Performance Study on Preconditioned Conjugate Gradient Method Implemented in MPI and Stochastic Gradient Descent Method Implemented in Spark of Solving a System of Linear Equations

Jie Zhou

Student ID: 1399121

09/04/2016

Contents

List of Figures

1 Introduction

The preconditioned conjugate gradient (CG) method is generally used for solving large sparse systems of equations [1]. The linear system in the form of $Ax = b$ is the result of discretized partial differential equations using finite element method (FEM). For FEM applications, it is necessary to solve the system of equations in the form of stiff matrix and force vector. Getting a efficient solution of a linear system, where A is a large, sparse matrix, is an important and costly step. Although direct methods are robust and predictable techniques used to solve these types of linear system problems, they are costly in terms of computational time and memory requirements. Thus, the reason of using conjugate gradient method is because it is a fast iterative method which can efficiently estimate the exact solution of a system of equations without costing a lot of computational resources.

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications [2].

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function in the formulation given by the squared loss:

$$L(w, x, y) := \frac{1}{2}(w^T x - y)^2 \tag{1}$$

where the x corresponds to the matrix in the linear system and y corresponds to the right hand side vector. The w^T which is the weights in the regression problem will be the solution to the linear system. Various related regression methods are derived by using different types of regularization: ordinary least squares or linear least squares uses no regularization; ridge regression uses L2 regularization; and Lasso uses L1 regularization. For all of these models, the average loss or training error is known as the mean squared error. Stochastic gradient descent (often shortened in SGD) is a gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions.

In the report, a distributed implementation of preconditioned CG algorithm and SGD implementation in Spark are presented.

2 Implementation of preconditioned CG using MPI

2.1 Implementation CG using 1D partitioning

The classic conjugate gradient method is an iterative algorithm applied to solve systems of linear equations of the type $Ax = b$, where A is a sparse symmetric positive-definite matrix. However, in the real

application of solving coupled equations, the matrix A is not always symmetric. Therefore, it requires one to multiple the transpose of the initial matrix to the original matrix A before solving the problem shown in equation 2.

$$A^T Ax = A^T b \tag{2}$$

This method is known as the CGNR method. This step requires computing the transpose of a matrix, a matrix matrix multiplication and a matrix vector multiplication.

The implementation of getting the transpose of a matrix is shown in the following code:

```
1    double* x;
2    double* y;
3    int i,j;
4    x = malloc(n*n*sizeof(double));
5    y = malloc(n*n*sizeof(double));
6    MPI_Allgather(local_in, local_n*n, MPI_DOUBLE,
7                  x, local_n*n, MPI_DOUBLE, comm);
8    if (my_rank ==0)
9    {
10       for (i = 0; i < n; ++i){
11           for (j = 0; j< n; ++j){
12               y[i*n+j] = x[j*n+i];
13           }
14       }
15       MPI_Scatter(y, local_n*n, MPI_DOUBLE,
16                   local_out, local_n*n, MPI_DOUBLE, 0, comm);
17   } else {
18       MPI_Scatter(y, local_n*n, MPI_DOUBLE,
19                   local_out, local_n*n, MPI_DOUBLE, 0, comm);
20   }
```

The x and y are the two temporary arrays to store the origin matrix and its transpose. The *MPI_Allgather* is used to gather all local row matrices and put them into array x. In the rank zero, the transpose of the matrix y is computed, then distributed into each local matrix. In each local rank, the matrix is received by calling the *MPI_Scatter* again.

The matrix matrix multiplication can be represented $AB = C$. The implementation of the matrix matrix multiplication is shown in the following code:

```
1    double* x;
2    int local_i, j,k;
3    x = malloc(n*n*sizeof(double));
4    MPI_Allgather(local_B, local_n*n, MPI_DOUBLE,
5          x, local_n*n, MPI_DOUBLE, comm);
6    for (local_i = 0; local_i < local_n; local_i++) {
```

```
7         for (k = 0; k < n; k++){
8             local_C[local_i*n + k] = 0;
9             for (j = 0; j <n; j++)
10            {
11                local_C[local_i*n + k] += local_A[local_i*n+j]*x[j*n+k];
12    }}}
```

The array x is also a temporary array which is used to store the global matrix B. Then, the local matrix C is computed by multiplying the row-wise local matrix A with the global matrix B.

The matrix vector multiplication can be represented in the form of $Ax = y$. The implementation of the matrix vector multiplication is shown in the following code:

```
1     double* x;
2     int local_i, j;
3     x = malloc(n*sizeof(double));
4     MPI_Allgather(local_x, local_n, MPI_DOUBLE,
5           x, local_n, MPI_DOUBLE, comm);
6     for (local_i = 0; local_i < local_m; local_i++) {
7        local_y[local_i] = 0.0;
8        for (j = 0; j < n; j++)
9           local_y[local_i] += local_A[local_i*n+j]*x[j];
10    }
```

The array x is used to store the entire vector x temporarily. The row-wise decomposed local matrix A is multiplied by the global array x. The result array y is the local component of the vector y.

After applying the CGNR method to the numerical system. The new matrix $D = A^T A$ satisfies the requirement of symmetric positive-definite. The new right hand side (RHS) is the result of matrix vector multiplication of $A^T b$.

The preconditioned algorithm is shown in the figure 1. The convergence rate of CG method depends on the properties of the matrix D. If a matrix M approximates the input coefficient matrix D in some way, the transformed system $M^{-1}Ax = M^{-1}b$ has the same solution as the original system $Ax = b$, however the spectral properties of its coefficient matrix $M^{-1}A$ may be more favourable. The choice of M is the diagonal element of the matrix D in $Dx = A^T b$. The MPI implementation of getting the vector M is:

```
1     int i;
2     for (i = 0; i < local_m; ++i) {
3         M[i] += Adata[n*i + (i+my_rank*local_m)];
4     }
```

where each local array *Adata* is the row-wise decomposed global matrix D. The local matrix *Adata* is passing the corresponding global diagonal element into array M depending on the rank level.

