Solving System of Linear Equations

CSE 633 Course Project (Spring 2014)

Avinash Paruchuri (50097849)

Vinod Galla (50096982)

Jagadeesh Vallabhaneni (50096437)

Equation Form

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
a_{i1}x_1 & + & a_{i2}x_2 & + & \cdots & + & a_{in}x_n & = & b_i \ . \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n
\end{array}$$

$$Ax = b$$

where $A \in R^{n \times n}$ is an $n \times n$ matrix of real numbers, $b \in R^n$ is a vector of size n, and $x \in R^n$ is an unknown solution vector of size n.

Solution exists only when there exists a matrix A^{-1} such that $A \cdot A^{-1} = I$

Methods of Implementation

- Direct Method – *Gaussian Elimination*

  Exact solution except rounding errors

- Iterative Method – *Jacobi Method*

  Determination of an approximate solution than exact one

Gaussian Elimination

Gaussian Elimination

- Forward Elimination and Backward Substitution
- Upper Triangular form

$$A^{(k)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ \vdots & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}.$$

Parallel Implementation Details

- Block distribution
  - Processor with rank 0 scatters data
  - n rows, p processors – each processor has n/p rows

- Forward elimination
  - For first processor,

    For first row
      - Make pivot element 1 by doing row operations
      - For lower ranked rows in the same processor, make the corresponding elements 0 by row operations

    Do this for other rows one by one
  - Send the data to other processors ranked higher

  Do this for other processors (from low to high) one by one

- Backward substitution
  - The matrix is now in upper triangular form (i.e all the elements before pivot element in each row are zero and pivot element is 1)
  - From last to first processor,

    For last to first row
      - For higher ranked rows in same processor, make the elements in the same column 0 by doing row operations

    Do this for other rows from bottom to top
  - Send the data to other processors ranked lower

  Do this for other processors (from high to low) one by one

- Processor with rank 0 gathers the data from all other processors which is the output vector

Parallel Implementation using Example

$$\begin{bmatrix} 2 & 2 & 4 & 2 \\ 2 & 3 & 3 & 2 \\ 1 & 2 & 3 & 3 \\ 1 & 2 & 3 & 5 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 3 \\ 2 \end{bmatrix}$$

$$Ax = b$$

Demonstrated in class

Jacobi Method

Jacobi Method

Given a square system of n linear equations:

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

where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Then A can be decomposed into a diagonal component D, and the remainder R:

$$A = D + R \qquad \text{where} \qquad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

The solution is then obtained iteratively via

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}).$$

The element-based formula is thus:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \ldots, n.$$

Calculating X

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

$$\boxed{\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \end{aligned}} \Leftarrow P_0$$

$$\vdots$$

$$\boxed{x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})} \Leftarrow P_{k-1}$$

Distribution at each Process

$$A = \begin{array}{cc} \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n-1} & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n-1} & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1\,1} & a_{n-1\,2} & a_{n-1\,3} & \cdots & a_{n-1\,n-1} & a_{n-1\,n} \\ a_{n\,1} & a_{n\,2} & a_{n\,3} & \cdots & a_{n\,n-1} & a_{n\,n} \end{pmatrix} & \begin{array}{l} \Rightarrow P_0 \\ \\ \Rightarrow P_1 \\ \\ \\ \Rightarrow P_{k-1} \end{array} \end{array}$$

$$\mathbf{b} = \begin{array}{cc} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} & \begin{array}{l} \Rightarrow P_0 \\ \\ \\ \Rightarrow P_{k-1} \end{array} \end{array} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Algorithm

Choose an initial X^{new} = b.
int iteration = 0;
int k = num of rows per process;
do {
    $X^{old} = X^{new}$;
    for i=1…k, do
        $\sigma = 0$;
        for j=1…n, do
            if (i != j), then
                $\sigma = \sigma + a_{ij}x_j^{old}$;
            end-if.
        end-for.
        $x_i^{new} = (b_i - \sigma)/a_{ii}$;
    end-for.
    MPI_Allgather(X^{new});
    iteration++;
} while((!converged) && (iteration < MAX_ITERATIONS));

