

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

MAE 609 HIGH PERFORMANCE COMPUTING

Homework 5

NAME: LENG-FENG LEE

DATE: 19th Dec 2005.

MAE 609 HIGH PERFORMANCE COMPUTING' Fall 2005

HOMEWORK #5

Leng-Feng Lee

PROBLEM 1. Please start with the code from HW 3, 4 and improve its performance by adding suitable open MP Commands.

Solution: The parallel code for homework #3 and #4 is added with OpenMP directives. It was done in two ways. The first approach contains only the OpenMP code and the second approach contain the usage of both MPI and OpenMP. In the second case, the tricky part comes from the compilation of the program. To request the desired number of processors (for MPI) and the number of threads (for OpenMP) required modification of the “.bashrc” file – the thread number is set inside this file. After running this file, we can compile the parallel code as usual. In the following paragraphs, the performance analyses for both cases with various problem (grid) sizes were shown separately.

Case 1: Parallelization with OpenMP.

In this case, the OpenMP directives are added to the code accordingly (code is included for reference). The OpenMP uses a different concept in parallelization. The main difference is that in OpenMP, only part of the code is parallelized when necessarily. On the other hand, the entire program is parallelized in MPI. The disadvantage (or could be advantage for some) of using OpenMPI is that we have no control of the way OpenMP parallelize the code. The only thing that we need to set is to specify the variables that belong to either “private” or “shared”.

In the code that was written in homework 3, 4, the parallelized portion of the code that previously done using MPI is now replaced with OpenMP directives.

Case 2: Parallelization with both MPI and OpenMP.

In order to “get the best of both OpenMP and MPI”, we combine the usage of both in one program. In homework 3 & 4, we parallelized the grid row-wise and distributed processing each portion of the grid. Now, we further distribute each portion of

the grid that each processor has into different threads using OpenMP. This is best described in Figure 1.

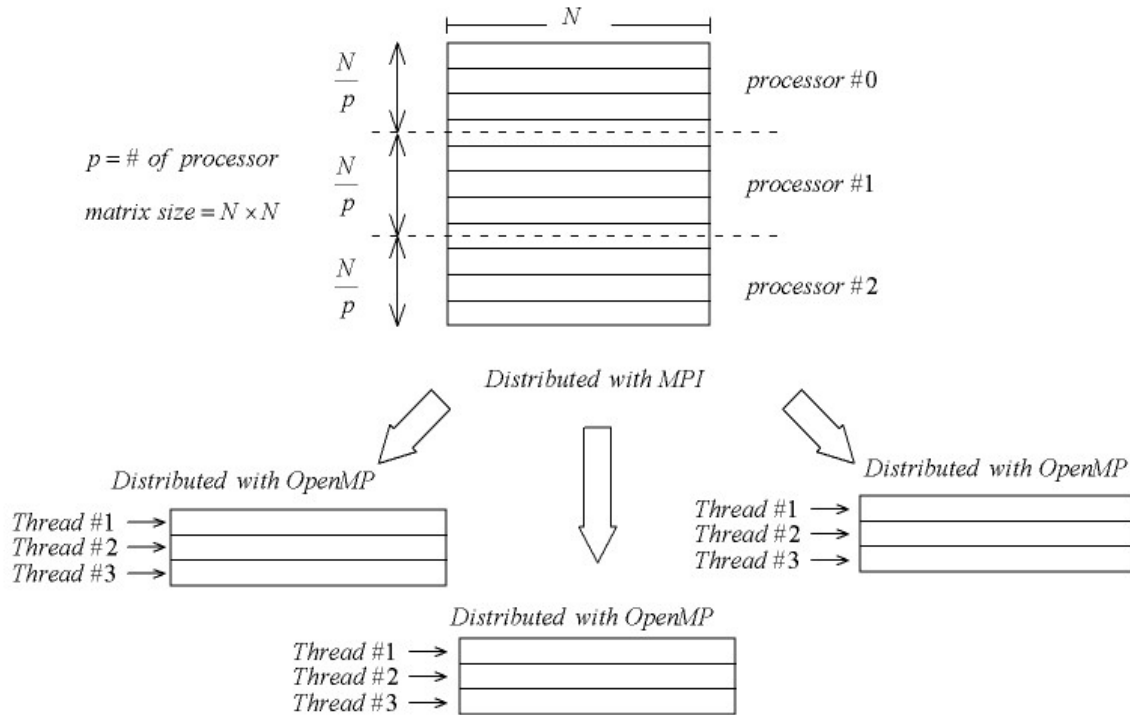


Figure 1: Parallelization scheme that utilize both MPI and OpenMP.

The difficulties come in during the compilation of the program and setting the correct numbers of threads used in each processor. To correctly set the threads used, the following need to be done. (1) Edit the \$HOME/.bashrc file to set number of threads using “export OMP_NUM_THREADS=4”; and (2) In the PBS script, invoke mpiexec using “-pernode” option.

The code for this scheme is also included.

