

STATE UNIVERSITY OF NEW YORK AT BUFFALO
Mechanical and Aerospace Engineering Department.

MAE 609 HIGH PERFORMANCE COMPUTING

Homework 2

NAME: LENG-FENG LEE

DATE: 10th Oct 2005.

MAE609 HIGH PERFORMANCE COMPUTING I

Homework #2

Leng-Feng Lee

Problem: Please program and test on 1-32 processors a parallel version of the Preconditioned Conjugate Gradient algorithm for solving systems of symmetric positive definite systems using MPI. The PCG solution algorithm for solving $Ax=B$ is documented as below in:

<http://www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/f3346.html>

You may choose M as inverse of the diagonal of A . $\text{norm}(z) = \sqrt{z^T z}$. The distribution (and possibly generation) of A and B among processors offers many possibilities. The matrix vector product $A*x$ is computationally most challenging.

Solution:

A serial code and a parallel code of the precondition conjugate method were written in C. Both codes were run on Lennon, the CCR SGI machine. A maximum number of 64 processors were used to test the parallel program. In the parallel code, the only part that was parallelized was the matrix vector multiplication part. This was done by each processor calculate an allocated amount of row-vector multiplication of a $N \times N$ matrix. The results of each row-vector multiplication were then collected on one processor, get the resulting $N \times 1$ vector, and send it back to each processor to continue the necessary process. Several tests were performed, and a maximum of 64 processors were used to calculate a maximum of a 3200×3200 matrix-vector multiplication. The result of the precondition conjugate method was compare to one solved using MATLAB, and they are identical.

The A , B , and preconditioned M matrix used are listed here:

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & \ddots & -1 & 0 & \vdots \\ \vdots & 0 & -1 & \ddots & -1 & 0 \\ \vdots & 0 & 0 & -1 & \ddots & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2 \end{bmatrix}_{N \times N}$$
$$B = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix}_{N \times 1}$$
$$M_{N \times N} = \text{diag}[A_{N \times N}]$$

Here is some information of the Lennon Machine used:

Front-End (interactive use, debugging, cpu time limit of 30m)

Hostname = lennon.ccr.buffalo.edu (front end for logins, job submission)

Vendor = [SGI](#)

Architecture = Altix 350 Server

Number of Processors = 4

Operating System: Redhat Enterprise Linux 3.0 + SGI Propack 3.0

Processor Description:

4 1.3GHz Itanium2 (Madison) Processors

Main memory size: 8 GBytes

Instruction cache size: 16 KBytes

Data cache size: 16 KBytes

Secondary unified instruction/data cache size: 256 KBytes

Level 3 Cache: 3MBytes

(More information: <http://www.ccr.buffalo.edu/hotpages/content/lennon/lennon.htm>)

The result:

Table 1: Serial Code :

Test #	Matrix Size	Computation time (sec)
1	640	24.11
2	1280	150.11
3	1920	337.46
4	2560	599.18
5	3200	966.65

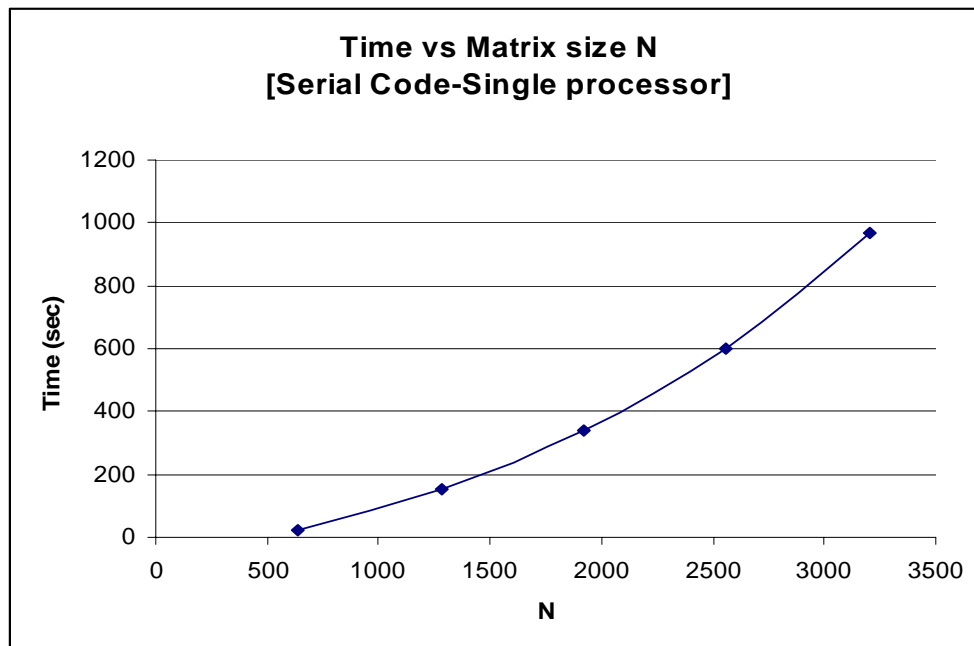


Figure 1

Table 2: Parallel code (utilized 4 processors):