Input

- A = Diagonally dominant square matrix

```
//      Generating random Diagonally Dominant matrix A  //
for (irow = 0; irow < n; irow++) {
        A[irow] = (double *) malloc(n*sizeof(double));
        for (jcol = 0; jcol < n; jcol++) {
                if (irow == jcol) {
                        A[irow][jcol] = (((double) rand())/((double) (RAND_MAX))) * (MAX_LIMIT * cols);
                } else {
                        A[irow][jcol] = (((double) rand())/((double) (RAND_MAX)));
                }
        }
}
```

MPI Functions Used

```
// Broadcast the number of rows and columns from root //

MPI_Bcast( &rows, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
MPI_Bcast( &cols, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
```

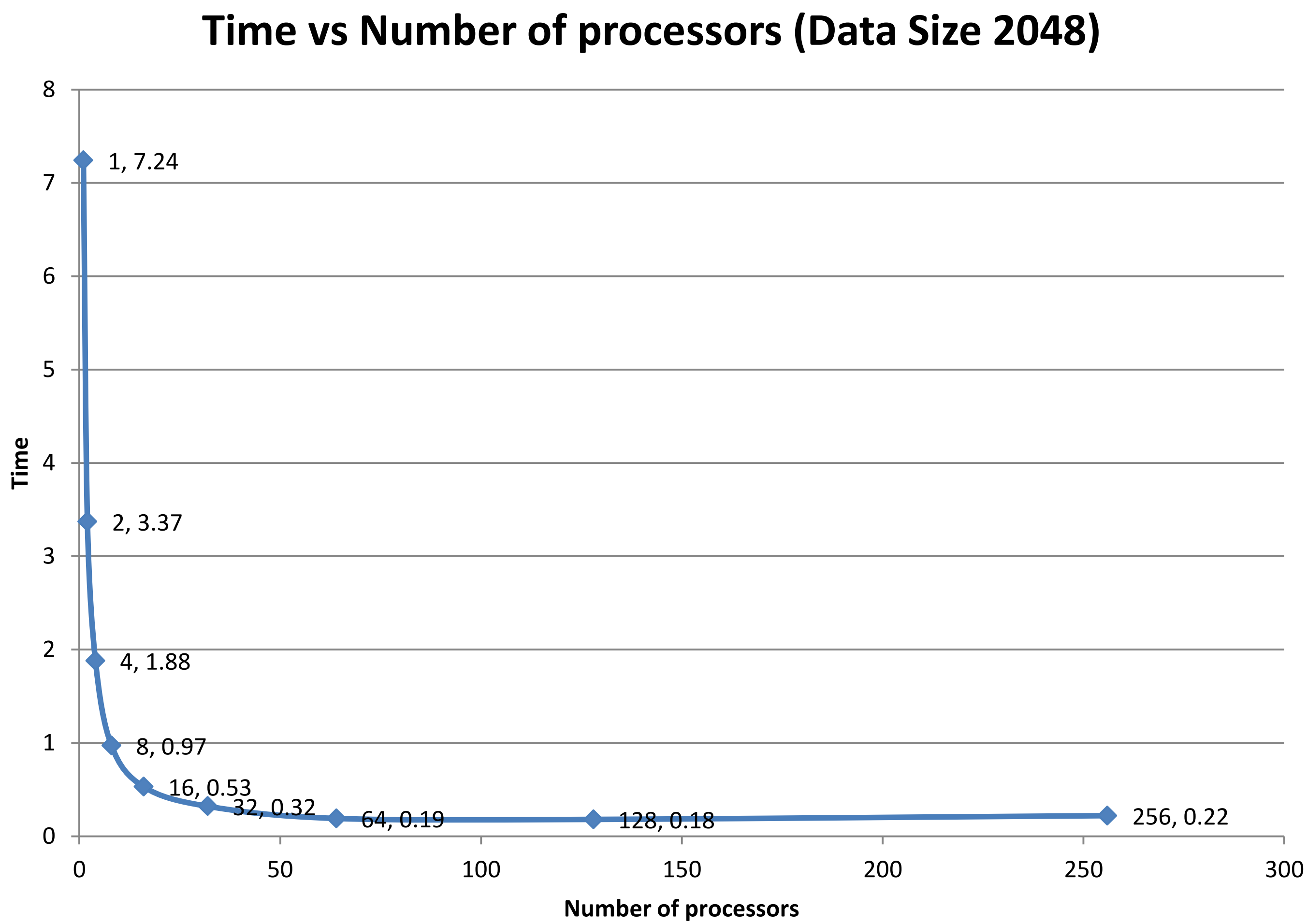
```
/* Scatter the input data from 1-D array to all the processes....
*  Each Processe captures its share of the data
*
*/

MPI_Scatter(arrayA, rows_process*n, MPI_DOUBLE,
            ARecv,  rows_process*n, MPI_DOUBLE,
            ROOT, MPI_COMM_WORLD);

MPI_Scatter(B,     rows_process, MPI_DOUBLE,
            BRecv, rows_process, MPI_DOUBLE,
            ROOT, MPI_COMM_WORLD);
```