Performance Study:

Case study 1: Using only OpenMP

The performance that obtained from using solely OpenMP is not good compare to the result that obtained using MPI. For example, it took 205 seconds to run a 256x256 size grid on 2 processors using MPI, but it took 525 seconds to the same grid size on 2

threads. Certainly this is not correct (but the result of the reaction-diffusion is identical in both cases). The result need to be further studied. (Note: most probably is that the number of threads not being set properly during compilation).

Case 2: Parallelization with both MPI and OpenMP.

Same result obtained as case study 1 above. The performance did not improve as the number of threads increased. It is most probably that the number of threads is not being properly set during the compilation of the program (the default thread number is being used). The code provides the correct result for the reaction-diffusion reaction, but did not give any improvement on the computing time. Again, need to work further on this.

PROBLEM 2. Rewrite the code for the CG version with an efficient sparse storage scheme for the matrix.

Solution: When choosing a storage scheme, one of the main issues is the reduction of ‘net’ storage requirements. With a sparse storage scheme, there are two types of storages needed: (1) The “primary” storage which contains the actual non-zero matrix components and (2) the “overhead” storage which contains all the indexing information dealing with the primary storage. It is obvious that there has to be a trade-off point. The more sophisticated the sparse storage scheme, the more indexing is needed and thus decreases the primary storage but gives an increase in overhead storage.

Using the CG solver code written in homework 2, the sparse matrix has the following form:

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & \ddots & -1 & 0 & \vdots \\ \vdots & 0 & -1 & \ddots & -1 & 0 \\ \vdots & 0 & 0 & -1 & \ddots & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix}_{N \times N}$$

In this case the matrix has a bandwidth that is fairly constant along the rows, hence it is worthwhile to take advantage of this structure by storing the subdiagonals in consecutive memory locations. Using the Compressed Diagonal Storage (CDS) form, the matrix A can be store in the following contiguous format:

$$A' = \begin{bmatrix} 0 & -1 & \dots & -1 & -1 \\ 2 & 2 & \dots & 2 & 2 \\ -1 & 1 & \dots & 1 & 0 \end{bmatrix}_{3 \times N}$$

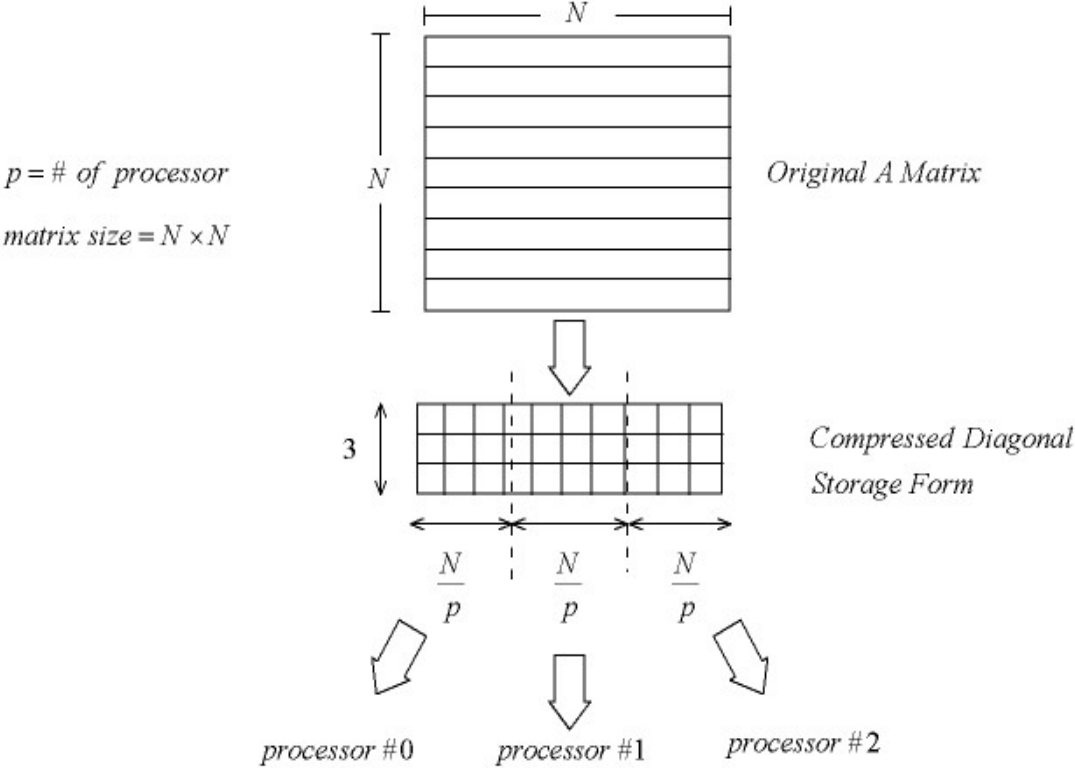
Note that in this storage scheme, the size of matrix A reduced from NxN to 3xN (size of A'). The matrix vector multiplication routine need to be modified to accordingly. The modification is shown in the following diagram (For serial code).