Test #	Matrix Size	# of Processors used	Computation time (sec)
1	640	8	3.14
2	1280	8	18.64
3	1920	8	41.19
4	2560	8	73.69
5	3200	8	112.57

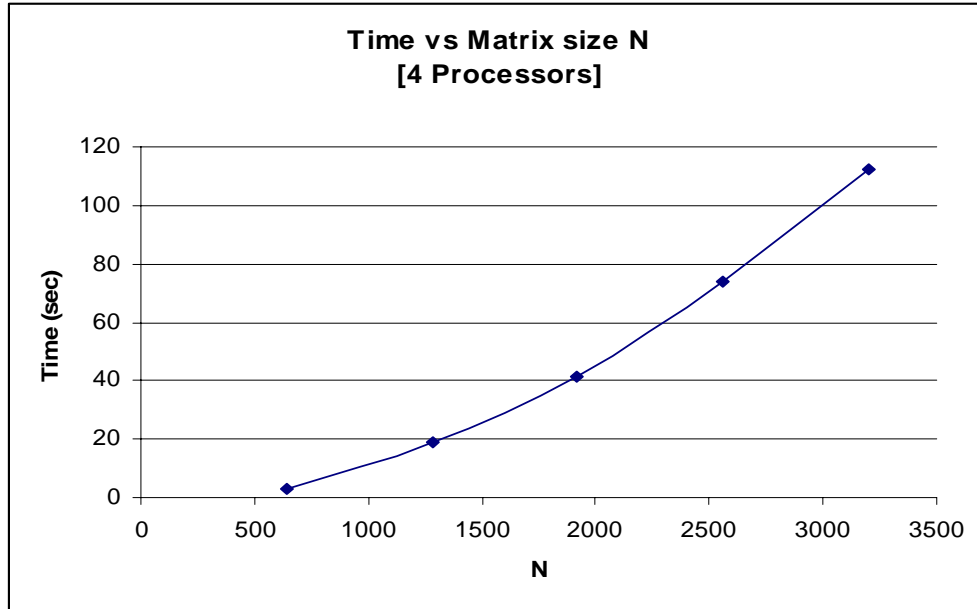


Figure 2

Table 3: Parallel code (utilized 8 processors):

Test #	Matrix Size	# of Processors used	Computation time (sec)
1	640	8	3.22
2	1280	8	17.25
3	1920	8	37.62
4	2560	8	67.39
5	3200	8	101.12

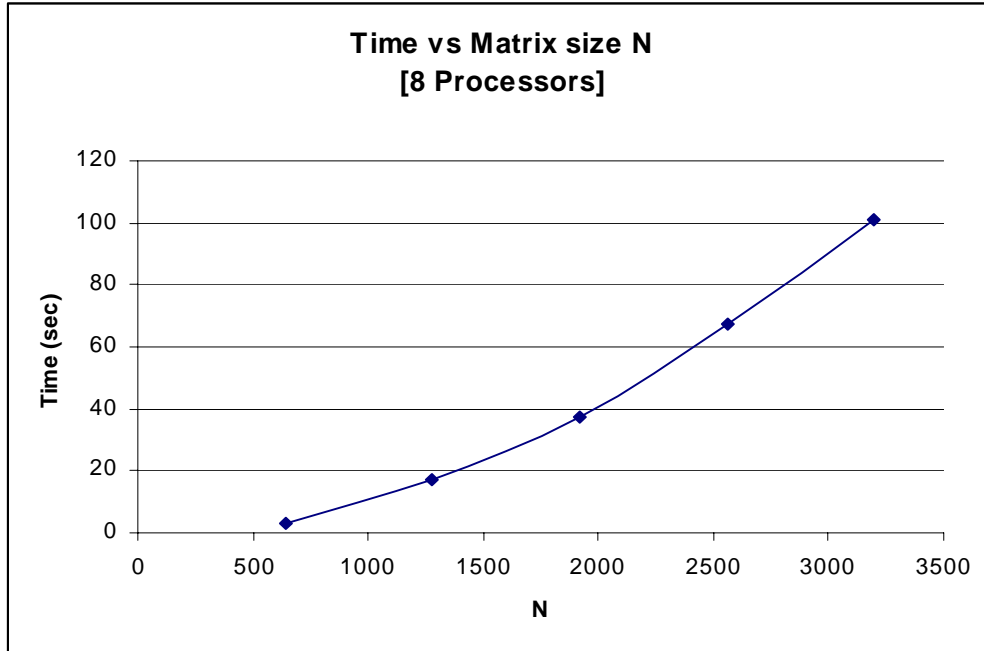


Figure 3

Table 4: Parallel code (utilized 16 processors):