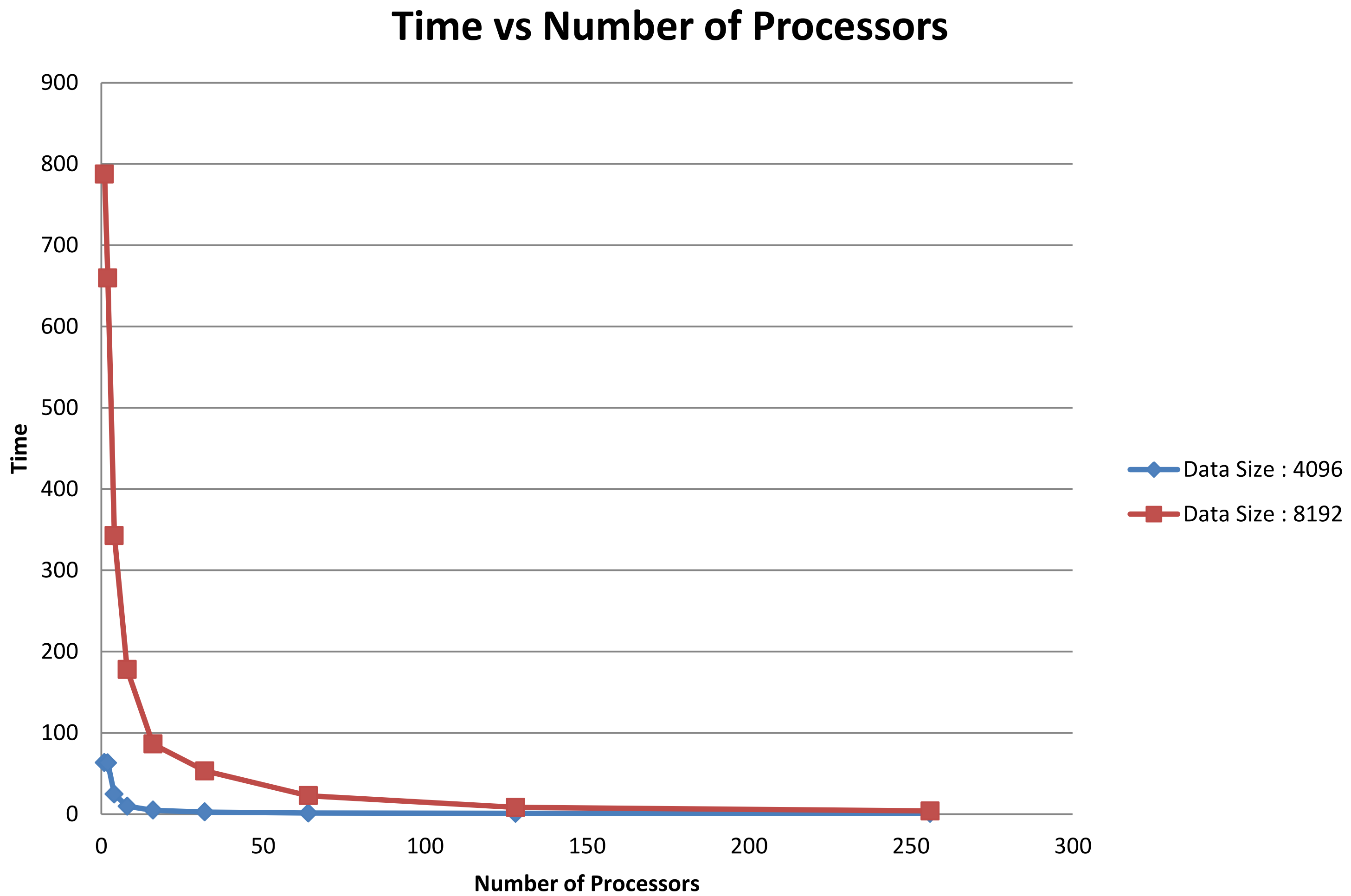
```
// Collecting all the values of x from 1...n into X_new //
MPI_Allgather(process_X, rows_process, MPI_DOUBLE,
              X_new,     rows_process, MPI_DOUBLE,
              MPI_COMM_WORLD);
```