<pre>for (i=0; i<N; i=i++) { for (j=0; j<N; j++) W[i]=A[i][j]*D[j]+W[i]; } </pre>	<pre>for (j=0; j<N; j++) { for (i=0; i<3; i++) W[j]=A[i][j]*D[j+i-1]+W[j]; } </pre>
---	---

Original Matrix-vector Multiplication

Modified Matrix-vector Multiplication

To parallelized the matrix-vector multiplication, the 3xN matrix A' is divided column-wise (In the without using the compressed diagonal storage, it is divided row-wise among processor, as shown in homework #2 completed by the author). This parallelization scheme can be illustrated using the following diagram.

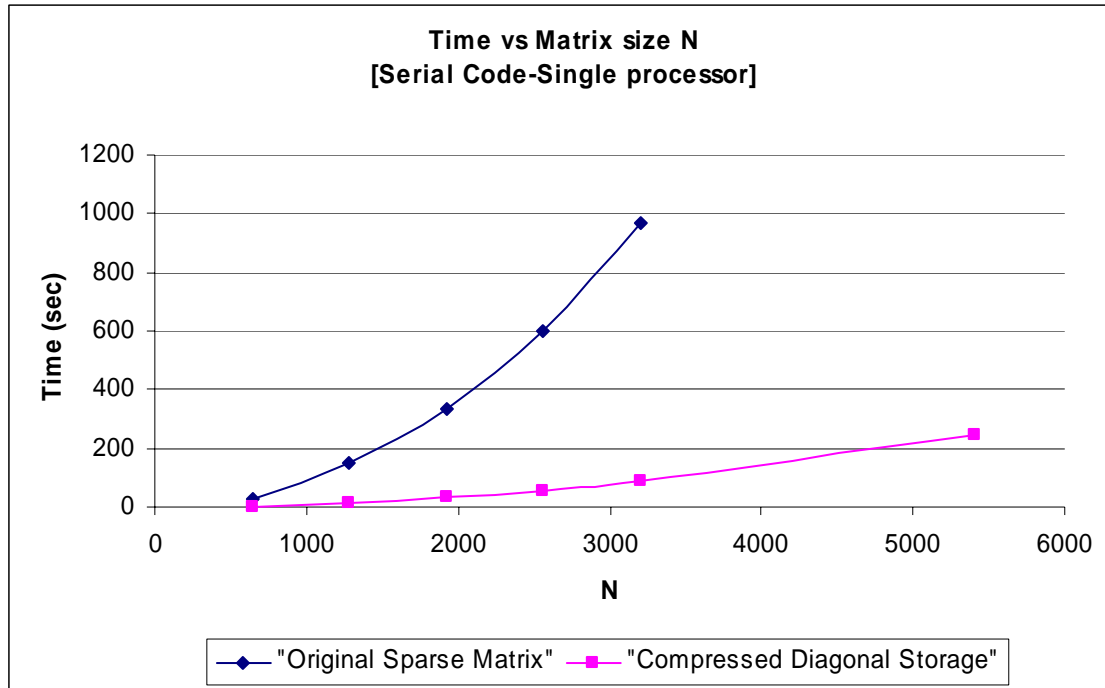


The sparse storage scheme described above was successfully implemented in the CG solver code that the author wrote in homework #2. The codes that utilize this sparse storage scheme, both in serial and in parallel format, are included for reference.

Performance Study:

Case 1: Serial code –single processor.

The time taken to run the matrix vector multiplication in the CG code that we wrote in homework 2 is greatly reduced once the proposed sparse storage scheme is used. The result is shown in the following figure.



Case 2: Parallel Code – Multiple Processors.

The parallel version of the CG code used in homework 2 was also modified to utilize the compressed diagonal sparse storage scheme in the matrix vector multiplication. The result shows a reduction in computational time at the mean time it allow a larger matrix size to be use – the maximum matrix size that can be used in the original storage scheme only allow a matrix size of roughly 3200x3200, but with the new storage scheme, the matrix size are able to goes up to 5400x 5400. The new storage scheme also shows a reduction in computational time in performing matrix vector multiplication. These result were shown in following two case studies: One with 4 processors and one with 8 processors being utilized.