Test #	Matrix Size	# of Processors used	Computation time (sec)
1	640	16	3.05
2	1280	16	17.22
3	1920	16	37.25
4	2560	16	64.14
5	3200	16	98.43

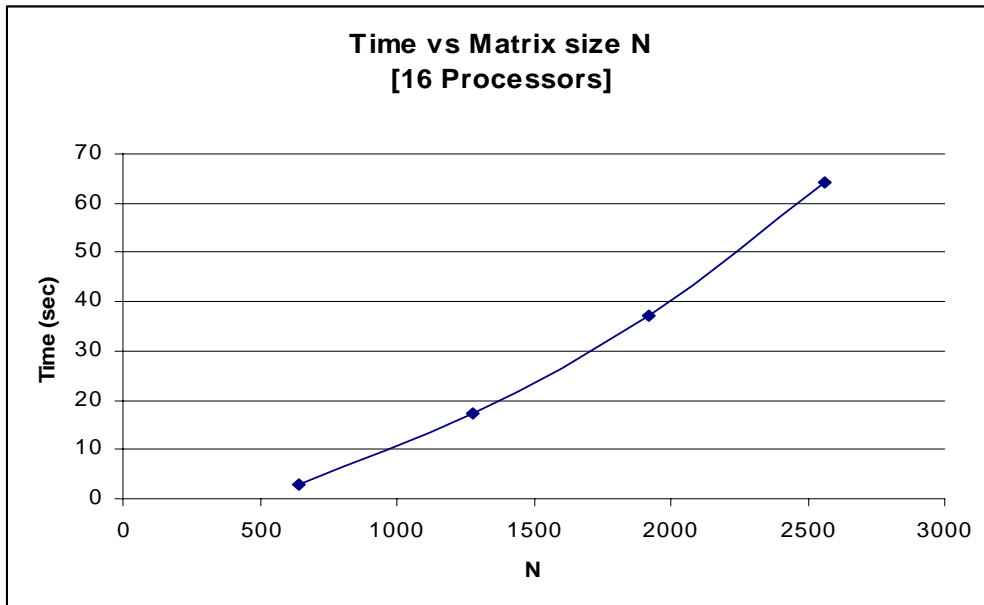


Figure 4

Table 5: Parallel code (utilized 32 processors):

Test #	Matrix Size	# of Processors used	Computation time (sec)
1	640	32	3.44
2	1280	32	17.86
3	1920	32	37.55
4	2560	32	64.93
5	3200	32	99.04

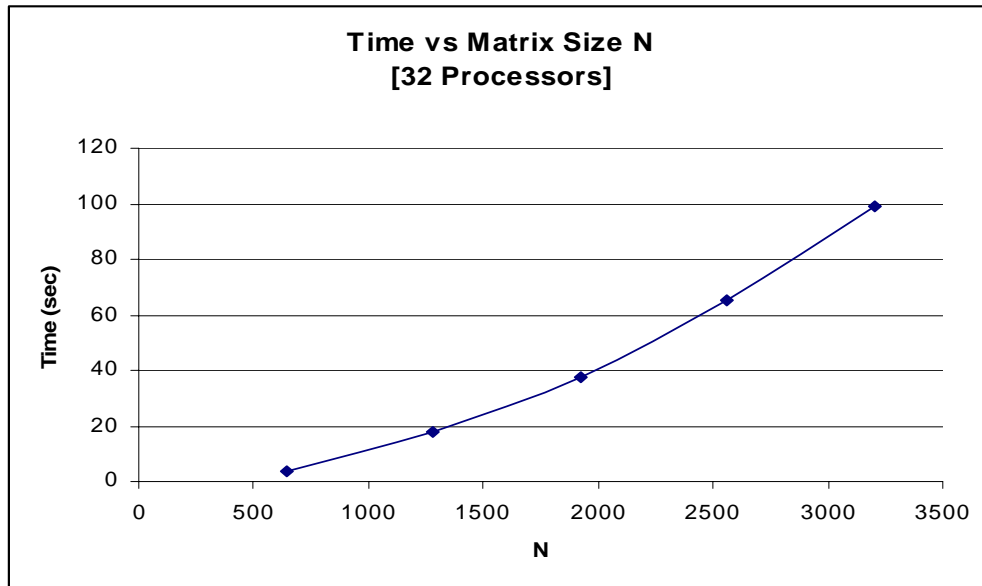


Figure 5

Table 6: Parallel code (utilized 64 processors):