Preconditioned Conjugate Gradient method

$k = 0$; $x_0 = 0$; $r_0 = b$; initialization

while $(r_k \neq 0)$ do termination criterion

$z_k = M^{-1} r_k$ preconditioning

$k := k + 1$

if $k = 1$ do

$p_1 = z_0$

else

$\beta_k = \frac{r_{k-1}^T z_{k-1}}{r_{k-2}^T z_{k-2}}$ update of p_k

$p_k = z_{k-1} + \beta_k p_{k-1}$

end if

$\alpha_k = \frac{r_{k-1}^T z_{k-1}}{p_k^T A p_k}$

$x_k = x_{k-1} + \alpha_k p_k$ update iterate

$r_k = r_{k-1} - \alpha_k A p_k$ update residual

end while

Figure 1: Preconditioned Conjugate Gradient Algorithm

There are three vectors p (searching direction), r (residual vector) and z (preconditioned residual vector) in the iteration process. Following the algorithm is figure 1, z is computed by doing $M^{-1}r$ which is done by the following code:

```
1    int i;
2    for(i=0; i<n; i++){
3        Minvx[i] = 1/Mdata[i]*x[i];
4    }
```

As the calculation is done locally, this part is being embarrassingly parallelized.

The updating α in k^{th} iteration involves a dot product which has been parallelized by the following code:

```
1    int i;
2    double local_sum = 0;
3    double final_sum = 0;
4    for (i = 0; i < n; ++i)
5        local_sum += x[i] * y[i];
6    MPI_Allreduce(&local_sum,&final_sum,1,MPI_DOUBLE,MPI_SUM,comm);
7    return final_sum;
```

The $final_sum$ is the total summation of each local summation of the dot product of two local vectors.

Then, the iterative solution x is updated by doing the vector addition with x from last iteration and the multiplication of α and p in the current iteration. The vector operation involved in this case can be embarrassingly parallelized shown as the following code:

```
1     int i;
2     for (i = 0; i < n; ++i)
3         dest[i] = a * x[i] + y[i];
```

The overall implementation of the CG method is attached in the appendix which includes the functions of reading the matrix A and RHS vector b from text files and print matrix and vector functions to help one to debug the code.

The convergence criterion is set to be as the following:

$$||r||_2 <= tolerance \tag{3}$$

where the tolerance is set to be 10^{-8} in this CG solver.

2.2 Implementation of Matrix vector multiplication in 2D

A lot of effort has been made to tackle the CG method using 2D partitioning of matrix matrix multiplication, matrix vector multiplication and matrix transpose. The following code demonstrate the way of loading the matrix from the file and distribute the matrix into block wise piece. The key part is to construct the *MPI_Datatype* for each dimension (row and column). After constructing the *MPI_Datatype*, the data needs to be stored in a *MPI_Type_vector* in order to be scattered in the code. The for loop for *disps* and *counts* are for the dimensions to be used in the *MPI_Scatterv*.

```
1  void Loadinput_Matrix(int *n, int *local_n, double **local_A,int my_rank,int comm_size,
       MPI_Comm comm)
2  {
3      FILE* ip;
4      int i,j;
5      double* A = NULL;
6      int npes;
7      MPI_Comm_size(comm, &npes);
8      int ROW=0, COL=1;
9      int dims[2];
10     dims[ROW]= dims[COL] = sqrt(npes);
11     if (my_rank == 0) {
12         if ((ip = fopen("data_input_matrix","r")) == NULL){
13             printf("error opening the input data.\n");
14         }
15         fscanf(ip, "%d\n", n);
```

```
16
17          A = malloc((*n)*(*n)*sizeof(double));
18          for (i = 0; i < *n; ++i){
19              for (j = 0; j< *n; ++j){
20                  fscanf(ip, "%lf\t", &A[i*(*n)+j]);
21              }
22          }
23          fclose(ip);
24      }
25      MPI_Bcast(n, 1, MPI_INT, 0, comm);
26      *local_n = *n/dims[ROW];
27      *local_A = malloc((*local_n)*(*local_n)*sizeof(double));
28
29      int ROWS = *n;
30      int COLS = *n;
31      const int NPROWS=*local_n;  /* number of rows in _decomposition_ */
32      const int NPCOLS=*local_n;  /* number of cols in _decomposition_ */
33      const int BLOCKROWS = ROWS/NPROWS;  /* number of rows in _block_ */
34      const int BLOCKCOLS = COLS/NPCOLS; /* number of cols in _block_ */
35      MPI_Datatype blocktype;
36      MPI_Datatype blocktype2;
37
38      MPI_Type_vector(BLOCKROWS, BLOCKCOLS, COLS, MPI_DOUBLE, &blocktype2);
39      MPI_Type_create_resized( blocktype2, 0, sizeof(double), &blocktype);
40      MPI_Type_commit(&blocktype);
41
42      int ii,jj;
43      int disps[NPROWS*NPCOLS];
44      int counts[NPROWS*NPCOLS];
45      for (ii=0; ii<NPROWS; ii++) {
46          for (jj=0; jj<NPCOLS; jj++) {
47              disps[ii*NPCOLS+jj] = ii*COLS*BLOCKROWS+jj*BLOCKCOLS;
48              counts [ii*NPCOLS+jj] = 1;
49          }
50      }
51      MPI_Scatterv(A, counts, disps, blocktype, *local_A, BLOCKROWS*BLOCKCOLS, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
52  }
```

The following code is used to load the RHS vector and distribute it into the first process in each row. The 2D Cartesian topology is constructed to separate the data and store them only in the first process in each row. The column communicator is used to pass the data.

```
1  void Loadinput_RHS_vector(int n, int local_n, double **local_b, int my_rank, int
       comm_size, MPI_Comm comm)
```

```
2 {
3     FILE* ip;
4     int i;
5     double* b = NULL;
6     int ROW=0, COL=1;
7     int nlocal;
8     int npes, dims[2], periods[2], keep_dims[2];
9     int myrank, my2drank, mycoords[2];
10    MPI_Comm comm_2d, comm_row,comm_col;
11    MPI_Comm_size(comm, &npes);
12    MPI_Comm_rank(comm, &myrank);
13    dims[ROW]= dims[COL] = sqrt(npes);
14    nlocal = n/dims[ROW];
15    *local_b = malloc(nlocal*sizeof(double));
16    periods[ROW] = periods[COL] = 1;
17    //Create a 2D Cartesian topology and get the rank and coordinates of the process
18    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
19    MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
20    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */
21    /* Create the row-based sub-topology*/
22    keep_dims[ROW] = 0;
23    keep_dims[COL] = 1; /* Column is still connected*/
24    MPI_Cart_sub(comm_2d, keep_dims, &comm_row); /*Each row get its own row communicator
          */
25    /* Create the column-based sub-topology*/
26    keep_dims[ROW] = 1;
27    keep_dims[COL] = 0;
28    MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
29    if (my2drank == 0) {
30        if ((ip = fopen("data_input_RHS","r")) == NULL){
31            printf("error opening the input data.\n");
32        }
33        b = malloc(n*sizeof(double));
34        for (i = 0; i < n; ++i){
35            fscanf(ip, "%lf\t", &b[i]);
36        }
37        fclose(ip);
38    }
39    if (my2drank==0)
40    {
41        MPI_Scatter(b, nlocal, MPI_DOUBLE,
42                    *local_b, nlocal, MPI_DOUBLE, 0, comm_col);
43    }
44    else if (mycoords[COL]==0)  {
```

```
45          MPI_Scatter(b, nlocal, MPI_DOUBLE,
46                      *local_b, nlocal, MPI_DOUBLE, 0, comm_col);
47      }
48  }
```