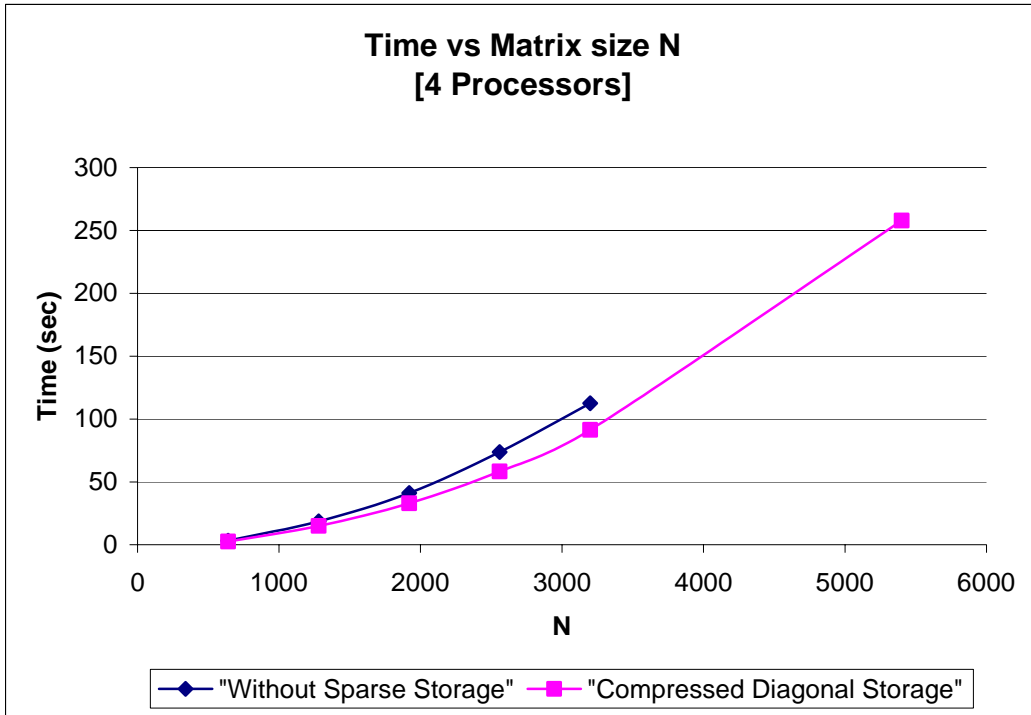


Figure 2: Performance gain with compressed diagonal storage scheme, utilized 4 processors.

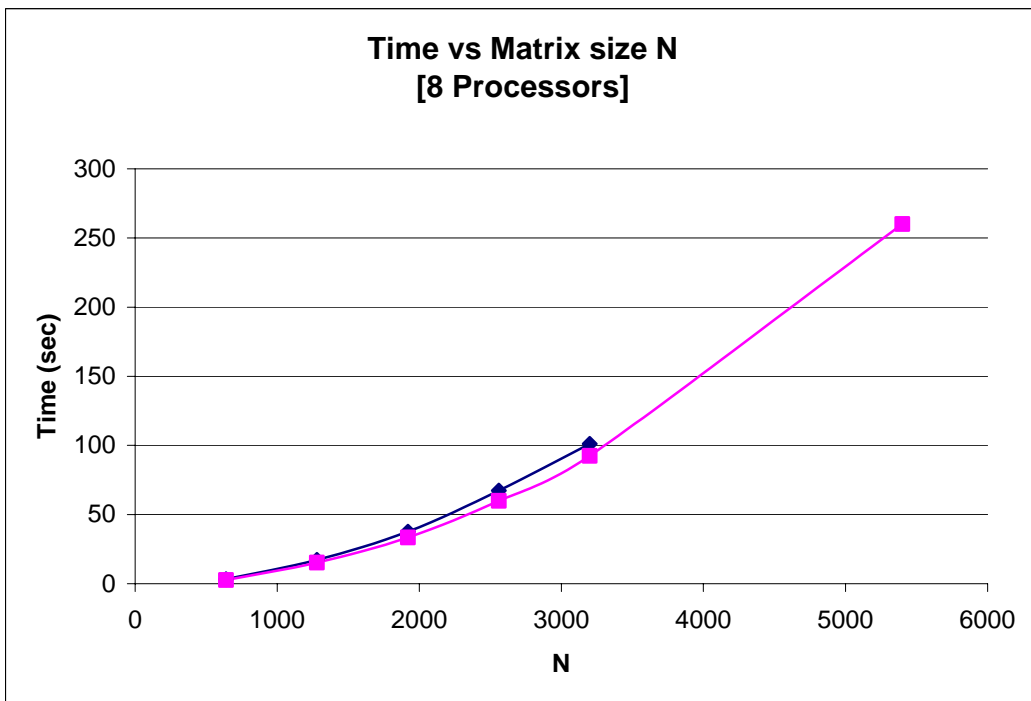


Figure 3: Performance gain with compressed diagonal storage scheme, utilized 8 processors.

PROBLEM 3. Instrument the code using the Performance API and illustrate single processor performance in terms of MFLOPs attained, cache efficiency and TLB misses.

Solution: The code that being used in this study is the serial code that computes the Reaction-Diffusion that wrote in homework #3 and #4. We study the performance of single processor performance for two cases: 1 with grid size 32x32 and the other with grid size 128 x 128. The performance studies were performed using Performance API (PAPI).

