve two stages :
- Sieving step – Relations can be generated independently and massively parallelizable.
  - Linear Sieve [6].
  - Cubic Sieve [6].
  - Number Field Sieve [15, 11].
  - Function Field Sieve [2]

Solving the Discrete Logarithm Problem

- The fastest known algorithms for solving the discrete logarithm problem involve two stages :
- Sieving step – Relations can be generated independently and massively parallelizable.
  - Linear Sieve [6].
  - Cubic Sieve [6].
  - Number Field Sieve [15, 11].
  - Function Field Sieve [2]
- Sieving step generates large sparse linear systems of equations.

Solving the Discrete Logarithm Problem(..continued)

Solving the Discrete Logarithm Problem(..continued)

- Linear Algebra step –Resistant to parallelization - Bottleneck

Solving the Discrete Logarithm Problem(..continued)

- Linear Algebra step –Resistant to parallelization - Bottleneck
  - Gaussian Elimination - $\mathcal{O}(n^3)$- Not pragmatic for large system.

Solving the Discrete Logarithm Problem(..continued)

- Linear Algebra step –Resistant to parallelization - Bottleneck
  - Gaussian Elimination - $\mathcal{O}(n^3)$- Not pragmatic for large system.
  - Lanczos algorithm [4] - $\tilde{\mathcal{O}}(n^2)$ & requires n iterations.

Solving the Discrete Logarithm Problem(..continued)

- Linear Algebra step –Resistant to parallelization - Bottleneck
  - Gaussian Elimination - $\mathcal{O}(n^3)$- Not pragmatic for large system.
  - Lanczos algorithm [4] - $\tilde{\mathcal{O}}(n^2)$ & requires n iterations.
  - Wiedemann algorithm [16] - $\tilde{\mathcal{O}}(n^2)$ & requires $2n$ iterations.

Solving the Discrete Logarithm Problem(..continued)

- Linear Algebra step –Resistant to parallelization - Bottleneck
  - Gaussian Elimination - $\mathcal{O}(n^3)$- Not pragmatic for large system.
  - Lanczos algorithm [4] - $\tilde{\mathcal{O}}(n^2)$ & requires n iterations.
  - Wiedemann algorithm [16] - $\tilde{\mathcal{O}}(n^2)$ & requires $2n$ iterations.
- We aim to study the Wiedemann Algorithm and implement it efficiently over a multi-core platform.

Related Work

Related Work

- Most reported works on parallelizing iterative solvers focus on solving systems over $GF(2)$, generated from the integer factorization algorithms. [17, 9, 12, 5]

Related Work

- Most reported works on parallelizing iterative solvers focus on solving systems over $GF(2)$, generated from the integer factorization algorithms. [17, 9, 12, 5]
- Representing elements over $GF(2)$ requires much less space compared to $GF(p)$.

Related Work

- Most reported works on parallelizing iterative solvers focus on solving systems over $GF(2)$, generated from the integer factorization algorithms. [17, 9, 12, 5]
- Representing elements over $GF(2)$ requires much less space compared to $GF(p)$.
- Further, for systems over $GF(2)$, it suffices to perform only efficient bitwise operations instead of expensive multi-precision modular operations needed for $GF(p)$.

Related Work

- Most reported works on parallelizing iterative solvers focus on solving systems over $GF(2)$, generated from the integer factorization algorithms. [17, 9, 12, 5]
- Representing elements over $GF(2)$ requires much less space compared to $GF(p)$.
- Further, for systems over $GF(2)$, it suffices to perform only efficient bitwise operations instead of expensive multi-precision modular operations needed for $GF(p)$.
- Therefore, we concentrate on systems of linear equations over $GF(p)$—a topic that has not received substantial research attention.

Most Significant Work

Most Significant Work

- In exactly same platform, the implementation of Lanczos Algorithm [3] over a multi-core platform and also over multi-node multi-core platform was quite successful.

Most Significant Work

- In exactly same platform, the implementation of Lanczos Algorithm [3] over a multi-core platform and also over multi-node multi-core platform was quite successful.
- The above mentioned work has shown quite attractive speed up (4.51 using same library & 6.57 using different library).

Outline

Assumptions

Assumptions

- Sieving step gives an $m \times n$ matrix B over prime field $GF(p)$ with $m > n$ to represent the linear system:

$$B\mathbf{x} \equiv \mathbf{u}(mod p) \tag{1}$$

Assumptions

- Sieving step gives an $m \times n$ matrix B over prime field $GF(p)$ with $m > n$ to represent the linear system:

$$B\mathbf{x} \equiv \mathbf{u}(mod p) \tag{1}$$

- The equations are consistent and $\mathbf{u}$ is in the column space of B.

Assumptions

- Sieving step gives an $m \times n$ matrix B over prime field $GF(p)$ with $m > n$ to represent the linear system:

$$B\mathbf{x} \equiv \mathbf{u}(modp) \quad (1)$$

- The equations are consistent and $\mathbf{u}$ is in the column space of B.
- The matrix B must be of full column rank as the solution of *DLP* must be unique.

Preparing the inputs

Preparing the inputs

- Now, the *Wiedemann algorithm* is classically applicable to systems of the following form:

$$A\mathbf{x} = \mathbf{b} \tag{2}$$

where A is a square matrix of dimension $n \times n$, $\mathbf{u}$ and $\mathbf{x}$ are vectors of dimension $n \times 1$. In order to fit this algorithm to our case, we transform Eqn(1) to Eqn(2) by letting

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

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

Preparing the inputs

- Now, the *Wiedemann algorithm* is classically applicable to systems of the following form:

$$A\mathbf{x} = \mathbf{b} \tag{2}$$

where A is a square matrix of dimension $n \times n$, **u** and **x** are vectors of dimension $n \times 1$. In order to fit this algorithm to our case, we transform Eqn(1) to Eqn(2) by letting

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

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

- A need not be symmetric or positive definite.- **Advantage over Lanczos**

The Algorithm

The Algorithm

- Step 1:

The Algorithm

- Step 1:
  - Generate a random vector **v** of dimension $n \times 1$.

The Algorithm

- Step 1:
  - Generate a random vector **v** of dimension $n \times 1$.
  - Multiply A to **v** repeatedly to generate $A^i\mathbf{v}$ where $i = 0, 1, ...2n - 1$.(n is the dimension of A).

The Algorithm

- Step 1:
  - Generate a random vector $\mathbf{v}$ of dimension $n \times 1$.
  - Multiply A to $\mathbf{v}$ repeatedly to generate $A^i\mathbf{v}$ where $i = 0, 1, ...2n - 1$.(n is the dimension of A).
- Step 2:

The Algorithm

- Step 1:
  - Generate a random vector **v** of dimension $n \times 1$.
  - Multiply A to **v** repeatedly to generate $A^i\mathbf{v}$ where $i = 0, 1, ...2n - 1$.(n is the dimension of A).
- Step 2:
  - The elements of $A^i\mathbf{v}$ are fed to an algorithm(say *Min_Poly*) which finds the *minimal polynomial* for A.