The matrix vector 2D multiplication implementation is shown in the following code. This is the updated version of the code as provided in the class. One thing needs to know is that the outputted vector is stored in the diagonal processes instead of the first process in each row.

```
1  void MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,MPI_Comm comm)
2  {
3      int ROW=0, COL=1; /* Improve readability */
4      int i, j, nlocal;
5      double *px; /* Will store partial dot products */
6      int npes, dims[2], periods[2], keep_dims[2];
7      int myrank, my2drank, mycoords[2];
8      int other_rank, coords[2];
9      MPI_Status status;
10     MPI_Comm comm_2d, comm_row, comm_col;
11     MPI_Comm_size(comm, &npes);
12     MPI_Comm_rank(comm, &myrank);
13     dims[ROW]= dims[COL] = sqrt(npes);
14     nlocal = n/dims[ROW];
15     px = malloc(nlocal*sizeof(double));
16     periods[ROW] = periods[COL] = 1;
17     //Create a 2D Cartesian topology and get the rank and coordinates of the process
18     MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
19     MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
20     MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */
21     /* Create the row-based sub-topology*/
22     keep_dims[ROW] = 0;
23     keep_dims[COL] = 1; /* Column is still connected*/
24     MPI_Cart_sub(comm_2d, keep_dims, &comm_row); /*Each row get its own row communicator
           */
25     /* Create the column-based sub-topology*/
26     keep_dims[ROW] = 1;
27     keep_dims[COL] = 0;
28     MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
29     /* the vector b is in the first column nlocal is the last column*/
30     if (mycoords[COL] == 0 && mycoords[ROW] != 0) {
31         coords[ROW] = mycoords[ROW]; /* coords[0] = mycoords[0], mycoords[0] = 1 */
32         coords[COL] = mycoords[ROW];
33         MPI_Cart_rank(comm_2d, coords, &other_rank); /* other rank is the place you need
               to send*/
34         MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
```

```
35      }
36      if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
37          coords[ROW] = mycoords[ROW];
38          coords[COL] = 0;
39          MPI_Cart_rank(comm_2d, coords, &other_rank);
40          MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d, &status);
41      }
42      coords[0] = mycoords[COL];
43      MPI_Cart_rank(comm_col, coords, &other_rank);
44      MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);
45
46      for (i=0; i<nlocal; i++) {
47          px[i] = 0.0;
48          for (j=0; j<nlocal; j++){
49              px[i] += a[i*nlocal+j]*b[j];
50          }
51      }
52      coords[0] = mycoords[ROW];; /*In the course coords[1] = 0; */
53      MPI_Cart_rank(comm_row, coords, &other_rank);
54      MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank, comm_row);
55      MPI_Comm_free(&comm_2d); /* Free up communicator */
56      MPI_Comm_free(&comm_row); /* Free up communicator */
57      MPI_Comm_free(&comm_col); /* Free up communicator */
58      free(px);
59  }
```

3 Implementation of SGD using python Spark

The implementation of SGD in Spark demonstrates how to load training data, parse it as an RDD of LabeledPoint in python script. The LinearRegressionWithSGD is used to build a simple linear model to predict label values. The outputs are the model weights and the mean squared error at the end to evaluate goodness of fit.

The first part of the code is to load the data from files. These files are generated using the random matrix generation function in Matlab. Then using the output function in Matlab, the files are saved with the names of $Matrix$ and RHS. After loading the data, the object of LabeledPoint need to be created to store both the x (Matrix) and y (RHS) described in equation 1.

```
1 from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD,
      LinearRegressionModel
2 import time
3 start_time = time.time()
4 Matrix = sc.textFile("Inputdata/Matrix")
```

```
RHS     = sc.textFile("Inputdata/RHS")
NewMatrix = Matrix.map(lambda line: line.split(' '))
x = NewMatrix.collect()
y = RHS.collect()
y = map(float,y)
for i in range(0,len(x)):
    x[i] = map(float,x[i])
a = []
for i in range(0,len(y)):
    a.append(LabeledPoint(y[i],x[i]))
```

After getting the LabeledPoint object, one needs to store it as an RDD in order to pass it to the SGD model in Spark. Then the model is built using the function provided in Spark and the weights are solved by minimizing the error. Depending on the complexity of the problem, a parametric study on the step size in SGD is needed to minimize the error.

```
results = sc.parallelize(a)
model = LinearRegressionWithSGD.train(results, iterations=100000, step=0.5)
```

The weights are saved in the output file and the mean squared error are outputted in the console. The model then is saved for the later use.

```
file = open('output.txt', 'w')
c = map(float,model.weights)
for item in c:
 file.write("%f\n" % item)
valuesAndPreds = results.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds.map(lambda (v, p): (v - p)**2).reduce(lambda x, y: x + y) /
    valuesAndPreds.count()
#Output the results
model.save(sc, "SGD")
print("---- %s seconds ----" % (time.time() - start_time))
```

4 Test and verification

The tests with CG using MPI are done by randomly generating a matrix and RHS vector using the Matlab and comparing the results with the Matlab. With a matrix size of 16, 256 and 512 the results are matching with the results solved with the same matrix and RHS vector in the Matlab.

The Spark SGD model is tested first with a very small problem size to make sure the correctness of the algorithm. Due to the different conditional number of the matrix (complexity of the problem), however, the step size plays an very important role to reduce the error. The Figure 2 shows that the mean

Testing and verification of SGD

PROBLEM SIZE	MEAN SQUARED ERROR
4	0.0003
40	0.0012
80	0.0019
100	0.0020
128	0.0360

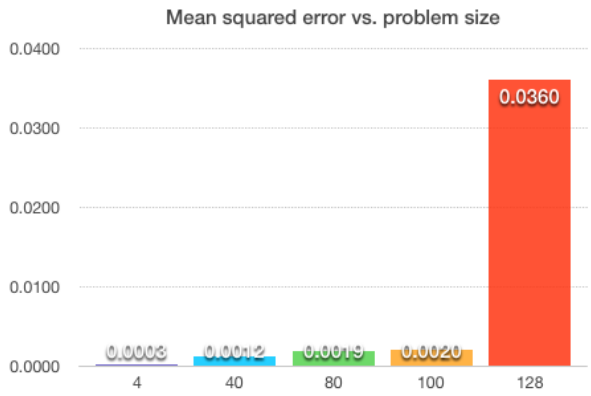

Figure 2: Problem size vs. Mean squared error using SGD in Spark

squared error is increasing with the growth of problem size using SGD method at a fixed conditional number of the matrix. The step size in this case is fixed at 0.5.