Summary of results:

	Grid Size: 32 x 32	Grid Size: 128 x 128
Processor Clock Rate	1.4×10^9 Hz	1.4×10^9 Hz
Processor Cycles	8.456×10^9	1.1×10^{12}
L1 cache missed	36396	3349802
L2 cache missed	2767	388741638
L3 cache missed	870	31937
Flops per sec	4.17×10^9	5.42×10^{11}
Performance	690.4 MFLOPS	689.8 MFLOPS
L1 cache efficiency	99.99%	99.99%
L2 cache efficiency	99.99%	99.82%
L3 cache efficiency	99.99%	99.99%

Formula used:

$$\text{Time} = \frac{\text{Process Cycle}}{\text{Processor Speed}} \quad (1)$$

$$\text{Performance} = \frac{\text{Floating Points Operations}}{\text{Time}} \quad (2)$$

$$\text{Cache Efficiency} = \left\{ 1 - \frac{\# \text{ of cache missed}}{\# \text{ of cache accessed}} \right\} \times 100\% \quad (3)$$

Data utilized in calculation:

L1 cache latency: 1 cycles
 L2 cache latency: 5-6 cycles
 L3 cache latency: 14 cycles

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

Screenshot for case 1: Grid size of 32 x 32:

PapiEx Version:	0.99rc2
Executable:	/san3/user/llee3/lennon/RDSerial
Processor:	Itanium 2
Clockrate:	1400.000000
Parent Process ID:	15562
Process ID:	15563
Hostname:	lennon.ccr.buffalo.edu
Options:	PAPI_L1_TCM,PAPI_L2_TCM,PAPI_L3_TCM
Start:	Fri Dec 16 22:20:13 2005
Finish:	Fri Dec 16 22:20:19 2005
Domain:	User
Real usecs:	6040136
Real cycles:	8456188720
Proc usecs:	6022896
Proc cycles:	8432054400
PAPI_L1_TCM:	36396
PAPI_L2_TCM:	2767
PAPI_L3_TCM:	870
PAPI_TOT_CYC:	8420584456
PAPI_FP_OPS:	4168936420

Screenshot of Case 2: Grid size of 128 x 128

CPU time used is 786.672592	
PapiEx Version:	0.99rc2
Executable:	/san3/user/llee3/lennon/RDSerial
Processor:	Itanium 2
Clockrate:	1400.000000
Parent Process ID:	17417
Process ID:	17418
Hostname:	lennon.ccr.buffalo.edu
Options:	PAPI_L1_TCM,PAPI_L2_TCM,PAPI_L3_TCM
Start:	Fri Dec 16 22:46:36 2005
Finish:	Fri Dec 16 22:59:44 2005
Domain:	User
Real usecs:	787750805
Real cycles:	1102851125320
Proc usecs:	786672592
Proc cycles:	1101341628800
PAPI_L1_TCM:	3349802
PAPI_L2_TCM:	388741638
PAPI_L3_TCM:	31937
PAPI_TOT_CYC:	1101352033975
PAPI_FP_OPS:	541844798620

SUMMARY AND CONCLUSION:

The author tried his best to complete this homework but nonetheless he failed to perform the performance analysis on problem 1. The code for problem 1 was successfully compiled, run, and gives the correct result. However, it does not give the performance as expected.

Problem 2 and problem 3 was completed. In problem 2, we can see that the both performance and storage size are improved when the new storage scheme (compressed diagonal storage) was implemented in the code. The computing time is improved tremendously in the serial implementation, but only improved slightly (about 10%) in the parallel setting. However, the storage ability was improved tremendously with the new storage scheme. This is because the original storage needs a size $N \times N$ but the new storage scheme only needs a size of $3 \times N$. In problem 3, PAPI was used to perform the required analysis and it was successfully done.

The code written for all the problem was included for references.

```

/* -----
Name: Leng-Feng Lee
Advisor: Dr. Patra & Dr. Jones
Course: MAE609-High Performance Computing
Description: Solve a Reaction-Diffusion Problem by using
            using Gauss-Seidal Method.
Created on: 29 October 2005.
Last Modify: 30 November 2005.
Note: This is the parallel Code-Using OpenMP.
-----*/

#include <stdio.h>
/*#include <iomanip.h>*/
#include <math.h>
#include <time.h>
/*#include <fstream.h>*/
#include <omp.h>

#define N 256 /* Number of grid used */

double C_current[N+2][N+2];
double C_previous[N+2][N+2];

main(int argc, char** argv)
{
    /* Initialization for performing Reaction Difussion Iteration: */
    int i, j;
    double x[N+2],y[N+2],xa,ya;
    double Length = 1.0; /* Length of the grid */
    double h = Length/(N+1); /* Spacing of each grid */
    double a = 1.0; /* R-D parameters */
    double b = 1.0; /* R-D parameters */
    double R = 0.0; /* R-D parameters */
    double e = 0.000001; /* Tolerance for convergence */
    double del_T = 1.0*h*h/(2*a*a); /* Define Time Step - ensured stability */
    double T = 0.0; /* Time T */
    double T_end = 0.1; /* Time stepping end time */
    int go = 1; /* Control value for the while-loop*/
    int iter = 1; /* Num of iterations in each time-step*/
    double M, K; /* Temporary Storage */

    /* Initialization use for Parallel processing */