Test #	Matrix Size	# of Processors used	Computation time (sec)
1	640	64	4.25
2	1280	64	20.28
3	1920	64	41.68
4	2560	64	69.36
5	3200	64	103.47

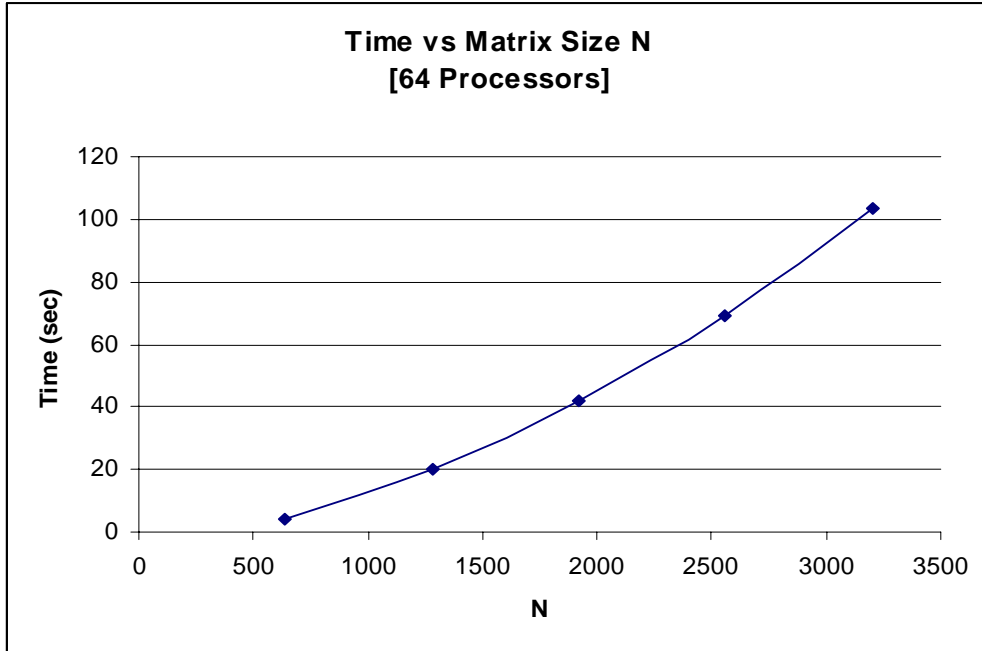


Figure 6

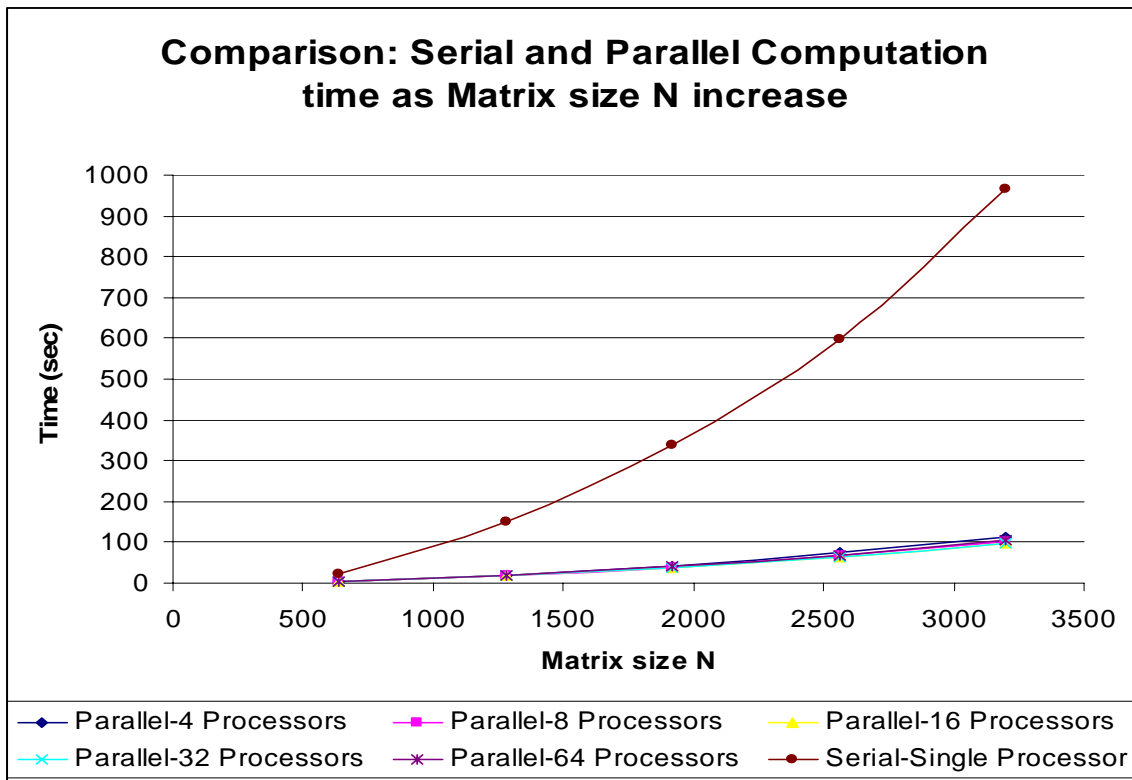


Figure 7

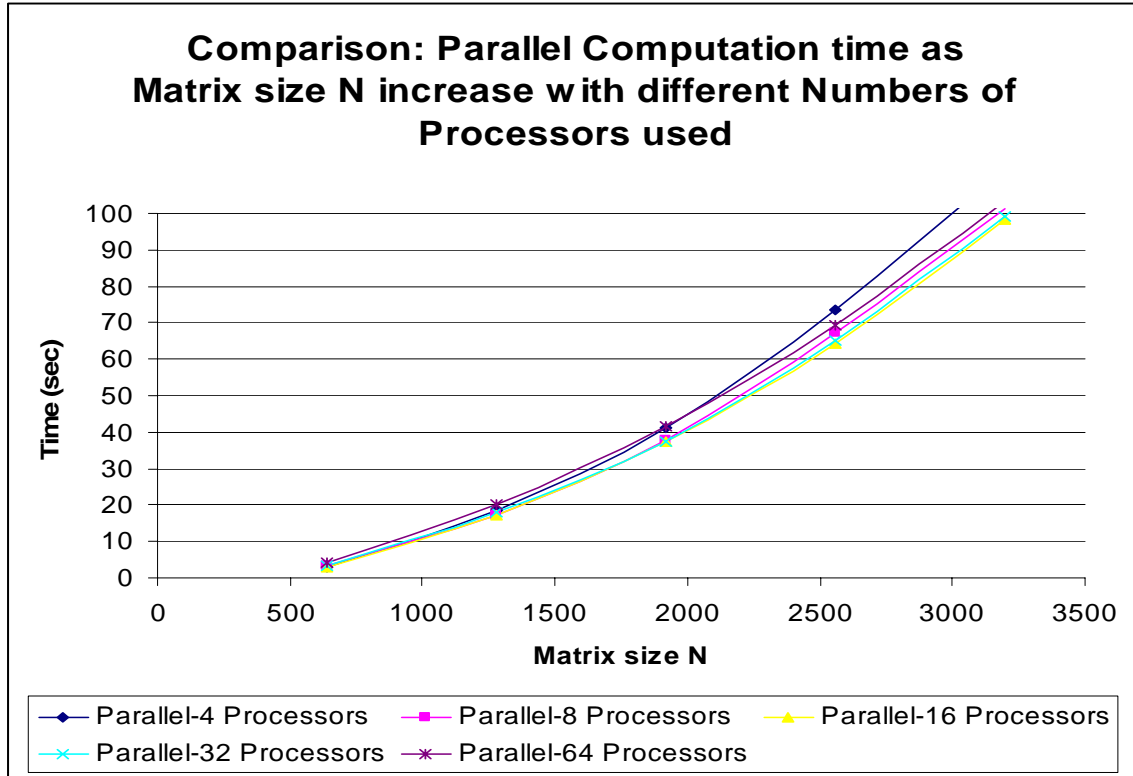


Figure 8

Discussion:

From Table (1-6) and Figure (1-8) shown in the previous section, it shows that the parallel code implementation gives a faster computational time (avg 3.2 sec to calculate a 640x640 matrix-vector multiplication) compare to the serial code result (24 sec). We performed the comparison in two ways: (1). We want to know how fast we can achieved by implementing parallel processing to the matrix-vector multiplication compare to the serial processing of the same matrix-vector multiplication. (2). We want to know as we increase the processors in parallelizing the matrix-vector multiplication, how fast we can improve our computational speed.