The Algorithm

- Step 1:
  - Generate a random vector **v** of dimension $n \times 1$.
  - Multiply A to **v** repeatedly to generate $A^i\mathbf{v}$ where $i = 0, 1, ...2n - 1$.(n is the dimension of A).
- Step 2:
  - The elements of $A^i\mathbf{v}$ are fed to an algorithm(say *Min_Poly*) which finds the *minimal polynomial* for A.
- Step 3:

The Algorithm

- Step 1:
  - Generate a random vector **v** of dimension $n \times 1$.
  - Multiply A to **v** repeatedly to generate $A^i\mathbf{v}$ where $i = 0, 1, ...2n - 1$.(n is the dimension of A).
- Step 2:
  - The elements of $A^i\mathbf{v}$ are fed to an algorithm(say *Min_Poly*) which finds the *minimal polynomial* for A.
- Step 3:
  - $A^i\mathbf{b}$ is computed for $i = 0, 1, ..., n - 1$.

The Algorithm

- Step 1:
  - Generate a random vector **v** of dimension $n \times 1$.
  - Multiply A to **v** repeatedly to generate $A^i\mathbf{v}$ where $i = 0, 1, ...2n - 1$.(n is the dimension of A).
- Step 2:
  - The elements of $A^i\mathbf{v}$ are fed to an algorithm(say *Min_Poly*) which finds the *minimal polynomial* for A.
- Step 3:
  - $A^i\mathbf{b}$ is computed for $i = 0, 1, ..., n - 1$.
  - This values are substituted for the variable with apt degree of the *minimal polynomial* to compute the solution **x**.

The Minimal Polynomial

The Minimal Polynomial

- The *characteristic equation* of A is

$$\chi_A(x) = det(xI - A), \tag{5}$$

The Minimal Polynomial

- The *characteristic equation* of A is

$$\chi_A(x) = det(xI - A), \tag{5}$$

- Again, $\chi_A(A) = 0$ (*Calley-Hamilton theorem*).

The Minimal Polynomial

- The *characteristic equation* of A is

$$\chi_A(x) = det(xI - A), \tag{5}$$

- Again, $\chi_A(A) = 0$ (*Calley-Hamilton theorem*).
- The *minimal polynomial* $\mu_A(x) | \chi_A(x)$ in $K[x]$.

The Minimal Polynomial

- The *characteristic equation* of A is

$$\chi_A(x) = det(xI - A), \tag{5}$$

- Again, $\chi_A(A) = 0$ (*Calley-Hamilton theorem*).
- The *minimal polynomial* $\mu_A(x) | \chi_A(x)$ in $K[x]$.
- Assume, the *minimal polynomial*

$$\mu_A(x) = x^d - c_{d-1}x^{d-1} - c_{d-2}x^{d-2} - ..... - c_1 x - c_0 \in K[x] \tag{6}$$

The Computation

The Computation

- Since $\mu_A(A) = 0$, for any $n \times 1$ non-zero vector $\mathbf{v}$ and for any integer $k \geq d$, we have :

$$A^k\mathbf{v} - c_{d-1}A^{k-1}\mathbf{v} - .....c_1A^{k-d+1}\mathbf{v} - c_0A^{k-d}\mathbf{v} = 0. \quad (7)$$

The Computation

- Since $\mu_A(A) = 0$, for any $n \times 1$ non-zero vector $\mathbf{v}$ and for any integer $k \geq d$, we have :

$$A^k\mathbf{v} - c_{d-1}A^{k-1}\mathbf{v} - .....c_1A^{k-d+1}\mathbf{v} - c_0A^{k-d}\mathbf{v} = 0. \qquad (7)$$

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

$$v_k = c_{d-1}v_{k-1} + c_{d-2}v_{k-2} + ... + c_1v_1 + c_0v_0 \qquad (8)$$

The Computation

- Since $\mu_A(A) = 0$, for any $n \times 1$ non-zero vector $\mathbf{v}$ and for any integer $k \geq d$, we have :

$$A^k\mathbf{v} - c_{d-1}A^{k-1}\mathbf{v} - .....c_1A^{k-d+1}\mathbf{v} - c_0A^{k-d}\mathbf{v} = 0. \qquad (7)$$

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

$$v_k = c_{d-1}v_{k-1} + c_{d-2}v_{k-2} + ... + c_1v_1 + c_0v_0 \qquad (8)$$

- The sub-algorithm *Min_Poly* finds the *minimal polynomial* from the v_j's ($j = 0, 1, 2, ..., k$).

The Computation (contd...)

The Computation (contd...)

- For computing a solution of $A\mathbf{x} = \mathbf{b}$,Putting $k = d$ and $\mathbf{v} = \mathbf{b}$ in Eqn.(7) yields:

$$A(A^{d-1}\mathbf{b} - c_{d-1}A^{d-2}\mathbf{b} - c_{d-2}A^{d-3}\mathbf{b} - \ldots\ldots - c_1A\mathbf{b}) = c_0\mathbf{b}, \quad (9)$$

The Computation (contd...)

- For computing a solution of $A\mathbf{x} = \mathbf{b}$,Putting $k = d$ and $\mathbf{v} = \mathbf{b}$ in Eqn.(7) yields:

$$A(A^{d-1}\mathbf{b} - c_{d-1}A^{d-2}\mathbf{b} - c_{d-2}A^{d-3}\mathbf{b} - \ldots\ldots - c_1 A\mathbf{b}) = c_0\mathbf{b}, \tag{9}$$

- That is, if $c_0 \neq 0$,it becomes:

$$\mathbf{x} = c_0^{-1}(A^{d-1}\mathbf{b} - c_{d-1}A^{d-2}\mathbf{b} - c_{d-2}A^{d-3}\mathbf{b} - \ldots\ldots - c_1 A\mathbf{b}) \tag{10}$$

which is a solution of $A\mathbf{x} = \mathbf{b}$.

Competing Algorithms as *Min_Poly*

Competing Algorithms as *Min_Poly*

- We used two different algorithms.

Competing Algorithms as *Min_Poly*

- We used two different algorithms.
- First one is **Berlekamp-Massey Algorithm** as classically used by Wiedemann.

Competing Algorithms as *Min_Poly*

- We used two different algorithms.
- First one is **Berlekamp-Massey Algorithm** as classically used by Wiedemann.
- Second one is **Levinson-Durbin Algorithm** first proposed by Kaltofen [13] to use here.