```

```

int p_rank;                /* Processor ID */
int p;                     /* Total No. of processors used */
int local_N =0;           /* Size of matrix in each Processors */
/* int source=0;          */ /* where the message come from */
/* int tag=0;             */ /* Tag of the message passing -Not used*/
/* MPI_Status status;    */ /* used for message passing */
double time0, time1;      /* used for timing */

/*          local_N = N/p;          */ /* #rows assigned to each processor */
/*double local_Check,local_Check_temp; */ /* Check value for Local processor convergence*/
double Check_temp, Check=0.0; /* Check value for Global convergence */
/*double local_C_current[local_N+2][N+2];*/ /* local C_current matrix use for Gauss-Seidal
Iteration*/
/*double local_C_previous[local_N+2][N+2];*/ /* local C_previous matrix use for Gauss-Seidal
Iteration*/
/*double temp_C_row[N+2];          */ /* used for storing boundary value */
/*double temp_C[local_N+2][N+2];   */ /* used for storing final C_current matrix */

/* Initialization for file output */
/*ofstream RDfile;*/              /* Create output file object */
/*RDfile.open("RDdata.dat");*/    /* Output file name is "RDdata.dat" */

/* ----- Data Initialization -----*/
for (i=0; i<=N+1; i++)           /*Initialize x & y vector*/
{
    x[i]=h*(i);
    y[i]=h*(i);
}

/* --- The grid was labeled from 0 -> (N+1)~> for total size of (N+2),
      where at 0 and (N+1) is the boundary condition, which always maintained at zero.*/
for (i=0; i<=N+1; i++)          /*Initialize the grid*/
{
    for(j=0; j<=N+1; j++)
    {
        C_current[i][j] = 0.0;
        C_previous[i][j] = 0.0;

        xa = h*((double)i);
        ya = h*((double)j);
    }
}

```

```

        if ((xa>=0.2 && xa<=0.8) && (ya>=0.2 && ya<=0.3 | ya>=0.7 && ya<=0.8))
            C_previous[i][j]=1.0;
        else if ((ya>=0.2 && ya<=0.8) && (xa>=0.2 && xa<=0.3 | xa>=0.7 && xa<=0.8))
            C_previous[i][j]=1.0;
    }
}
time0 = omp_get_wtime();          /* Start Timing */
/* ----- Starts Gauss Seidel Iteration -----*/
for (T=0.0; T<=T_end; T=T+del_T) /*Start Time looping*/
{
    /* Start OpenMP */
    #pragma omp parallel shared(e,C_current, C_previous) private
(Check,T,T_end,del_T,i,M,K,Check_temp,p, p_rank)

    /* Get the thread ID */
    p_rank = omp_get_thread_num();

    /* Get total Number of processors used */
    p = omp_get_num_threads();

    go=1;
    while (go==1)
    {
        /* --- Each processor calculate N/p number of rows ---*/
        #pragma omp for private(i, j) schedule(static)
        for (i=1; i<=N; i++) /* row #0 & row #(local_N+1) are boundaries */
        {
            for (j=1; j<=N; j++) /* column #0 & column #(N+1) are boundaries */
            {
                M = ( (1/del_T) + ((4*a*a)/(h*h)) - b);
                if (iter==1)
                    K = 0.5;
                else
                    K = a*a*(C_current[i+1][j]+C_current[i-1][j]+C_current[i][j+1]+C_current[i][j-1])/
(h*h);

                C_current[i][j] = (1/M)*(K + (1/del_T)*C_previous[i][j] + R);
            }
        }
    }
    #pragma omp for private(i, j) schedule(static)

```

```

/* ----- Now calculate the residue for convergence on each processor----*/
for (i=1; i<=N; i++)
{
  for (j=1; j<=N; j++)
  {
    M = ( (1/del_T) + ((4*a*a)/(h*h)) - b);
    K = a*a*(C_current[i+1][j]+C_current[i-1][j]+C_current[i][j+1]+C_current[i][j-1])/(h*h);
    Check_temp = C_current[i][j] - (1/M)*(K + (1/del_T)*C_previous[i][j] + R);

    if (Check_temp < 0)                                /* Make sure the error is positive */
      Check_temp = -Check_temp;
    if (i==1 && j==1)                                  /* Set the first error as reference */
      Check = Check_temp;
    if (Check_temp > Check)                            /* Find the maximum error for convergence test*/
      Check = Check_temp;
  }
}

iter=iter+1;
if (Check < e )          /* Check for convergence*/
{
  iter=1;                /* Reset the iteration for the next time-step*/
  go=2;                  /* To break from the while() loop */
  /* Assign the current C value to the previous C value */
  for (i=0; i<=N+1; i++)
  {
    for (j=0; j<=N+1; j++)
    {
      C_previous[i][j]=C_current[i][j];
    }
  }
}

} /*End while (space) loop*/

/*End Parallelization*/

} /*End time loop*/

time1 = omp_get_wtime();          /* End Timing */

```

```

printf("\n\nTime used for this process is:%f on processor %d\n ", time1-time0,p_rank);
printf("\n\nNo of processors used in this process is %d\n\n ", p);

/* --- Set the format flag for file output ----*/

/* --- At the end of simulation, all processors send their blocks to processor #0.
Processor #0 combine the C_current matrix and save in file "RDdata.dat". ----- */
if (T >= T_end && p_rank == 0)
{
    /* ----PRINT ON SCREEN: Print the Final C_current matrix ----- */
    /*printf("FINAL C_Current matrix is: \n");
    for (i=0; i<=N+1; i++)
    {
        for (j=0; j<=N+1; j++)
        {
            printf("%f  ", C_current[i][j]);
        }
        printf("\n");
    }*/

} /* End if (T>T_end) statement */
return 0;

} /*End of main()*/