The first question can be answer by observing Figure 7. Note that in Figure 7, as soon as we parallelize the code, we can see an improvement in computation speed compare to conventional serial code implementation. For N=640, the it give an average of 20 seconds faster in obtaining the result. For N=3200, the increase in computation speed is an average of 868 seconds. This result shown a significant improvement in computational speed after parallelized the code.

Figure 8 answer the second question: from Figure 8, we see that there is no significant gain in computational speed as we increase the processors used in our parallel implementation. That is, running the parallel code on 64 separate processors did not improve the result very much compare to running the parallel code on 4 separate processors. This, we believe can cause by several possible implementation issues. The first is the possibly the nature of the matrix-vector multiplication parallelization scheme that we utilized here, and other might be the communication scheme that we implemented in the code. The matrix-vector multiplication method we used here is that each processor is assign N/P (where N is the size of the matrix and P is number of processor) number of row-vector multiplication, send it to a single processor, assembled the resulting $N \times 1$ vector, then send it back to each processor to complete the computation. There might be a better way to handle this operation. Also, the codes that we wrote are generally not being optimized in any sense. This part of code may be consuming time and overcome the overall improvement that increase number of processors might bring to us.

Note: The serial and parallel codes are attached for reference.

```

/* -----
Name: Leng-Feng Lee
Course: High Performance Computing I
Instructor: Dr Patra
Description: Solve  $Ax = B$  equation using preconditioned
            Conjugate Gradient Method.
Note: This is the serial solution.
Last Modify: 03 October 2005.
-----*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#define N 128

int i, j, k;
double alpha=0.0, beta=0.0;
double A[N][N], B[N], M[N][N], P[N], R[N], Z[N], D[N], W[N], InvM[N][N];
int main()
{
    int i, j, k;
    int Max_iter=1000;

    /*double A[N][N], B[N], M[N][N], P[N], R[N], Z[N], D[N], W[N];
    double InvM[N][N];*/
    double inner1=0, inner2=0;
    double denom=0, sumD=0, sumZ=0, normZ=0;
    double tol = 0.001;
    clock_t time0, time1;          /* used for timing */
    double cpu_time_used;

    for (i=0;i<N;i++)
    {
        B[i]= i+1;
        for (j=0;j<N;j++)
            A[i][j]=0.0;
    }

    for (i=0;i<N;i++)
    {
        for (j=0;j<N;j++)

```

```

    {
        if(i==j)
        {
            A[i][j]=2.0;
            if (j==0)
                A[i][j+1]=-1.0;
            else if (j==(N-1))
                A[i][j-1]=-1.0;
            else
            {
                A[i][j+1]=-1.0;
                A[i][j-1]=-1.0;
            }
        }
    }
}
/*for (i=0;i<N;i++)
{
    for(j=0;j<N;j++)
        printf("%f ", A[i][j]);
    printf("\n");
}*/

/*printf("\n\n");*/
/*for (i=0;i<N;i++)
{
    for (j=0;j<N;j++)
    {
        A[i][j]=A[i][j]*A[j][i];
        printf("%f ",A[i][j]);
    }
    printf("\n");
}*/

/* ---- Original 3x3 Matrix -----
A[0][0]=17.0;   A is H in the code
A[0][1]=10.0;
A[0][2]=18.0;
A[1][0]=10.0;
A[1][1]=6.0;
A[1][2]=11.0;
A[2][0]=18.0;
A[2][1]=11.0;
A[2][2]=21.0;

```

```

B[0]=4.0;          B is G in the code
B[1]=2.0;
B[2]=1.0;
----- Replace with Nx1 matrix -----*/

time0 = clock();
/*Initialization*/
for(i=0; i<N; i++)
{
    P[i]=0;
    R[i]=B[i];
    W[i]=0;
    Z[i]=0;
    for(j=0; j<N; j++)
    {
        M[i][j]=0;
        InvM[i][j]=0;
        if (i==j)
        {
            M[i][j]=A[i][j];          /*Assign M is the diagonal of A*/
            InvM[i][j]=1/M[i][j];
        }
    }
}

/*for (i=0;i<N;i++)
{
    for(j=0;j<N;j++)
        printf("%f  ",InvM[i][j]);
    printf("\n");
}*/
/*-----Precondition----- */

for (i=0;i<N;i++)
{
    for (j=0;j<N;j++)
        Z[i]= InvM[i][j]*R[j] + Z[i];
}

for (i=0;i<N;i++)
{
    inner1 = R[i]*Z[i]+ inner1;
}