5 Performance of the code using MPI

5.1 Performance of 1D CG MPI implementation

The theoretical computational time of matrix and matrix multiplication in parallel is:

$$T_P = \frac{n^3}{p} + t_s log p + 2t_w \frac{n^2}{\sqrt{p}} \tag{4}$$

It is optimal for $p = \mathcal{O}(n^2)$. The total computational time also involves the time taken by data transfer and communication time between processes. The graphical results in Figure 3 and 4 below give insights onto the execution time versus number of processes for a data size of 2048 and 4096. The running time is optimized with the number of processes to be 8. When the number of processes is over 8, the amount of overhead time increases slightly. The running time of 1 process is the time running of a serial CG implementation. From the figure 3 and 4, one can conclude that with the use of the MPI, the total amount of the running time with the CG solver is reduced by approximately 3 times.

Figure 5 and 6 show the comparison of the speedup and efficiency between two different problem size. The figure 5 illustrate with the increase the problem size, the speedup of the using MPI is decreasing. One of the reasons for this is because of the overhead associated with the data communications. It is clear that for both of the problem size, the efficiency is decreasing with the increase of the number of processes. With small number of the processes, the efficiency is higher due to the smaller amount of the communication time.

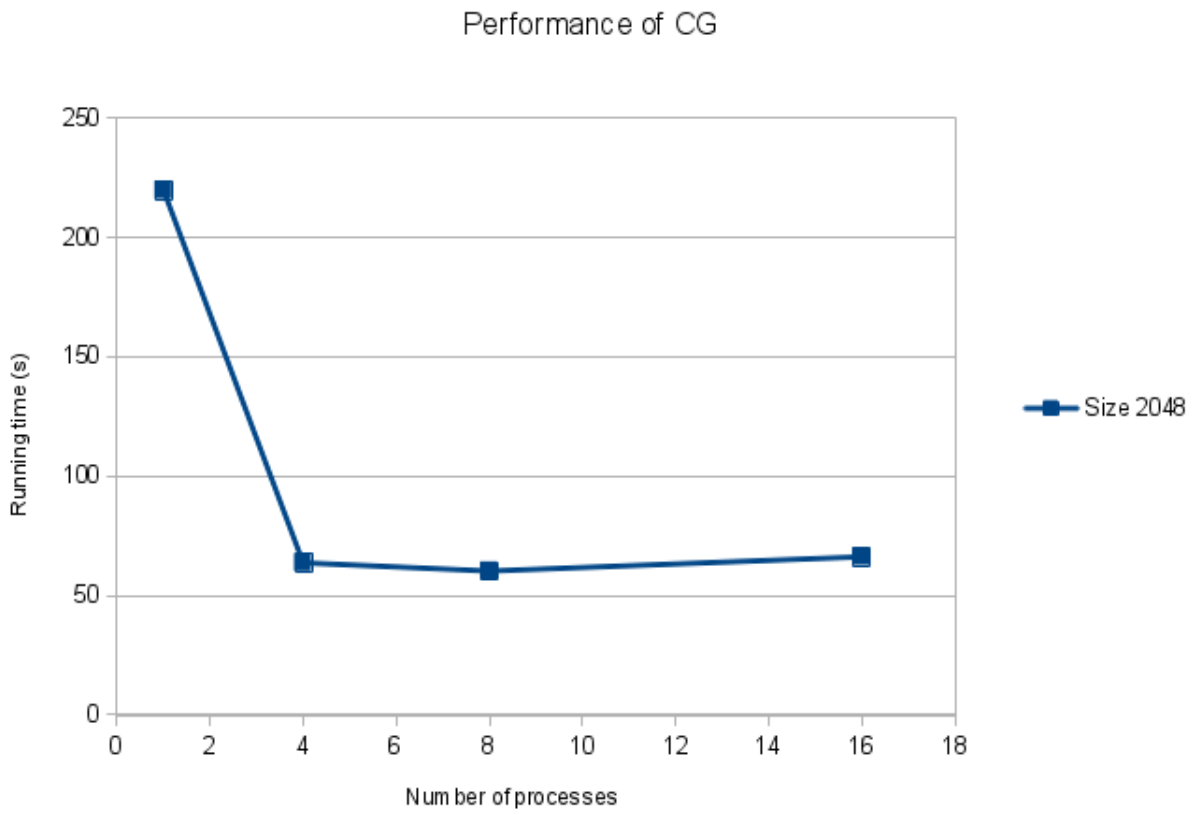

Figure 3: No. of processes Vs. execution time for a matrix size of 2048

6 Performance of the code using Spark SGD

The computational time of SGD to solve the linear system of equations of size 128 is 1.51 seconds with the mean squared error at 0.26. The computational time of computing SGD algorithm itself is 0.5 seconds at step size of 0.001. It is obvious that the computational time using Spark is higher than the MPI conjugate gradient algorithm when the problem size is smaller than 1000. Moreover, the accurate of the SGD algorithm is worse than the conjugate gradient algorithm. Even with a very small problem size, the SGD algorithm is not able to predict the exact results. However, when dealing with very complex problems (the matrix has a very large conditional number), the conjugate gradient method is not able to solve the problem. The SGD method can always estimate the results with a coarse prediction.

A study is performed on step size to understand the relationship between step size and the mean squared error and the relationship between step size and the computational time at the problem size of 128. Figure 7 shows that if the step size is decreasing, the computational time is decreasing. Whereas the mean squared error is increasing. Depending on the practical engineering problems, the accurate of the problem varies from 10^{-1} to 10^{-8}. Therefore, the SGD method is better for solving the problems requiring less accuracy.