Results – Gaussian Elimination

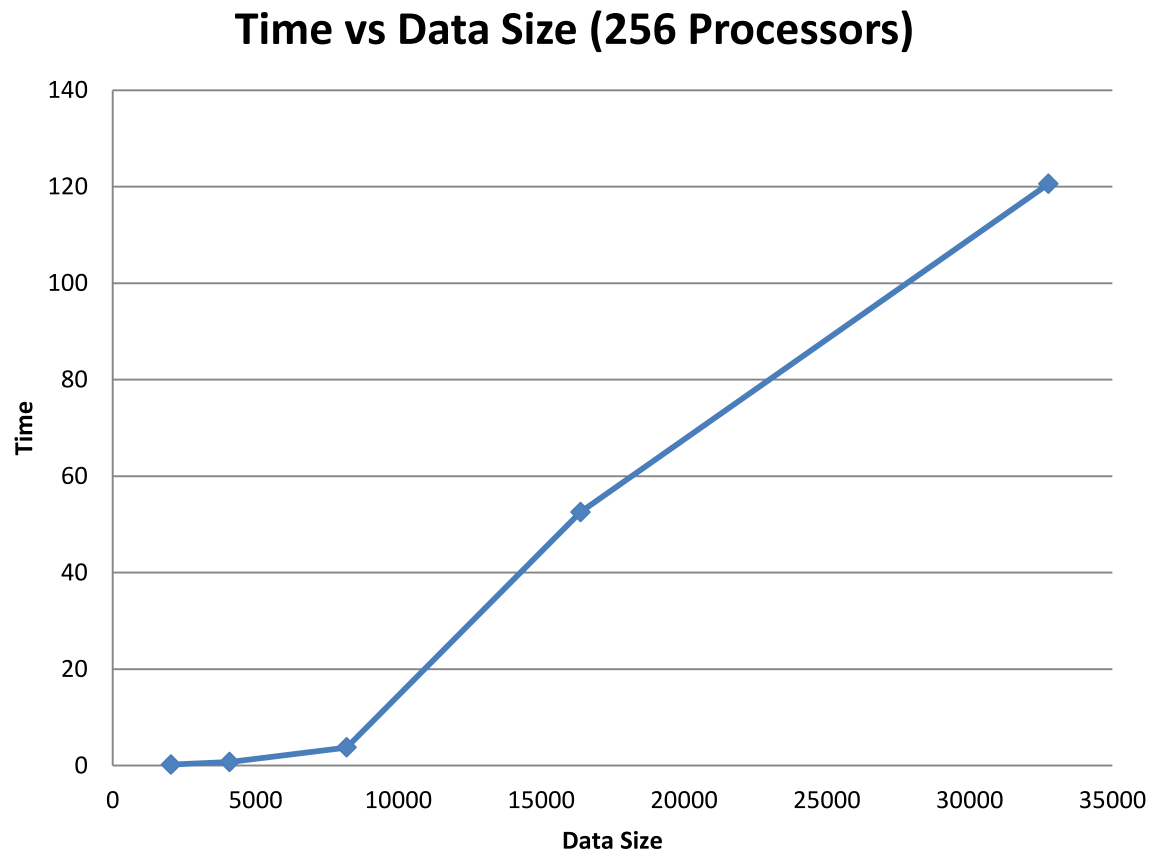


#processers	time taken
1	7.24
2	3.37
4	1.88
8	0.97
16	0.53
32	0.32
64	0.19
128	0.18
256	0.22