```

```

    D[i]=Z[i];
}
/*printf("inner1 is %f\n\n",inner1);
printf("\n\ninner2 is %f",inner2);*/
/* printf("The D vector is");
   for (i=0; i<N; i++) printf(" %f \n",D[i]);*/

/*----Conjugate Gradient Iteration-----*/

for (k=1; k < Max_iter; k++)
{
    /*k=Max_iter;*/
    if (k > 1)
    {
        beta = inner1/inner2;
        for (i=0;i<N;i++) D[i]=Z[i]+beta*D[i];
        /*printf("\nbeta is %f \n",beta);*/
    }
    for (i=0;i<N;i++) W[i]=0;
    /*printf("The W vector is");
    for (i=0; i<N; i++) printf(" %f \n",W[i]);*/

    for (i=0; i<N; i=i++)
    {
        for (j=0; j<N; j++)
            W[i]=A[i][j]*D[j]+W[i];    /*nxn matrix times a nx1 matrix*/
    }
    /*printf("The W vector is");
    for (i=0; i<N; i++) printf(" %f \n",W[i]);*/

    denom=0;
    for (i=0; i<N; i++) denom = D[i]*W[i]+denom;

    if (denom <= 0)
    {
        sumD=0;
        for (i=0;i<N;i++) sumD = D[i]*D[i] + sumD;           /* Find the norm(D) */
        for (i=0;i<N;i++) P[i] = D[i]/sqrt(sumD);           /* Find p=norm(D) */
        printf(" denom is less than zero");
        break;
    }
    else

```

```

{
    alpha = inner1/denom;
    for (i=0;i<N;i++) P[i] = P[i] + alpha*D[i];
    for (i=0;i<N;i++) R[i] = R[i] - alpha*W[i];
}
/*Update Z */
for (i=0;i<N;i++) Z[i]=0;
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
        Z[i]=InvM[i][j]*R[j]+Z[i];
}

sumZ=0;
for(i=0; i<N; i++) sumZ = Z[i]*Z[i] + sumZ;
normZ = sqrt(sumZ);

if (normZ <= tol)
    k=Max_iter;

inner2 = inner1;
inner1 = 0;
for(i=0; i<N; i++) inner1 = R[i]*Z[i] + inner1;
}
time1= clock();
cpu_time_used = ((double)(time1-time0))/CLOCKS_PER_SEC;

printf("\n\nThe Answer is:\n");
for (i=0;i<N;i++) printf("%f \n", P[i]);

printf("\n\n CPU time used is %f \n", cpu_time_used);

return 0;
}

```

```

/* -----
Name: Leng-Feng Lee
Course: High Performance Computing I
Instructor: Dr Patra
Description: Solve  $Ap = B$  equation using preconditioned
            Conjugate Gradient Method.
Note: This is the Parallel solution for Homework #2.
Last Modify: 02 October 2005.
Note:
1. This code compiled successfully on lennon.ccr.buffalo.edu
2. Use N number of processor to compile, N is the number of the A
   matrix (NxN) size.
-----*/

```

```

#include <stdio.h>
#include <math.h>
#include "mpi.h"

#define N 3200

double A[N][N], B[N], M[N][N], P[N], R[N], Z[N], D[N], W[N], InvM[N][N];

main(int argc, char** argv)
{
    int i, j, k;
    int Max_iter=1000;          /*Maximum iterations */

    /*double A[N][N], B[N], M[N][N],P[N],R[N],Z[N],D[N],W[N];
    double InvM[N][N];*/
    double inner1=0, inner2=0;
    double alpha=0.0, beta=0.0;
    double denom=0, sumD=0, sumZ=0, normZ=0;
    double tol = 0.0001;
    /*double temp_W=0;*/

    /* Initialization use for Parallel processing */
    int p_rank;          /* Processor ID */
    int p;              /* Total No. of processors used*/
    int local_i=0;      /* row numbering for each processor*/
    int local_j=0;      /* column numbering for each processor*/
    int local_N =0;
    int source=0;       /* where the message come from */
    int tag=0;

```

```

double temp_W[N];
MPI_Status status;
double time0, time1; /* used for timing */

/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get the process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

local_N=N/p;
double local_W[local_N];
local_W[0]=0;
/* ---- Generated N size A and B Matrix -----*/
for (i=0; i<N; i++)
{
    B[i]=i+1;
    for (j=0; j<N; j++)
        A[i][j]=0.0;
}

for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
    {
        if (i==j)
        {
            A[i][j]=2.0;
            if (i==0)
                A[i][j+1]=-1.0;
            else if (j==(N-1))
                A[i][j-1]=-1.0;
            else
            {
                A[i][j+1]=-1.0;
                A[i][j-1]=-1.0;
            }
        }
    }
}
}