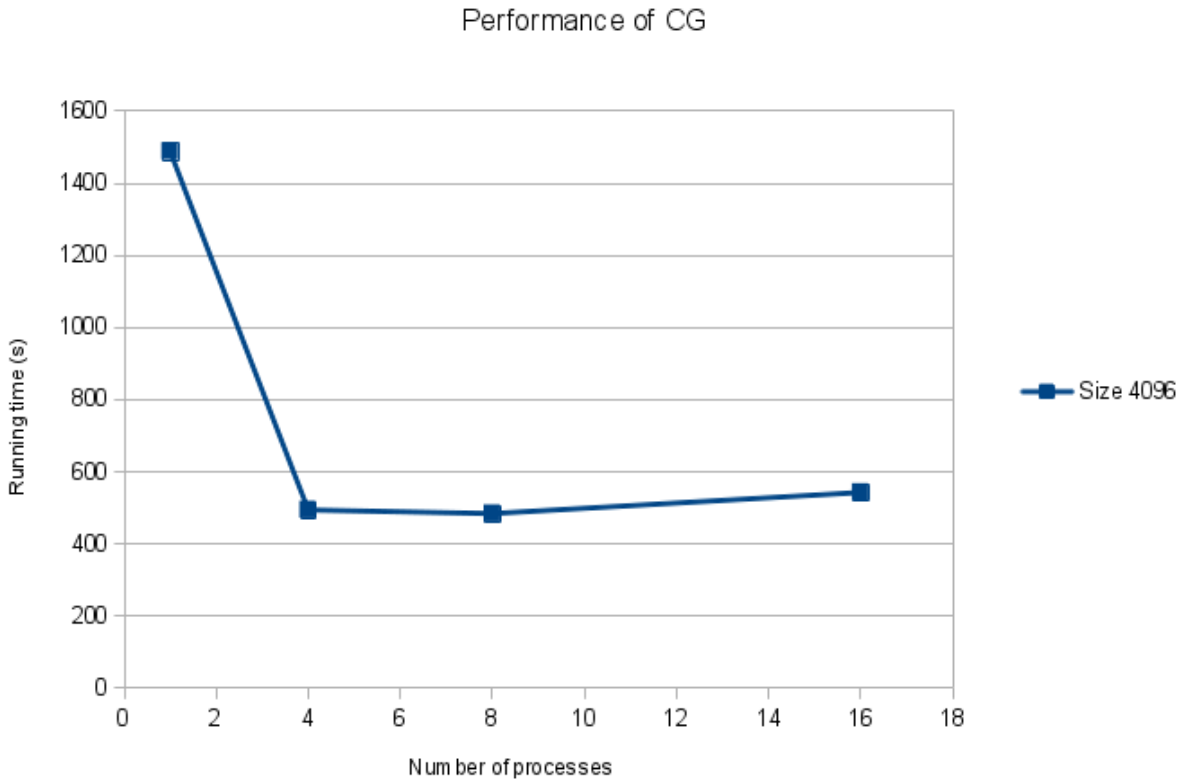

Figure 4: No. of processes Vs. efficiency for a matrix size of 4096

6.1 Performance of 1D and 2D matrix and vector multiplication

The theoretical computational time of calculating the matrix vector multiplication in 1D row-wise decomposition is:

$$T_P = \frac{n^2}{p} + t_s log p + t_w n \tag{5}$$

where n is the problem size, p is the number of processes, t_s is the transfer startup time and t_w is the per-word transfer time. On the other hand, the computational time of running matrix vector multiplication in 2D requires the time of:

$$T_P \approx \frac{n^2}{p} + t_s log p + t_w \frac{n}{\sqrt{p}} log(p) \tag{6}$$

The practical results are shown in the Figure 8. The computational time of running the matrix vector multiplication in 1D is similar to 2D matrix vector implementation with 4 processes at different matrix sizes. The running time is even higher in 2D implementation with 4 processes. The reason for this is that, for 2D matrix vector implementation, it requires more time to split the input matrix and RHS. Also the time of generating the 2D Cartesian topology may effect the final running time. The theoretical computational time formula also suggests that it is better to run with more processes in 2D matrix vector multiplication as the p is in denominator. It can be seen in Figure 8 that when the number