Competing Algorithms as *Min_Poly*

- We used two different algorithms.
- First one is **Berlekamp-Massey Algorithm** as classically used by Wiedemann.
- Second one is **Levinson-Durbin Algorithm** first proposed by Kaltofen [13] to use here.
- Target is to compare the performances of these algorithms both in sequential and parallel scenario.

Berlekamp-Massey Algorithm(features)

Berlekamp-Massey Algorithm(features)

- Iterative algorithm based on extended GCD algorithm.

Berlekamp-Massey Algorithm(features)

- Iterative algorithm based on extended GCD algorithm.
- Involves basic polynomial arithmetic routines.

Berlekamp-Massey Algorithm(features)

- Iterative algorithm based on extended GCD algorithm.
- Involves basic polynomial arithmetic routines.
- Overall running time is $\mathcal{O}(d^2)$ for d iterations.

Levinson-Durbin Algorithm(Features)

Levinson-Durbin Algorithm(Features)

- Iterative algorithm involving basic vector arithmetics.

Levinson-Durbin Algorithm(Features)

- Iterative algorithm involving basic vector arithmetics.
- Toeplitz Matrix is formed from the vector elements.

Levinson-Durbin Algorithm(Features)

- Iterative algorithm involving basic vector arithmetics.
- Toeplitz Matrix is formed from the vector elements.
- Iteratively solve Toeplitz system $T^i\mathbf{x}^i = \mathbf{b}^i$.

Levinson-Durbin Algorithm(Features)

- Iterative algorithm involving basic vector arithmetics.
- Toeplitz Matrix is formed from the vector elements.
- Iteratively solve Toeplitz system $T^i \mathbf{x}^i = \mathbf{b}^i$.
- Overall running time is $\mathcal{O}(d^2)$ for d iterations.

Outline

Problem with trivial storage

Problem with trivial storage

- Can not be stored in 2-dimensional array form.

Problem with trivial storage

- Can not be stored in 2-dimensional array form.
- Memory overflow occurs if dimension is too large ($\approx 10^6$).

Problem with trivial storage

- Can not be stored in 2-dimensional array form.
- Memory overflow occurs if dimension is too large ($\approx 10^6$).
- Computation becomes inefficient.

Problem with trivial storage

- Can not be stored in 2-dimensional array form.
- Memory overflow occurs if dimension is too large ($\approx 10^6$).
- Computation becomes inefficient.
- Since the matrix is very sparse, only storing non-zero entry should suffice.

Efficiently storing the sparse matrix

Efficiently storing the sparse matrix

- Can be stored as an array of triplet(data, row, column).

Efficiently storing the sparse matrix

- Can be stored as an array of triplet(data, row, column).
- Can be stored in *Compressed Row Storage(CRS)* format.

Efficiently storing the sparse matrix

- Can be stored as an array of triplet(data, row, column).
- Can be stored in *Compressed Row Storage(CRS)* format.
- Can be stored in *Compressed Column Storage(CCS)* format.

Example(Compressed Row Format)

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

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

Using Multiple-Precision Library

Using Multiple-Precision Library

- The elements of vectors belong to $GF(p)$ where p may be as big as 512 bits.

Using Multiple-Precision Library

- The elements of vectors belong to $GF(p)$ where p may be as big as 512 bits.
- Normal integer variable can not handle the values.

Using Multiple-Precision Library

- The elements of vectors belong to $GF(p)$ where p may be as big as 512 bits.
- Normal integer variable can not handle the values.
- We used GNU/MP [10] multiple-precision library (version 4.3.1) for integer field arithmetics.

Special structure of the input matrix

Special structure of the input matrix

- A very sparse $m \times n$ matrix B, with $m > n$.

Special structure of the input matrix

- A very sparse $m \times n$ matrix B, with $m > n$.
- n columns can be viewed as t and $2M + 1$ separate columns and $M \gg t$.

Special structure of the input matrix

- A very sparse $m \times n$ matrix B, with $m > n$.
- n columns can be viewed as t and $2M + 1$ separate columns and $M \gg t$.
- For the first t columns, i-th column approximately contains $\frac{m}{p_i}$ non-zero entries.(p_i is the i-th prime)

Special structure of the input matrix

- A very sparse $m \times n$ matrix B, with $m > n$.
- n columns can be viewed as t and $2M + 1$ separate columns and $M \gg t$.
- For the first t columns, i-th column approximately contains $\frac{m}{p_i}$ non-zero entries.(p_i is the i-th prime)
- For the next $2M + 1$ columns, each row contains exactly two -1.

Special structure of the input matrix

- A very sparse $m \times n$ matrix B, with $m > n$.
- n columns can be viewed as t and $2M + 1$ separate columns and $M \gg t$.
- For the first t columns, i-th column approximately contains $\frac{m}{p_i}$ non-zero entries.(p_i is the i-th prime)
- For the next $2M + 1$ columns, each row contains exactly two -1.
- Almost three-fourths of the non-zero entries are $+1$. Most of the other entries are -1.

Special structure of the input matrix

- A very sparse $m \times n$ matrix B, with $m > n$.
- n columns can be viewed as t and $2M + 1$ separate columns and $M \gg t$.
- For the first t columns, i-th column approximately contains $\frac{m}{p_i}$ non-zero entries.(p_i is the i-th prime)
- For the next $2M + 1$ columns, each row contains exactly two -1.
- Almost three-fourths of the non-zero entries are $+1$. Most of the other entries are -1.
- Non-zero entries lie in $[-2, 50]$.

Avoiding explicit matrix-multiplication

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.
- Wiedemann Algorithm takes a square matrix as input.

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.
- Wiedemann Algorithm takes a square matrix as input.
- Multiplying B^t to B makes a square matrix A.

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.
- Wiedemann Algorithm takes a square matrix as input.
- Multiplying B^t to B makes a square matrix A.
- A is not as sparse as B.

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.
- Wiedemann Algorithm takes a square matrix as input.
- Multiplying B^t to B makes a square matrix A.
- A is not as sparse as B.
- Also the matrix multiplication B^tB is very costly.

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.
- Wiedemann Algorithm takes a square matrix as input.
- Multiplying B^t to B makes a square matrix A.
- A is not as sparse as B.
- Also the matrix multiplication B^tB is very costly.
- It is avoided keeping (B^tB) as it is: B and B^t stored separately.

Avoiding explicit matrix-multiplication

- The matrix (B) coming from sieving step is overdetermined.
- Wiedemann Algorithm takes a square matrix as input.
- Multiplying B^t to B makes a square matrix A.
- A is not as sparse as B.
- Also the matrix multiplication B^tB is very costly.
- It is avoided keeping (B^tB) as it is: B and B^t stored separately.
- The multiplication (Av) is replaced by two successive multiplications: (Bv) and $B^t(Bv)$.