Results – Gaussian Elimination

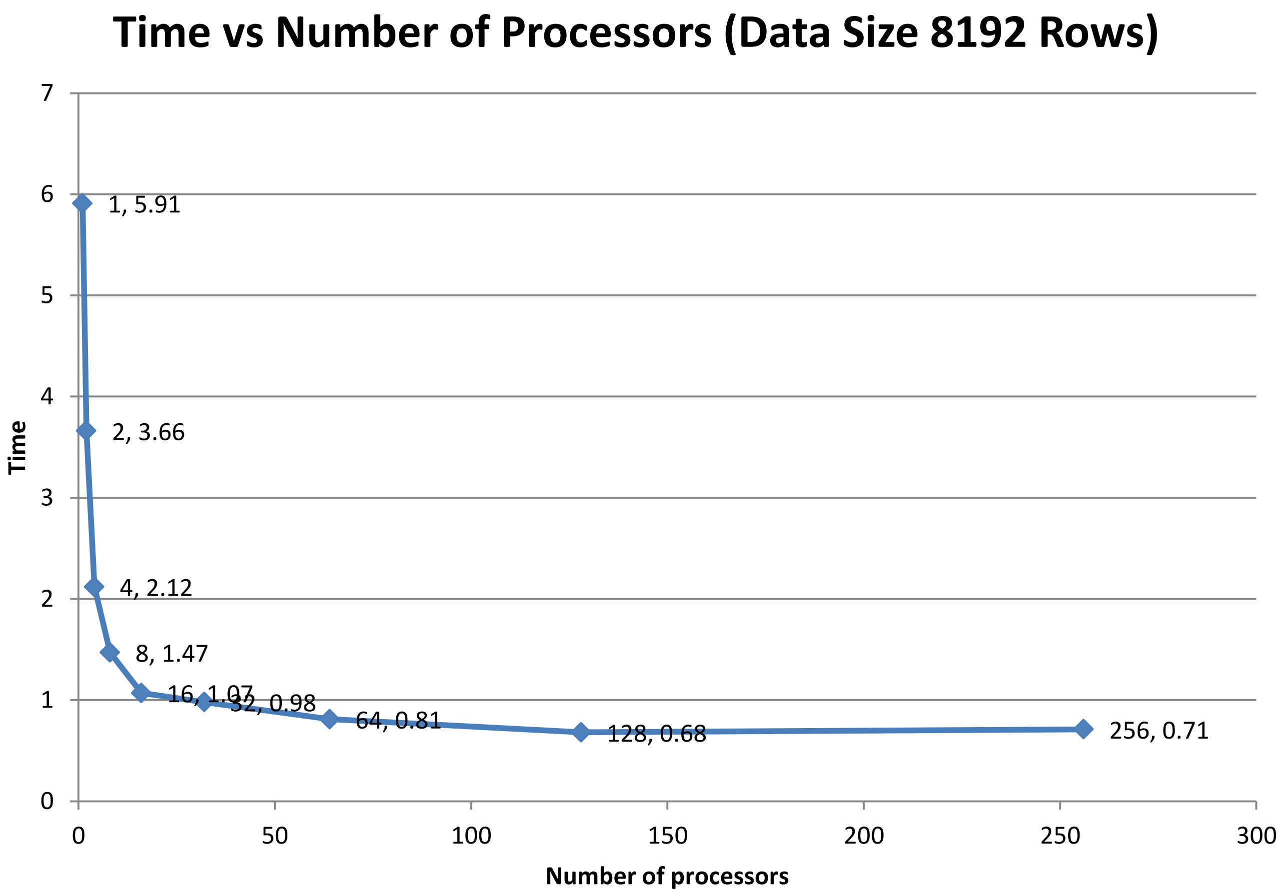


Results – Gaussian Elimination