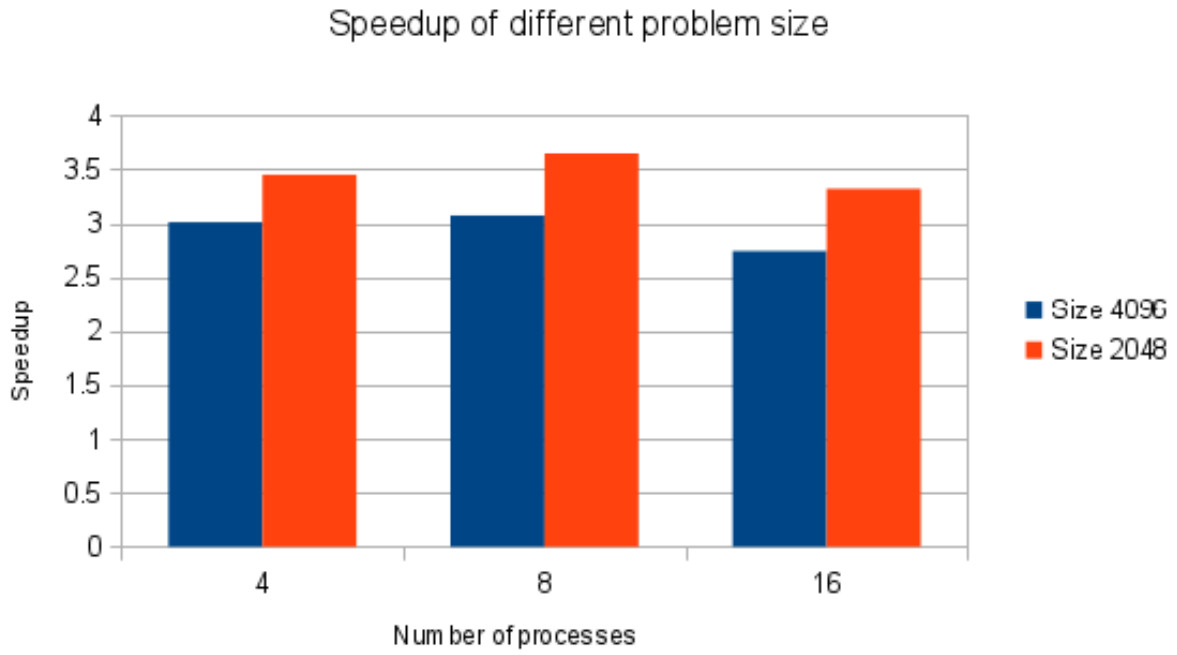

Figure 5: No. of processes Vs. Speedup of problem size 2048 and 4096

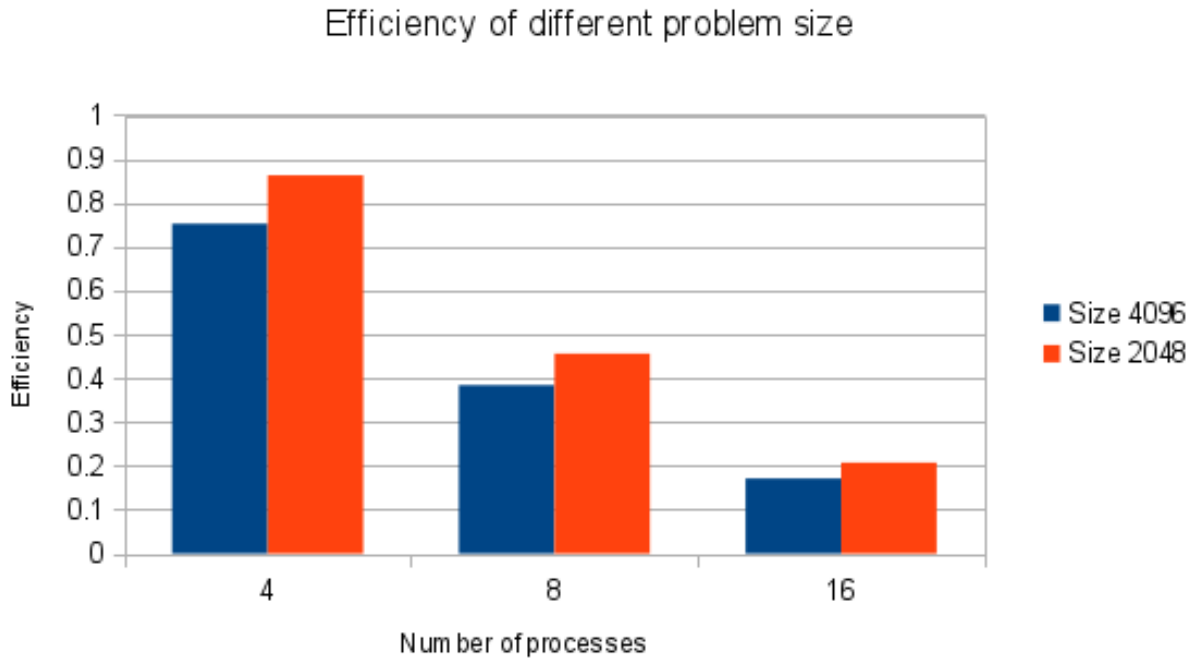

Figure 6: No. of processes Vs. efficiency of problem size 2048 and 4096

of processes is increased, the total amount of executing time in 2D implementation is less than the 1D implementation. However, comparing to the processes of 4, the amount of running time is increased in

Step size study

Step size	Error	Computational time(s)
1	0.0217	6.6900
0.5	0.0366	4.7700
0.1	0.0900	3.6800
0.01	0.2470	2.1500
0.001	0.2600	1.5300
0.0001	0.2600	1.5100

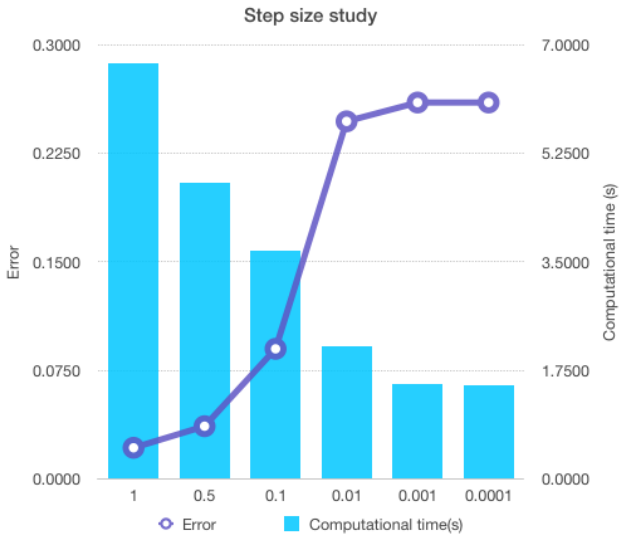

Figure 7: Step size study

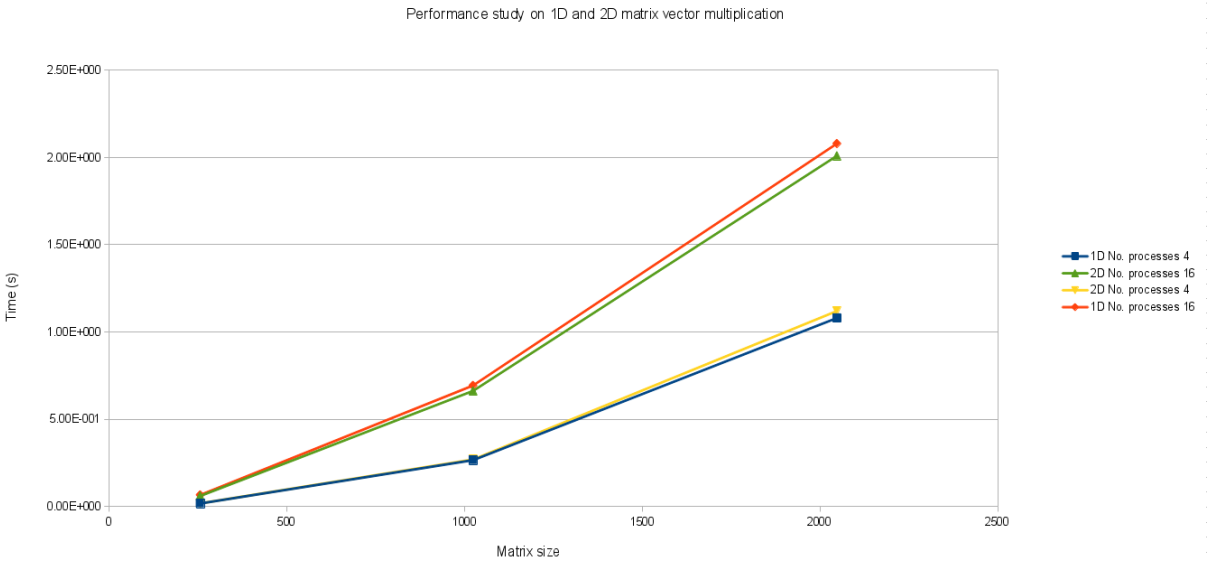

Figure 8: Performance study on 1D and 2D matrix vector multiplication

the case of 16 processes as it takes more time of launching more processes.