Reducing number of multiplications

Reducing number of multiplications

- About three fourth of matrix entries are ± 1.

Reducing number of multiplications

- About three fourth of matrix entries are ± 1.
- Multiplication and addition/subtraction takes place in loop of matrix-vector multiplication.

Reducing number of multiplications

- About three fourth of matrix entries are ± 1.
- Multiplication and addition/subtraction takes place in loop of matrix-vector multiplication.
- For $+1$ only addition and for -1 only subtraction suffices.

Reducing number of multiplications

- About three fourth of matrix entries are ± 1.
- Multiplication and addition/subtraction takes place in loop of matrix-vector multiplication.
- For $+1$ only addition and for -1 only subtraction suffices.
- Multiplication is needed only when matrix entry is other than ± 1.

Reducing number of multiplications

- About three fourth of matrix entries are ± 1.
- Multiplication and addition/subtraction takes place in loop of matrix-vector multiplication.
- For $+1$ only addition and for -1 only subtraction suffices.
- Multiplication is needed only when matrix entry is other than ± 1.
- Multiplication is avoided in most of the cases.

Delaying modulo prime operation

Delaying modulo prime operation

- The modulo operation is costly one.

Delaying modulo prime operation

- The modulo operation is costly one.
- This can be done after computing the product $\sum b_r v_r$ for a row.

Delaying modulo prime operation

- The modulo operation is costly one.
- This can be done after computing the product $\sum b_r v_r$ for a row.
- Matrix elements are single precision integers: The word size of product may be slightly larger than p.

Experimental set-up

Experimental set-up

- A special matrix of dimension $2.2mil \times 1.6mil$ is taken.

Experimental set-up

- A special matrix of dimension $2.2mil \times 1.6mil$ is taken.
- Each step of the algorithm is experimented separately running a few iterations.

Experimental set-up

- A special matrix of dimension $2.2mil \times 1.6mil$ is taken.
- Each step of the algorithm is experimented separately running a few iterations.
- Each step is dependent on the previous step: We could not afford running a step fully.

Experimental set-up

- A special matrix of dimension $2.2mil \times 1.6mil$ is taken.
- Each step of the algorithm is experimented separately running a few iterations.
- Each step is dependent on the previous step: We could not afford running a step fully.
- Random data generated to give input to next step.

Representing B^t in CRS

Representing B^t in CRS

- Initially B^t was represented in CCS which is CRS of B: Space was saved.

Representing B^t in CRS

- Initially B^t was represented in CCS which is CRS of B: Space was saved.
- The matrix-vector multiplication takes elements row-wise.

Representing B^t in CRS

- Initially B^t was represented in CCS which is CRS of B: Space was saved.
- The matrix-vector multiplication takes elements row-wise.
- So, the second multiplication was taking enormous amount of time for very large matrix.

Representing B^t in CRS

- Initially B^t was represented in CCS which is CRS of B: Space was saved.
- The matrix-vector multiplication takes elements row-wise.
- So, the second multiplication was taking enormous amount of time for very large matrix.
- Explicitly the transpose of B is computed. Extra space needed to store CRS of B^t, but huge of gain in time.

Representing B^t in CRS

- Initially B^t was represented in CCS which is CRS of B: Space was saved.
- The matrix-vector multiplication takes elements row-wise.
- So, the second multiplication was taking enormous amount of time for very large matrix.
- Explicitly the transpose of B is computed. Extra space needed to store CRS of B^t, but huge of gain in time.
- One iteration of matrix-vector multiplication takes 22 sec.

Reducing Cache miss

Reducing Cache miss

- Observation: whenever the value of matrix is accessed, immediately the column-index of that value is accessed.

Reducing Cache miss

- Observation: whenever the value of matrix is accessed, immediately the column-index of that value is accessed.
- In CRS form values and column-indices are stored in different arrays.

Reducing Cache miss

- Observation: whenever the value of matrix is accessed, immediately the column-index of that value is accessed.
- In CRS form values and column-indices are stored in different arrays.
- The two arrays are merged into one single structure.

Reducing Cache miss

- Observation: whenever the value of matrix is accessed, immediately the column-index of that value is accessed.
- In CRS form values and column-indices are stored in different arrays.
- The two arrays are merged into one single structure.
- Reduce cache misses while accessing the large CRS structure.

Reducing Cache miss

- Observation: whenever the value of matrix is accessed, immediately the column-index of that value is accessed.
- In CRS form values and column-indices are stored in different arrays.
- The two arrays are merged into one single structure.
- Reduce cache misses while accessing the large CRS structure.
- After this: one iteration of matrix-vector multiplication takes 12.5 sec.

Using faster functions

Using faster functions

- In the matrix-vector multiplication loop initially multiplication and addition routines are used separately.

Using faster functions

- In the matrix-vector multiplication loop initially multiplication and addition routines are used separately.
- They are replaced by one single `add_mul` routine in GNU/MP.

Using faster functions

- In the matrix-vector multiplication loop initially multiplication and addition routines are used separately.
- They are replaced by one single `add_mul` routine in GNU/MP.
- It is lower level function and works much faster.

Using faster functions

- In the matrix-vector multiplication loop initially multiplication and addition routines are used separately.
- They are replaced by one single `add_mul` routine in GNU/MP.
- It is lower level function and works much faster.
- After this: one iteration of matrix-vector multiplication takes 10.82 sec.

Results (1st step)

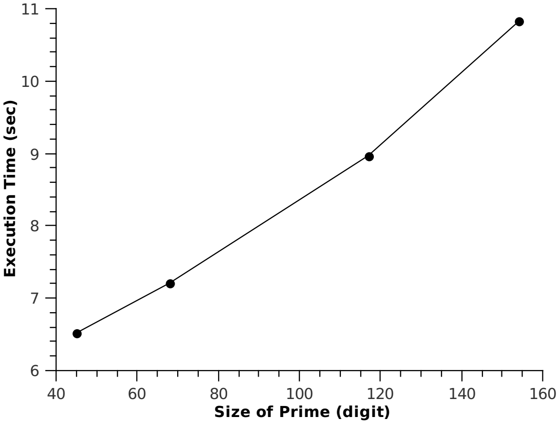

Figure: Size of prime vs Execution time in First Step

Results (2nd step)

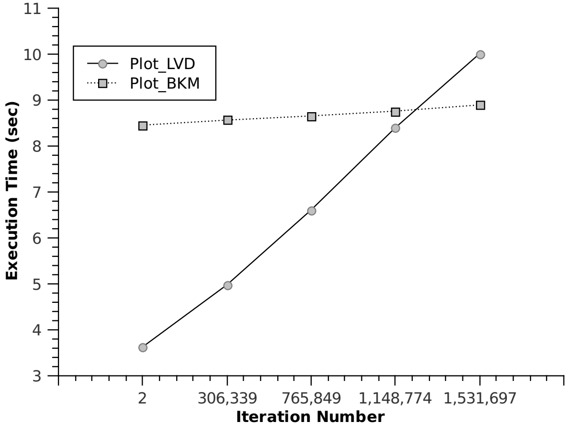

Figure: Comparative Execution Time in different iterations

Observations

Observations

- The first step execution time varies linearly with size of prime as expected.