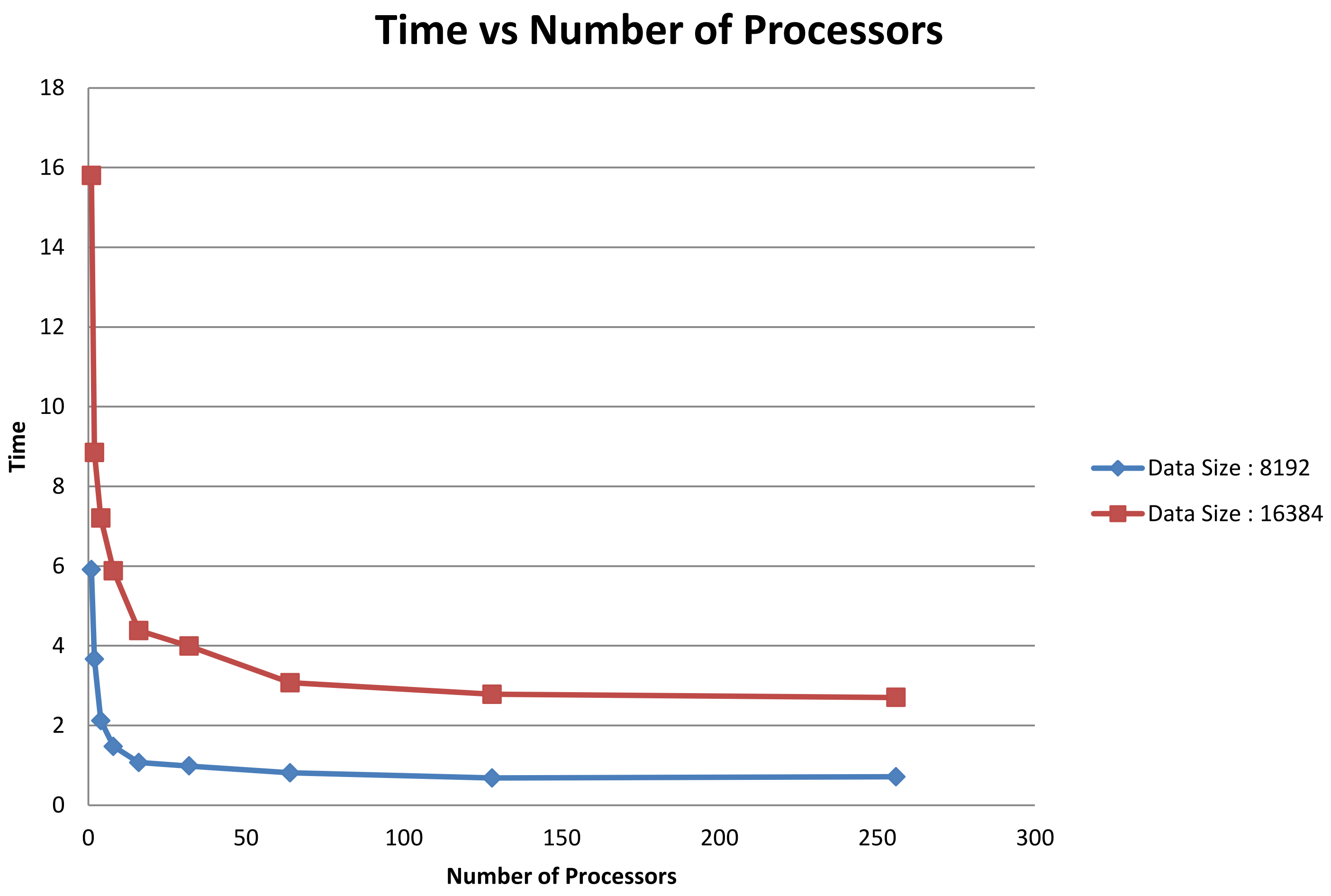
Data size (Number of Row)	Time Taken
2048	0.22
4096	0.73
8192	3.76
16384	52.53
32768	120.6