7 Conclusion

The report has presented the implementation of preconditioned CG method using MPI. From the evolution of the running time versus the number of processes under different matrix size and the knowledge

learned from class, increasing the number of processes is not always helping with improving in terms of running time since it can potentially introduce more overhead in the executing time i.e. communication time between the processes. The CG method is generally designed to be used to solve the engineering FEM problem. If the matrix is too ill-conditioned with a high condition number. It might not lead to a converge solution. Also, our example case in the code with a matrix size of $128 * 128$, the condition number of the matrix is very high at 10000. Therefore, the theoretical convergence iteration times (the problem will converge within the iteration time less than matrix size) will not match with the practical one as the complexity of the problem is very high. Thus, to address that, a more elegant method may be used when deals with the complex problems. In practical, a preconditioned CG method is capable to solve most of the engineering problems.

When the problem size is smaller, 1D matrix vector multiplication is already good enough to deal with most of the problems. When the problem size is large, 2D matrix vector multiplication with more processes is better in terms of performance.

The SGD method in Spark is also presented to compare with the performance in CG method implemented with MPI. The results of SGD method suggest that the SGD method is not able to predict the results with the same accuracy as CG method. However, the SGD method does not have the problem with the divergence and it is very robust to tackle with the complex problems.

Overall, it is suggested to use CG method to solve the engineering problems with sparse matrix. The SGD method is more designed for the machine learning algorithms.

8 Further work

The implementation of 2D CG algorithm for large size of engineering problems.

Appendices

The implementation of CG method.

```
1  /*Implementation of preconditioned CG method*/
2  int main(void) {
3      double* local_A;
4      double* local_D;
5      double* local_M;
6      double* local_At_b;
7      double* local_b;
8      double* local_Ab;
9      double* local_cg_x;
10     int n, local_n;
11     int my_rank, comm_sz;
12     MPI_Comm comm;
13     double start, finish, loc_elapsed, elapsed;
14     double r_norm,z_norm,z_norm_old;
15     double alpha,beta;
16     int k;
17     double maximum_iteration = 1000;
18     double rtol = 1e-8;
19
20     MPI_Init(NULL, NULL);
21     comm = MPI_COMM_WORLD;
22     MPI_Comm_size(comm, &comm_sz);
23     MPI_Comm_rank(comm, &my_rank);
24     /*Load matrix A from file data_input_matrix */
25     Loadinput_Matrix(&n, &local_n, &local_A, my_rank, comm_sz, comm);
26
27 #  ifdef DEBUG
28     Print_matrix("Matrix A", local_A, n, local_n, n, my_rank, comm);
29 #  endif
30
31     /*Load vector b from file data_input_RHS */
32     Loadinput_RHS_vector(n, local_n, &local_b, my_rank, comm_sz, comm);
33
34 #  ifdef DEBUG
35     Print_vector("RHS vector b", local_b, n, local_n, my_rank, comm);
36 #  endif
37     /*Allocating the memory */
38     Allocate_arrays(&local_D, &local_At_b, &local_cg_x, &local_Ab, &local_M, n, local_n,
           comm);
39     /*Computing transpose of A*/
```

```
40      Mat_Transpose(local_A , local_Ab ,n,local_n ,my_rank,comm);
41
42  #   ifdef DEBUG
43      Print_matrix("A_transpose", local_Ab , n, local_n , n, my_rank, comm);
44  #   endif
45      //matrix matrix multiplication AT * A = D
46      Mat_Mat_mult(local_Ab , local_A , local_D , n, local_n , comm);
47
48  #   ifdef DEBUG
49      Print_matrix("D", local_D , n, local_n , n, my_rank, comm);
50  #   endif
51
52      //Right hand side = AT b
53      Mat_vect_mult(local_Ab , local_b , local_At_b , local_n , n, local_n , comm);
54
55  #   ifdef DEBUG
56      Print_vector("AT_b", local_At_b , n, local_n , my_rank, comm);
57  #   endif
58
59      //Get the Mij = Aii
60      Get_M(local_M , local_D , local_n , n, my_rank);
61  #   ifdef DEBUG
62      Print_vector("Get M", local_M , n, local_n , my_rank, comm);
63  #   endif
64
65      MPI_Barrier(comm);
66
67      /*Time measurement */
68      start = MPI_Wtime();
69
70      double *p; /* Search direction */
71      double *r; /* Residual */
72      double *z; /* Temporary vector */
73      const int nbytes = local_n * sizeof(double);
74      p = (double *)malloc (nbytes);
75      r = (double *)malloc (nbytes);
76      z = (double *)malloc (nbytes);
77
78      /*
79       * Initialize all the varialbe at the first iteration step k = 0
80       */
81
82      /* x0 = 0 */
83      memset (local_cg_x , 0, nbytes);
```

```
84
85      /* r0 = b0 */
86      memcpy (r, local_At_b, nbytes);
87
88      /* z0 = M-1*r0 */
89      psolve(z, local_M, r, local_n);
90
91      /* p1 = r0 */
92      memcpy (p, z, nbytes);
93
94      /* residual norm = r0 * r0 */
95      r_norm = ddot(r,r,local_n,comm);
96
97      /*computing the derivative = r0*z0 */
98      z_norm = ddot(r,z,local_n,comm);
99
100     /*
101      * Entering the k loop
102      */
103     for (k = 0; k < maximum_iteration ; ++k){
104
105         z_norm_old = z_norm;                                    // assign zk-1 = zk
106
107         Mat_vect_mult(local_D, p, z, local_n, n, local_n, comm);  //  zk = A * pk
108
109         alpha = z_norm_old / ddot(p,z,local_n,comm);            // alpha = rk-1 * zk-1/
                pkT * zk (Apk)
110
111         axpy(local_cg_x, alpha, p, local_cg_x, local_n);     // xk+1 = xk + alpha pk
                updating the solution vector
112
113         axpy(r, -alpha, z, r, local_n);                     //rk+1 = rk- alpha zk
                updating the residual vector
114
115         psolve(z, local_M, r, local_n);                      // zk+1 = M-1/rk+1
                computing new zk+1
116
117         r_norm = ddot(r,r,local_n,comm);                    // r2 = rk+1 rK+1
                computing the new residual norm
118
119         z_norm = ddot(r,z,local_n,comm);                     // z2 = rk+1 zk+1
                computing the new searching derivative
120
121         beta = z_norm/z_norm_old;                            // beta = zk+1*zk+1/zk*zk
```

```
            computing the beta
122
123         axpy(p, beta, p, z, local_n);                    // pk+1 = zk + beta*pk
            updating the p vector
124
125         if (my_rank == 0){
126             printf("At iteration %d, alpha is %f and the residual is %lf \n",k,alpha,
                    r_norm);
127         }
128         /* Checking the convergence criterion*/
129         if(r_norm <= rtol)
130              break;
131     }
132
133 #   ifdef DEBUG
134     Print_vector("local_cg_x", local_cg_x, n, local_n, my_rank, comm);
135 #   endif
136
137     finish = MPI_Wtime();
138     loc_elapsed = finish-start;
139     MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
140     if (my_rank == 0)
141        printf("Elapsed time = %e\n", elapsed);
142
143     free(local_A);
144     free(local_D);
145     free(local_M);
146     free(local_Ab);
147     free(local_b);
148     free(local_At_b);
149     free(local_cg_x);
150     MPI_Finalize();
151     return 0;
152 }  /* main */
```