Observations

- The first step execution time varies linearly with size of prime as expected.
- Levinson-Durbin (LVD) takes more time in later iterations.(varies from 3.7 sec to 10.1sec)

Observations

- The first step execution time varies linearly with size of prime as expected.
- Levinson-Durbin (LVD) takes more time in later iterations.(varies from 3.7 sec to 10.1sec)
- Berlekamp-Massey (BKM) takes almost same time in every iterations.(varies from 8.4 sec to 8.9 sec)

Observations

- The first step execution time varies linearly with size of prime as expected.
- Levinson-Durbin (LVD) takes more time in later iterations.(varies from 3.7 sec to 10.1sec)
- Berlekamp-Massey (BKM) takes almost same time in every iterations.(varies from 8.4 sec to 8.9 sec)
- LVD performs better in almost 3/4 -th of total iterations.

Observations

- The first step execution time varies linearly with size of prime as expected.
- Levinson-Durbin (LVD) takes more time in later iterations.(varies from 3.7 sec to 10.1sec)
- Berlekamp-Massey (BKM) takes almost same time in every iterations.(varies from 8.4 sec to 8.9 sec)
- LVD performs better in almost 3/4 -th of total iterations.
- Averaging over all iterations LVD seems to perform better than BKM.

Analysis

Analysis

- BKM computes four polynomials in each iteration.

Analysis

- BKM computes four polynomials in each iteration.
- Two polynomials are initially big (degree $2n$ and $2n-1$) and decreases in degree with increasing iteration number.

Analysis

- BKM computes four polynomials in each iteration.
- Two polynomials are initially big (degree $2n$ and $2n-1$) and decreases in degree with increasing iteration number.
- Two other polynomials are initially small (degree 0) and increases in degree with increasing iteration.

Analysis

- BKM computes four polynomials in each iteration.
- Two polynomials are initially big (degree $2n$ and $2n - 1$) and decreases in degree with increasing iteration number.
- Two other polynomials are initially small (degree 0) and increases in degree with increasing iteration.
- Every iteration has to deal large polynomials: Takes comparable time in each iteration.

Analysis (...contd)

Analysis (...contd)

- LVD computes three vectors and four scalars in each iteration.

Analysis (...contd)

- LVD computes three vectors and four scalars in each iteration.
- The size of vectors are equal to the iteration number.

Analysis (...contd)

- LVD computes three vectors and four scalars in each iteration.
- The size of vectors are equal to the iteration number.
- Earlier iterations deal with smaller vectors: Results much lesser time.

Analysis (...contd)

- LVD computes three vectors and four scalars in each iteration.
- The size of vectors are equal to the iteration number.
- Earlier iterations deal with smaller vectors: Results much lesser time.
- The number of basic operations is greater than BKM : Results greater time than BKM in later iterations where the size of polynomials and vectors are close.

Outline

Parallel implementation issues

Parallel implementation issues

- Each step can only starts after the earlier step finishes computation.

Parallel implementation issues

- Each step can only starts after the earlier step finishes computation.
- Each iteration in a step depends on the previous one.

Parallel implementation issues

- Each step can only starts after the earlier step finishes computation.
- Each iteration in a step depends on the previous one.
- Only possibility: To parallelize individual arithmetic routines.

Parallel implementation issues (...contd)

Parallel implementation issues (...contd)

- The vectors/polynomials are dense.

Parallel implementation issues (...contd)

- The vectors/polynomials are dense.
- The routines handling only vectors/polynomials are inherently parallelizable: No explicit load-balancing required.

Parallel implementation issues (...contd)

- The vectors/polynomials are dense.
- The routines handling only vectors/polynomials are inherently parallelizable: No explicit load-balancing required.
- Matrix-vector multiplication is the costliest operation and trivially not parallelizable.

Problem with trivial parallelization

Problem with trivial parallelization

- Two matrix-vector multiplications : Bv and $B^t(Bv)$.

Problem with trivial parallelization

- Two matrix-vector multiplications : Bv and $B^t(Bv)$.
- Rows of B are almost balanced: First multiplication responds moderately in trivial parallelization.

Problem with trivial parallelization

- Two matrix-vector multiplications : Bv and $B^t(Bv)$.
- Rows of B are almost balanced: First multiplication responds moderately in trivial parallelization.
- Columns of B are quite imbalanced: Second multiplication responds poorly in trivial parallelization.

Problem with trivial parallelization

- Two matrix-vector multiplications : Bv and $B^t(Bv)$.
- Rows of B are almost balanced: First multiplication responds moderately in trivial parallelization.
- Columns of B are quite imbalanced: Second multiplication responds poorly in trivial parallelization.
- Trivial parallelization results:

Problem with trivial parallelization

- Two matrix-vector multiplications : Bv and $B^t(Bv)$.
- Rows of B are almost balanced: First multiplication responds moderately in trivial parallelization.
- Columns of B are quite imbalanced: Second multiplication responds poorly in trivial parallelization.
- Trivial parallelization results:
  - First multiplication takes 1.34 sec over 8-core (sequential takes 3.73 sec)

Problem with trivial parallelization

- Two matrix-vector multiplications : Bv and $B^t(Bv)$.
- Rows of B are almost balanced: First multiplication responds moderately in trivial parallelization.
- Columns of B are quite imbalanced: Second multiplication responds poorly in trivial parallelization.
- Trivial parallelization results:
  - First multiplication takes 1.34 sec over 8-core (sequential takes 3.73 sec)
  - Second multiplication takes 6.25 sec over 8-core (sequential takes 7.09 sec)

A non-trivial load balancing technique

A non-trivial load balancing technique

- The matrix B is very sparse and non-zero entries are scattered throughout the matrix.

A non-trivial load balancing technique

- The matrix B is very sparse and non-zero entries are scattered throughout the matrix.
- The non-zero entries are distributed equally to each thread/processor.

A non-trivial load balancing technique

- The matrix B is very sparse and non-zero entries are scattered throughout the matrix.
- The non-zero entries are distributed equally to each thread/processor.
- Total number of elements is divided by number of processors.