```

```

/* -----
Name: Leng-Feng Lee
Advisor: Dr. Patra & Dr. Jones
Course: MAE609-High Performance Computing
Description: Solve a Reaction-Diffusion Problem by using
            using Gauss-Seidal Method.
Created on: 6 December 2005.
Last Modify: 15 December 2005.
Note: This is the parallel Code - Using both MPI and OpenMP.
-----*/

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

#define N 12    /* Number of grid used */

double C_current[N+2][N+2];
double C_previous[N+2][N+2];

main(int argc, char** argv)
{
    /* Initialization for performing Reaction Difussion Iteration: */
    int i, j;
    double x[N+2],y[N+2],xa,ya;
    double Length = 1.0;          /* Length of the grid */
    double h = Length/(N+1);     /* Spacing of each grid */
    double a = 1.0;              /* R-D parameters */
    double b = 1.0;              /* R-D parameters */
    double R = 0.0;              /* R-D parameters */
    double e = 0.000001;         /* Tolerance for convergence */
    double del_T = 1.0*h*h/(2*a*a); /* Define Time Step - ensured stability */
    double T = 0.0;              /* Time T */
    double T_end = 0.1;          /* Time stepping end time */
    int go = 1;                  /* Control value for the while-loop*/
    int iter = 1;                /* Num of iterations in each time-step*/
    double M, K;                 /* Temporary Storage */

    /* Initialization use for Parallel processing */

```

```

int p_rank;                /* Processor ID */
int p;                    /* Total No. of processors used */
int local_N =0;          /* Size of matrix in each Processors */
int source=0;            /* where the message come from */
int tag=0;                /* Tag of the message passing -Not used*/
MPI_Status status;       /* used for message passing */
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;                /* #rows assigned to each processor */
double local_Check, local_Check_temp; /* Check value for Local processor convergence*/
double temp_Check, Check=0.0;        /* Check value for Global convergence */
double local_C_current[local_N+2][N+2]; /* local C_current matrix use for Gauss-Seidal
Iteration*/
double local_C_previous[local_N+2][N+2]; /* local C_previous matrix use for Gauss-Seidal
Iteration*/
double temp_C_row[N+2];                /* used for storing boundary value */
double temp_C[local_N+2][N+2];        /* used for storing final C_current matrix */

/* Initialization for file output */
/*ofstream RDfile;*/                    /* Create output file object */
/*RDfile.open("RDdata.dat");*/          /* Output file name is "RDdata.dat" */

/* ----- Data Initialization -----*/
for (i=0; i<=N+1; i++)                 /*Initialize x & y vector*/
{
    x[i]=h*(i);
    y[i]=h*(i);
}

/* --- The grid was labeled from 0 -> (N+1)~> for total size of (N+2),
        where at 0 and (N+1) is the boundary condition, which always maintained at zero.*/
for (i=0; i<=N+1; i++)                 /*Initialize the grid*/
{

```

```

for(j=0; j<=N+1; j++)
{
  C_current[i][j] = 0.0;
  C_previous[i][j] = 0.0;

  xa = h*((double)i);
  ya = h*((double)j);

  if ((xa>=0.2 && xa<=0.8) && (ya>=0.2 && ya<=0.3 | ya>=0.7 && ya<=0.8))
    C_previous[i][j]=1.0;
  else if ((ya>=0.2 && ya<=0.8) && (xa>=0.2 && xa<=0.3 | xa>=0.7 && xa<=0.8))
    C_previous[i][j]=1.0;
}
}

/* --- Initialize the local initial condition C matrix on each processor,
the C matrix is being divided in p pieces (row-wise), p=number of processor.
However, since boundary value are needed for each processor, the row at the
boundary is shared. ---*/
/* --- Note, that local_C_previous[][] in each processor is different;
also, local_C_current[][] in each processor is different. */
for (i=0; i<=local_N+1; i++)
{
  for (j=0; j<=N+1; j++)
  {
    local_C_current[i][j] = 0.0;
    local_C_previous[i][j] = C_previous[i+p_rank*local_N][j];
  }
}

time0 = MPI_Wtime(); /* Start Timing */

/* ----- Starts Gauss Seidel Iteration -----*/
for (T=0.0; T<=T_end; T=T+del_T) /*Start Time looping*/
{
  /* ---- Start OpenMP ---- */
  #pragma omp parallel shared(e,local_C_current, local_C_previous,local_Check) private
(T,j,T_end,del_T,Check,i,M,K, local_Check_temp,p, p_rank)
  go=1;
  while (go==1)
  {
    /* --- Each processor calculate N/p number of rows ---*/
    /* --Added Threads on each processor to improve performance --*/