The functions to load the matrix and vector, the file names for loading the matrix is *data_input_matrix* and for loading the right hand side is *data_input_RHS*.

```
1   int Loadinput_Matrix(int *n, int *local_n, double **local_A,int my_rank,int comm_size,
        MPI_Comm comm)
2 {
3       FILE* ip;
4       int i,j;
5       double* A = NULL;
6       if (my_rank == 0) {
```

```
7           if ((ip = fopen("data_input_matrix","r")) == NULL){
8               printf("error opening the input data.\n");
9               return 1;
10          }
11          fscanf(ip, "%d\n", n);
12
13          A = malloc((*n)*(*n)*sizeof(double));
14          for (i = 0; i < *n; ++i){
15              for (j = 0; j< *n; ++j){
16                  fscanf(ip, "%lf\t", &A[i*(*n)+j]);
17              }
18          }
19          fclose(ip);
20      }
21      MPI_Bcast(n, 1, MPI_INT, 0, comm);
22      *local_n = *n/comm_size;
23      *local_A = malloc((*local_n)*(*n)*sizeof(double));
24      if (my_rank==0)
25      {
26          MPI_Scatter(A, (*local_n)*(*n), MPI_DOUBLE,
27                      *local_A, (*local_n)*(*n), MPI_DOUBLE, 0, comm);
28      }
29      else {
30          MPI_Scatter(A, (*local_n)*(*n), MPI_DOUBLE,
31                      *local_A, (*local_n)*(*n), MPI_DOUBLE, 0, comm);
32      }
33      return 0;
34  }
35  void Loadinput_RHS_vector(int n, int local_n, double **local_b, int my_rank, int
        comm_size, MPI_Comm comm)
36  {
37      FILE* ip;
38      int i;
39      double* b = NULL;
40      if (my_rank == 0) {
41          if ((ip = fopen("data_input_RHS","r")) == NULL){
42              printf("error opening the input data.\n");
43          }
44          b = malloc(n*sizeof(double));
45          for (i = 0; i < n; ++i){
46              fscanf(ip, "%lf\t", &b[i]);
47          }
48          fclose(ip);
49      }
```

```
50      *local_b = malloc(local_n*sizeof(double));
51      if (my_rank==0)
52      {
53          MPI_Scatter(b, local_n, MPI_DOUBLE,
54                      *local_b, local_n, MPI_DOUBLE, 0, comm);
55      }
56      else {
57          MPI_Scatter(b, local_n, MPI_DOUBLE,
58                      *local_b, local_n, MPI_DOUBLE, 0, comm);
59      }
60  }
```

The functions to print the matrix and vector.

```
1   void Print_matrix(
2         char        title[]     /* in */,
3         double      local_A[]   /* in */,
4         int         m           /* in */,
5         int         local_m     /* in */,
6         int         n           /* in */,
7         int         my_rank     /* in */,
8         MPI_Comm    comm        /* in */) {
9     double* A = NULL;
10    int i, j, local_ok = 1;
11
12    if (my_rank == 0) {
13       A = malloc(m*n*sizeof(double));
14       if (A == NULL) local_ok = 0;
15       Check_for_error(local_ok, "Print_matrix",
16             "Can't allocate temporary matrix", comm);
17       MPI_Gather(local_A, local_m*n, MPI_DOUBLE,
18             A, local_m*n, MPI_DOUBLE, 0, comm);
19       printf("\nThe matrix %s\n", title);
20       for (i = 0; i < m; i++) {
21          for (j = 0; j < n; j++)
22             printf("%f ", A[i*n+j]);
23          printf("\n");
24       }
25       printf("\n");
26       free(A);
27    } else {
28       Check_for_error(local_ok, "Print_matrix",
29             "Can't allocate temporary matrix", comm);
30       MPI_Gather(local_A, local_m*n, MPI_DOUBLE,
31             A, local_m*n, MPI_DOUBLE, 0, comm);
```

```
32      }
33  }  /* Print_matrix */
34
35  /*--------------------------------------------------------------------*/
36  void Print_vector(
37        char        title[]      /* in */,
38        double      local_vec[]  /* in */,
39        int         n            /* in */,
40        int         local_n      /* in */,
41        int         my_rank      /* in */,
42        MPI_Comm    comm         /* in */) {
43     double* vec = NULL;
44     int i, local_ok = 1;
45
46     if (my_rank == 0) {
47        vec = malloc(n*sizeof(double));
48        if (vec == NULL) local_ok = 0;
49        Check_for_error(local_ok, "Print_vector",
50              "Can't allocate temporary vector", comm);
51        MPI_Gather(local_vec, local_n, MPI_DOUBLE,
52              vec, local_n, MPI_DOUBLE, 0, comm);
53        printf("\nThe vector %s\n", title);
54        for (i = 0; i < n; i++)
55           printf("%f ", vec[i]);
56        printf("\n");
57        free(vec);
58     }  else {
59        Check_for_error(local_ok, "Print_vector",
60              "Can't allocate temporary vector", comm);
61        MPI_Gather(local_vec, local_n, MPI_DOUBLE,
62              vec, local_n, MPI_DOUBLE, 0, comm);
63     }
64  }  /* Print_vector */
```

References

[1] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Non-linear Equations in Several Variables and Systems.* Academic Press, New York, 1970.

[2] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing.* Addison-Wesley, second edition, 2003.