A non-trivial load balancing technique

- The matrix B is very sparse and non-zero entries are scattered throughout the matrix.
- The non-zero entries are distributed equally to each thread/processor.
- Total number of elements is divided by number of processors.
- The row numbers corresponding to initial loop variable is pre-computed.

A non-trivial load balancing technique

- The matrix B is very sparse and non-zero entries are scattered throughout the matrix.
- The non-zero entries are distributed equally to each thread/processor.
- Total number of elements is divided by number of processors.
- The row numbers corresponding to initial loop variable is pre-computed.
- One row (starting/ending row) can be computed by two threads: This case is handled carefully.

A non-trivial load balancing technique

- The matrix B is very sparse and non-zero entries are scattered throughout the matrix.
- The non-zero entries are distributed equally to each thread/processor.
- Total number of elements is divided by number of processors.
- The row numbers corresponding to initial loop variable is pre-computed.
- One row (starting/ending row) can be computed by two threads: This case is handled carefully.
- Net Speed-up obtained: 3.72 (4.72 in First multiplication 3.34 in second)over 8-core.

Problem in memory allocation

Problem in memory allocation

- Memory allocation done using `malloc()`.

Problem in memory allocation

- Memory allocation done using `malloc()`.
- `malloc()` maintains a single heap.

Problem in memory allocation

- Memory allocation done using `malloc()`.
- `malloc()` maintains a single heap.
- Threads wait to access memory: Benefit of parallelization lost.

Problem in memory allocation

- Memory allocation done using `malloc()`.
- `malloc()` maintains a single heap.
- Threads wait to access memory: Benefit of parallelization lost.
- Allocation of memory done beforehand.

Using lower level functions

Using lower level functions

- Initially `mpz_`prefixed functions in GNU/MP were used.

Using lower level functions

- Initially `mpz_`prefixed functions in GNU/MP were used.
- The routines are higher level and allocates memory in run-time: Must be avoided.

Using lower level functions

- Initially `mpz_`prefixed functions in GNU/MP were used.
- The routines are higher level and allocates memory in run-time: Must be avoided.
- Lower level function with prefix `mpn_` in GNU/MP are used.

Using lower level functions

- Initially `mpz_`prefixed functions in GNU/MP were used.
- The routines are higher level and allocates memory in run-time: Must be avoided.
- Lower level function with prefix `mpn_` in GNU/MP are used.
- Functions with prefix `mpn_` handles only limbs: Multi-precision integers are stored in array of limbs.

Avoiding copy-vector

Avoiding copy-vector

- Copy-vector is memory intensive.

Avoiding copy-vector

- Copy-vector is memory intensive.
- Lot of cache misses : performs poorly in multi-core platform.

Avoiding copy-vector

- Copy-vector is memory intensive.
- Lot of cache misses : performs poorly in multi-core platform.
- Copy-vectors are replaced by simple pointer exchanges only when possible.

Levinson-Durbin Method in multi-core

Levinson-Durbin Method in multi-core

- Mostly simple vector arithmetics are handled.

Levinson-Durbin Method in multi-core

- Mostly simple vector arithmetics are handled.
- Vectors are dense: Trivial parallelization should work.

Levinson-Durbin Method in multi-core

- Mostly simple vector arithmetics are handled.
- Vectors are dense: Trivial parallelization should work.
- In vector-vector multiplication loop critical section exists to update the shared variable: Degrades performances.

Levinson-Durbin Method in multi-core

- Mostly simple vector arithmetics are handled.
- Vectors are dense: Trivial parallelization should work.
- In vector-vector multiplication loop critical section exists to update the shared variable: Degrades performances.
- Critical Section is avoided:

Levinson-Durbin Method in multi-core

- Mostly simple vector arithmetics are handled.
- Vectors are dense: Trivial parallelization should work.
- In vector-vector multiplication loop critical section exists to update the shared variable: Degrades performances.
- Critical Section is avoided:
  - Each thread stores its computed value in a private variable.

Levinson-Durbin Method in multi-core

- Mostly simple vector arithmetics are handled.
- Vectors are dense: Trivial parallelization should work.
- In vector-vector multiplication loop critical section exists to update the shared variable: Degrades performances.
- Critical Section is avoided:
  - Each thread stores its computed value in a private variable.
  - After a thread finishes its computation, the shared variable is updated.

Berlekamp-Massey Method in multi-core

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.
- Main difference lies in polynomial multiplication.

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.
- Main difference lies in polynomial multiplication.
- Critical Section exists in the nested loop.

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.
- Main difference lies in polynomial multiplication.
- Critical Section exists in the nested loop.
- The technique to avoid critical section by declaring private polynomial does not work :

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.
- Main difference lies in polynomial multiplication.
- Critical Section exists in the nested loop.
- The technique to avoid critical section by declaring private polynomial does not work :
  - This technique involves adding up the private polynomials after the threads are done.

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.
- Main difference lies in polynomial multiplication.
- Critical Section exists in the nested loop.
- The technique to avoid critical section by declaring private polynomial does not work :
  - This technique involves adding up the private polynomials after the threads are done.
  - The number of costly polynomial addition increases with number of threads.

Berlekamp-Massey Method in multi-core

- Mostly simple polynomial arithmetics are handled.
- Polynomials are dense: Trivial technique should work.
- Main difference lies in polynomial multiplication.
- Critical Section exists in the nested loop.
- The technique to avoid critical section by declaring private polynomial does not work :
  - This technique involves adding up the private polynomials after the threads are done.
  - The number of costly polynomial addition increases with number of threads.
  - It is kept un-parallelized.

Experimental Set Up

Experimental Set Up

- The experiments were carried out on an *Intel*® *Xeon*® *E5410* dual-socket quad-core Linux server.

Experimental Set Up

- 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.

Experimental Set Up

- 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.

Experimental Set Up

- 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 integers are handled using *GNU/MP* [10] (version 4.3.1).