```

```

/*----- End -----*/

/*for (i=0;i<N;i++)
{
  for (j=0;j<N;j++)
    printf("%f ", A[i][j]);
  printf("\n");
}*/

time0 = MPI_Wtime();

/*-----Initialization-----*/
for(i=0; i<N; i++)
{
  P[i]=0;
  R[i]=B[i];
  W[i]=0;
  Z[i]=0;
  for(j=0; j<N; j++)
  {
    M[i][j]=0;
    InvM[i][j]=0;
    if (i==j)
    {
      M[i][j]=A[i][j]; /*Assign M is the diagonal of A*/
      InvM[i][j]=1/M[i][j];
    }
  }
}

/*-----Precondition----- */

for (i=0;i<N;i++)
{
  for (j=0;j<N;j++)
    Z[i]= InvM[i][j]*R[j] + Z[i];
}

for (i=0;i<N;i++)
{
  inner1 = R[i]*Z[i]+ inner1;
  D[i]=Z[i];
}

```

```

/*----Conjugate Gradient Iteration-----*/

for (k=1; k < Max_iter; k++)
{
  if (k > 1)
  {
    beta = inner1/inner2;
    for (i=0;i<N;i++) D[i]=Z[i]+beta*D[i];
  }
  for (i=0;i<N;i++)
  {
    W[i]=0;
    /*local_W[i]=0;*/
    /*temp_W[i]=0;*/
  }

  /* ----- Parallel this part-----*/
  /*local_N      = N/p;*/
  /*local_i_ini  = p_rank*local_N;*/
  /*local_i_max  = p_rank*local_N + local_N;*/
  /*local_jmax   = N;*/
  for (i=0;i<local_N;i++) local_W[i] = 0;

  for (i = 0; i < local_N; i++)
  {
    for (j=0; j<N; j++)
    {
      local_W[i] =A[i+p_rank*local_N][j]*D[j]+local_W[i];
    }
    /*printf(" %f , from %d\n",local_W[i],p_rank);*/
  }

  if (p_rank == 0)
  {
    for (i=0; i<local_N;i++) W[i]=local_W[i];
    for (source=1; source<p; source++)
    {
      MPI_Recv(&temp_W, local_N, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
      for (i=0; i<local_N; i++) W[i+source*local_N]=temp_W[i];
    }
    /*printf("The W vector is:");
    for (i=0;i<N;i++) printf(" %f \n",W[i]);*/
  }
}

```

```

}
else /*The other remaining processors*/
{
    MPI_Send(&local_W, local_N, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
}
/*k=Max_iter;*/
if (p_rank == 0)
{
    for (source=1; source<p; source++)
        MPI_Send(&W, N, MPI_DOUBLE, source, tag, MPI_COMM_WORLD);
}
else
{
    MPI_Recv(&W, N, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
}

/* ---Original Serial Part-----
for (i=0; i<N; i=i++)
{
    for (j=0; j<N; j++)
        W[i]=A[i][j]*D[j]+W[i];    nxn matrix times a nx1 matrix
}----- */

/* -----End Parallelization-----*/
denom=0;
for (i=0; i<N; i++) denom = D[i]*W[i]+denom;

if (denom <= 0)
{
    sumD=0;
    for (i=0;i<N;i++) sumD = D[i]*D[i] + sumD;          /* Find the norm(D) */
    for (i=0;i<N;i++) P[i] = D[i]/sqrt(sumD);          /* Find p=norm(D) */
    break;
}
else
{
    alpha = inner1/denom;
    for (i=0;i<N;i++) P[i] = P[i] + alpha*D[i];
    for (i=0;i<N;i++) R[i] = R[i] - alpha*W[i];
}
/*Update Z */
for (i=0;i<N;i++) Z[i]=0;
for (i=0; i<N; i++)

```

```

    {
        for (j=0; j<N; j++)
            Z[i]=InvM[i][j]*R[j]+Z[i];
    }

    sumZ=0;
    for(i=0; i<N; i++) sumZ = Z[i]*Z[i] + sumZ;
    normZ = sqrt(sumZ);

    if (normZ <= tol)
        k=Max_iter;

    inner2 = inner1;
    inner1 = 0;
    for(i=0; i<N; i++) inner1 = R[i]*Z[i] + inner1;
}

time1 = MPI_Wtime();
if (p_rank==0)
{
    printf("\n\nThe Answer is (processor %d:\n",p_rank);
    for (i=0;i<N;i++) printf("%f \n", P[i]);
}
printf("\nTime used for this process is:%f on processor %d\n\n ",time1-time0,p_rank);

MPI_Finalize();
return 0;
}

```