Results – Jacobi Method

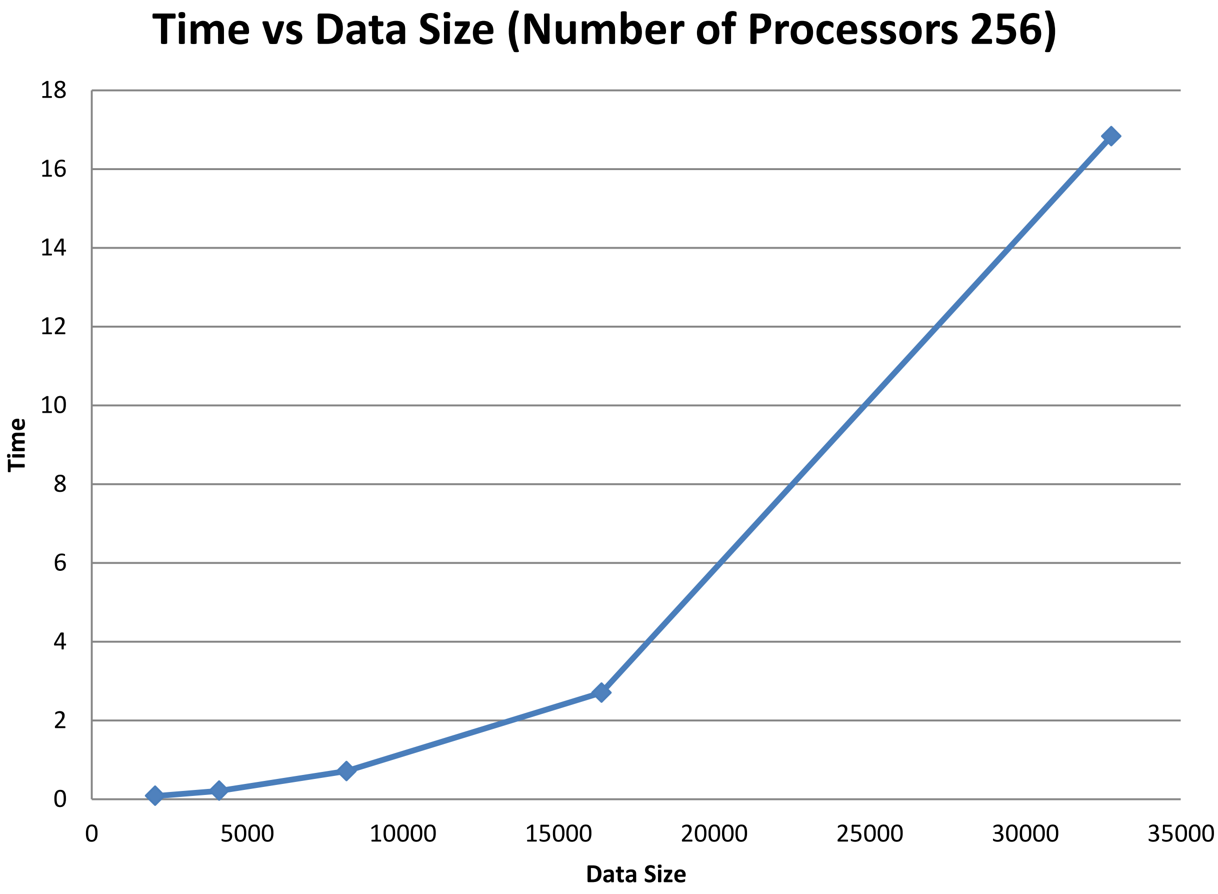


Number of Processors	Time
1	5.91
2	3.66
4	2.12
8	1.47
16	1.07
32	0.98
64	0.81
128	0.68
256	0.71