Execution Time in Multi-core (1st step)

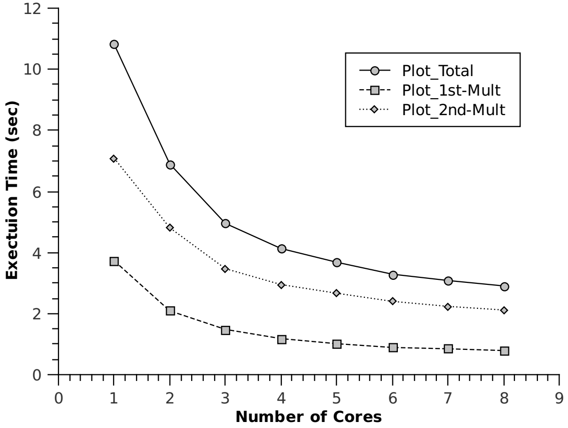

Figure: Execution Time using different numbers of cores (1st Step)

Speed-Up in Multi-core (1st Step)

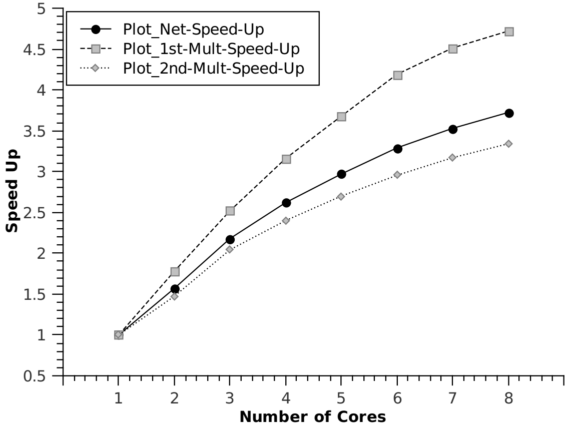

Figure: Speed-Up using different numbers of cores (1st Step)

Execution Time in LVD (2nd step)

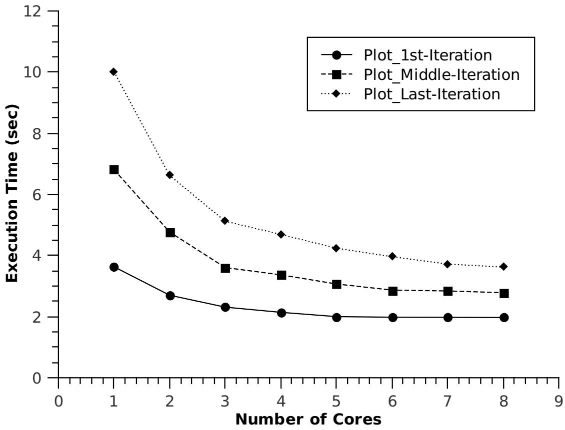

Figure: Execution time vs Number of Cores in different iterations

Execution Time in BKM (2nd step)

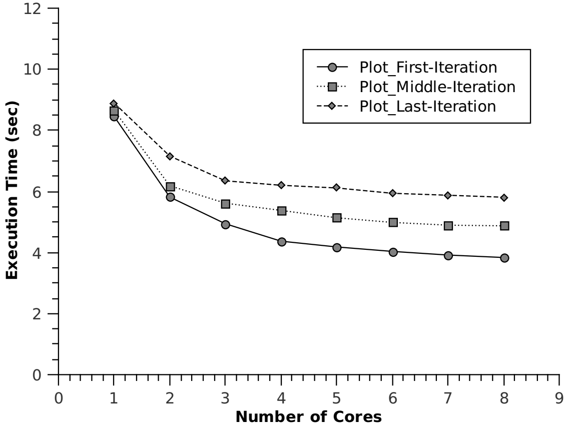

Figure: Execution time vs Number of Cores in different iterations

Comparative Exec Time of BKM and LVD (2nd Step)

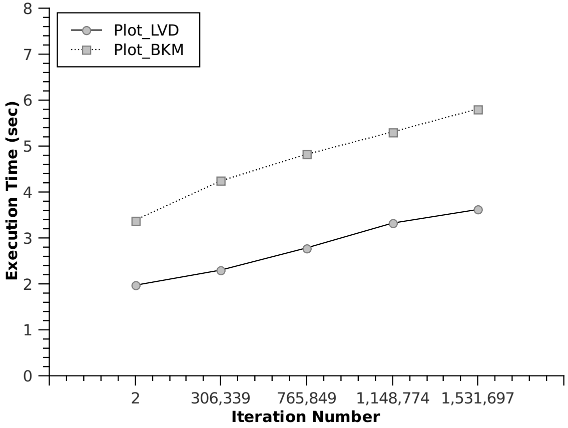

Figure: Comparison of Exec time in different iterations over 8-core

Comparative Speed Ups of BKM and LVD (2nd Step)

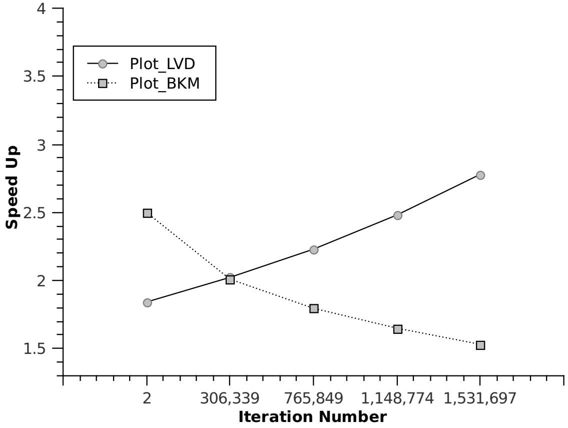

Figure: Comparison of Speed-Ups in different iterations over 8-core

Analysis (1st Step)

Analysis (1st Step)

- Execution times:

Analysis (1st Step)

- Execution times:
  - Sequential: First mult: 3.73 sec Second Mult: 7.09 sec

Analysis (1st Step)

- Execution times:
  - Sequential: First mult: 3.73 sec Second Mult: 7.09 sec
  - 8-core: First mult: 0.79 sec Second Mult: 2.12 sec

Analysis (1st Step)

- Execution times:
  - Sequential: First mult: 3.73 sec Second Mult: 7.09 sec
  - 8-core: First mult: 0.79 sec Second Mult: 2.12 sec
- Net Speed-up obtained: 3.72 (4.72 in First multiplication 3.34 in Second) over 8-core.

Analysis (1st Step)

- Execution times:
  - Sequential: First mult: 3.73 sec Second Mult: 7.09 sec
  - 8-core: First mult: 0.79 sec Second Mult: 2.12 sec
- Net Speed-up obtained: 3.72 (4.72 in First multiplication 3.34 in Second) over 8-core.
- First multiplication is more suitable for parallelization as rows of B are balanced but columns are not.

Analysis (1st Step)

- Execution times:
  - Sequential: First mult: 3.73 sec Second Mult: 7.09 sec
  - 8-core: First mult: 0.79 sec Second Mult: 2.12 sec
- Net Speed-up obtained: 3.72 (4.72 in First multiplication 3.34 in Second) over 8-core.
- First multiplication is more suitable for parallelization as rows of B are balanced but columns are not.
- Size of vector (1.6*mil*) in First mult is much smaller than the Second (2.2*mil*): Increases cache misses heavily in the second one.

Analysis (LVD in 2nd Step)

Analysis (LVD in 2nd Step)

- LVD is more parallelizable in the later iterations than earlier iterations.

Analysis (LVD in 2nd Step)

- LVD is more parallelizable in the later iterations than earlier iterations.
- Earlier iterations the size of vectors are smaller: The chunks handled by each thread are smaller.

Analysis (LVD in 2nd Step)

- LVD is more parallelizable in the later iterations than earlier iterations.
- Earlier iterations the size of vectors are smaller: The chunks handled by each thread are smaller.
- Loop overheads are more pronounced : Results degradation in performances.

Analysis (LVD in 2nd Step)

- LVD is more parallelizable in the later iterations than earlier iterations.
- Earlier iterations the size of vectors are smaller: The chunks handled by each thread are smaller.
- Loop overheads are more pronounced : Results degradation in performances.
- Parallelization works much better in later iterations which takes more time sequentially: Handle larger chunks.

Analysis (LVD in 2nd Step)

- LVD is more parallelizable in the later iterations than earlier iterations.
- Earlier iterations the size of vectors are smaller: The chunks handled by each thread are smaller.
- Loop overheads are more pronounced : Results degradation in performances.
- Parallelization works much better in later iterations which takes more time sequentially: Handle larger chunks.
- The average can be empirically given by middle-most iteration.

Analysis (BKM in 2nd step)

Analysis (BKM in 2nd step)

- BKM shows some parallelization in earlier iterations.

Analysis (BKM in 2nd step)

- BKM shows some parallelization in earlier iterations.
- Unlike sequential case, the execution time varies significantly.(3.35 sec to 5.8 sec)

Analysis (BKM in 2nd step)

- BKM shows some parallelization in earlier iterations.
- Unlike sequential case, the execution time varies significantly.(3.35 sec to 5.8 sec)
- The polynomial multiplication takes 5% of total time(in 8-core) in earlier iterations, 25% in later iterations.

Analysis (BKM in 2nd step)

- BKM shows some parallelization in earlier iterations.
- Unlike sequential case, the execution time varies significantly.(3.35 sec to 5.8 sec)
- The polynomial multiplication takes 5% of total time(in 8-core) in earlier iterations, 25% in later iterations.
- The speed-up decreases due to increase of the sequential part.

Comparative analysis of LVD and BKM

Comparative analysis of LVD and BKM

- In most (almost 3/4-th) of the iterations the speed-up in LVD is found better than in BKM.

Comparative analysis of LVD and BKM

- In most (almost 3/4-th) of the iterations the speed-up in LVD is found better than in BKM.
- The execution time taken by LVD is lesser than BKM in all the iterations over 8-core.

Comparative analysis of LVD and BKM

- In most (almost 3/4-th) of the iterations the speed-up in LVD is found better than in BKM.
- The execution time taken by LVD is lesser than BKM in all the iterations over 8-core.
- LVD is easier to parallelize as our techniques worked.

Comparative analysis of LVD and BKM

- In most (almost 3/4-th) of the iterations the speed-up in LVD is found better than in BKM.
- The execution time taken by LVD is lesser than BKM in all the iterations over 8-core.
- LVD is easier to parallelize as our techniques worked.
- Conclusion: LVD is found to perform better in multi-core scenario.

Lower speed-up in Second Step

Lower speed-up in Second Step

- Maximum speed-up: Last iteration of LVD : 2.81 , First iteration of BKM : 2.5

Lower speed-up in Second Step

- Maximum speed-up: Last iteration of LVD : 2.81 , First iteration of BKM : 2.5
- The vectors/polynomials active in this step are dense: No load-balancing issue.

Lower speed-up in Second Step

- Maximum speed-up: Last iteration of LVD : 2.81 , First iteration of BKM : 2.5
- The vectors/polynomials active in this step are dense: No load-balancing issue.
- Memory intensive operations are dominant: Results in huge number of cache misses.

Outline

Scope of improvements

Scope of improvements

- Some better load balancing technique like in [3] can be used.

Scope of improvements

- Some better load balancing technique like in [3] can be used.
- Process level parallelism can be implemented.

Scope of improvements

- Some better load balancing technique like in [3] can be used.
- Process level parallelism can be implemented.
- The Berlekamp-Massey implementation can be improved by using faster multiplication algorithm (*e.g.* FFT)

Scope of improvements

- Some better load balancing technique like in [3] can be used.
- Process level parallelism can be implemented.
- The Berlekamp-Massey implementation can be improved by using faster multiplication algorithm (*e.g.* FFT)
- Some good technique to make the memory-intensive routines faster should be invented: Some architecture-level analysis is needed.

Conclusion

Conclusion

- The superiority of LVD over BKM in both sequential and parallel scenario seems to remain valid after the improvements

Conclusion

- The superiority of LVD over BKM in both sequential and parallel scenario seems to remain valid after the improvements
  - Most of the routines are similar for both BKM and LVD.

Conclusion

- The superiority of LVD over BKM in both sequential and parallel scenario seems to remain valid after the improvements
  - Most of the routines are similar for both BKM and LVD.
  - The multiplication routine is different but it consumes less than 15% of total time in sequential.

Conclusion

- The superiority of LVD over BKM in both sequential and parallel scenario seems to remain valid after the improvements
  - Most of the routines are similar for both BKM and LVD.
  - The multiplication routine is different but it consumes less than 15% of total time in sequential.
- Our endeavour initiated the process to exploit Wiedemann Algorithm in multi-core platform.

Conclusion

- The superiority of LVD over BKM in both sequential and parallel scenario seems to remain valid after the improvements
  - Most of the routines are similar for both BKM and LVD.
  - The multiplication routine is different but it consumes less than 15% of total time in sequential.
- Our endeavour initiated the process to exploit Wiedemann Algorithm in multi-core platform.
- Applying more sophisticated techniques will definitely lead to better implementation in future.

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

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.

Souvik Bhattacherjee and Abhijit Das.
Parallelization of the lanczos algorithm on multi-core platforms.
In *ICDCN*, pages 231–241, 2010.

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

Don Coppersmith.

Solving homogeneous linear equations over gf(2) via block wiedemann algorithm.

Math. Comput., 62(205):333–350, 1994.

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.

Whitfield Diffie and Martin E. Hellman.

New directions in cryptography.

IEEE Transactions on Information Theory, 22(6):644–654, November 1976.

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.

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.

GNU.

The GNU MP Bignum Library.

http://gmplib.org/.

Daniel M. Gordon.

Discrete logarithms in GF(p) using the number field sieve.

SIAM J. Discrete Math., 6(1):124–138, 1993.

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

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

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

Arjen K. Lenstra, Hendrik W. Lenstra Jr., Mark S. Manasse, and John M. Pollard.

The number field sieve.

In *STOC*, pages 564–572. ACM, 1990.

Douglas H. Wiedemann.

Solving sparse linear equations over finite fields.

IEEE Transactions on Information Theory, 32(1):54–62, 1986.

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.

Thank You