Results – Jacobi Method

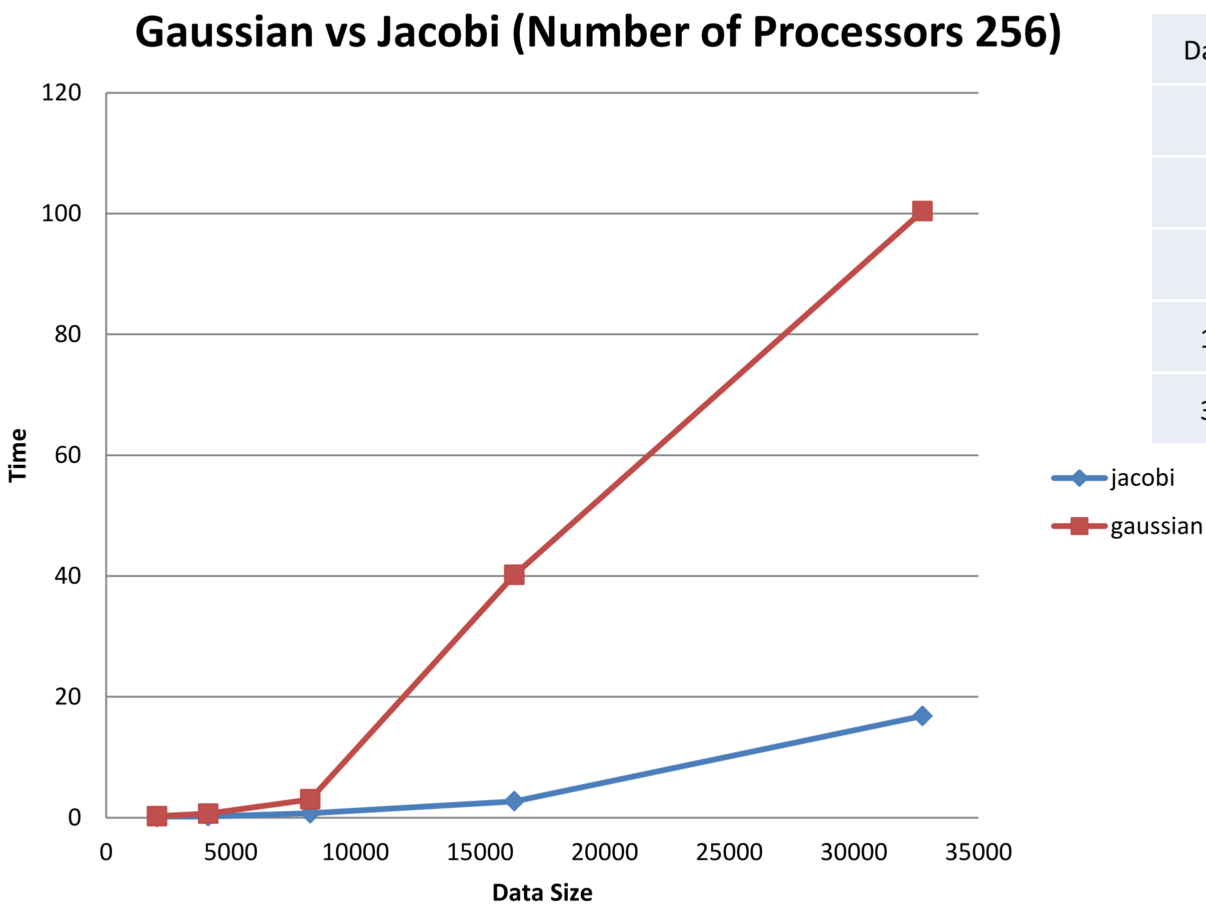


Results – Jacobi Method



Data Size	Time
2048	0.08
4096	0.21
8192	0.71
16384	2.7
32768	16.83

Results – Gaussian vs. Jacobi



Data Size	Jacobi	Gaussian
2048	0.08	0.23
4096	0.21	0.65
8192	0.71	3
16384	2.7	40.18
32768	16.83	100.4

Comparison between both methods for diagonally dominant data

Future Work

- Using Relaxation methods for Jacobi Method

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega) x_i^{(k)} \,, \quad i = 1, \ldots, n \,.$$

- Gauss – Seidel for large sparse matrices
- Use cyclic distribution in Gaussian Elimination

References

- Parallel Programming for Multicore and Cluster Systems (Thomas Rauber and Gudula Runger)
- Iterative Methods for Sparse Linear Systems, Second Edition (Yousef Saad)
- Message Passing Interface (MPI) forum http://www.mpi-forum.org
- MPI Tutorial http://mpitutorial.com/

Thank You