```

```

#pragma omp for private(i,j) schedule(static)

for (i=1; i<=local_N; i++) /* row #0 & row #(local_N+1) are boundaries */
{
  for (j=1; j<=N; j++) /* column #0 & column #(N+1) are boundaries */
  {
    M = ( (1/del_T) + ((4*a*a)/(h*h)) - b);
    if (iter==1)
      K = 0.5;
    else
      K = a*a*(local_C_current[i+1][j]+local_C_current[i-1][j]+local_C_current[i][j+1]
+local_C_current[i][j-1])/(h*h);

    local_C_current[i][j] = (1/M)*(K + (1/del_T)*local_C_previous[i][j] + R);
  }
}
//#pragma omp critical
/* ----- Now calculate the residue for convergence on each processor----*/
#pragma omp for private(i, j) schedule(static)

for (i=1; i<=local_N; i++)
{
  for (j=1; j<=N; j++)
  {
    M = ( (1/del_T) + ((4*a*a)/(h*h)) - b);
    K = a*a*(local_C_current[i+1][j]+local_C_current[i-1][j]+local_C_current[i][j+1]
+local_C_current[i][j-1])/(h*h);
    local_Check_temp = local_C_current[i][j] - (1/M)*(K + (1/del_T)*local_C_previous[i][j] +
R);

    if (local_Check_temp < 0) /* Make sure the error is positive */
      local_Check_temp = -local_Check_temp;
    if (i==1 && j==1) /* Set the first error as reference */
      local_Check = local_Check_temp;
    if (local_Check_temp > local_Check) /* Find the maximum error for convergence
test*/
      local_Check = local_Check_temp;
  }
}

/* ---- For each iteration, Exchange the updated boudary value on the processor that share the
same boundary. ----*/

```

```

/* ---- STEP 1: Pass the updated boundary row to processors BELOW IT---*/
if (p_rank < (p-1))
{
    MPI_Send(local_C_current[local_N],(N+2),MPI_DOUBLE, p_rank+1, tag, MPI_COMM_WORLD);
}

/* ---- STEP 2: Receive the updated boundary value from processor above and assign to
appropriate row in C_previous[]*/
if (p_rank > 0)
{
    MPI_Recv(temp_C_row, N+2, MPI_DOUBLE, p_rank-1, tag, MPI_COMM_WORLD, &status);
    for (j=0; j<=N+1; j++) local_C_current[0][j]=temp_C_row[j];
}

/* ---- STEP 3: Now pass the boundary value to the processor ABOVE it ---*/
if (p_rank > 0)
{
    MPI_Send(local_C_current[1], N+2,MPI_DOUBLE, p_rank-1, tag, MPI_COMM_WORLD);
}

/* ---- STEP 4: Receive the updated boundary value from processor below and assign to
appropriate row in C_previous[]*/
if (p_rank < (p-1))
{
    MPI_Recv(temp_C_row, N+2, MPI_DOUBLE, p_rank+1, tag, MPI_COMM_WORLD, &status);
    for (j=0; j<=N+1; j++) local_C_current[local_N+1][j]=temp_C_row[j];
}
/* ---- End of update Boundary value in C_Previous[] -----*/

/* ---- Now send check value to processor #0, processor #0 find the largest check value,
then broadcast the check value for convergence check ---*/

MPI_Reduce(&local_Check, &Check, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

MPI_Bcast(&Check, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

iter=iter+1;
if (Check < e )          /* Check for convergence*/
{
    iter=1;              /* Reset the iteration for the next time-step*/
    go=2;                /* To break from the while() loop */
    /* Assign the current C value to the previous C value */
    for (i=0; i<=local_N+1; i++)
    {

```

```

        for (j=0; j<=N+1; j++)
        {
            local_C_previous[i][j]=local_C_current[i][j];
        }
    }
} /*End while (space) loop*/

} /*End time loop*/
/* --- End OpenMP Threads -----*/

time1 = MPI_Wtime();      /* End Timing */

/* --- Set the format flag for file output ----*/
/*RDfile.setf(ios::fixed);*/

/* --- At the end of simulation, all processors send their blocks to processor #0.
Processor #0 combine the C_current matrix and save in file "RDdata.dat". ----- */
if (T >= T_end )
{
    if (p_rank > 0) /* All other processors send the final local_C_current to processor #0*/
        MPI_Send(&local_C_current, (local_N+2)*(N+2), MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    else /* Processor #0 */
    {
        for (i=1; i<=local_N; i++) /* Processor #0 put its own computed local_C_current to
C_current*/
        {
            for (j=1; j<=N; j++)
                C_current[i][j]=local_C_current[i][j];
        }

        for (source=1; source<p; source++) /* Received the local_C_current from other processor and
place it in C_current*/
        {
            MPI_Recv(&temp_C, (local_N+2)*(N+2), MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
            for (i=1; i<=local_N; i++)
            {
                for (j=1; j<=N; j++)
                    C_current[i+source*local_N][j] = temp_C[i][j];
            }
        }
    }
}

```

```

/* ----PRINT ON SCREEN: Print the Final C_current matrix ---- */
printf("FINAL C_Current matrix is: \n");
/* for (i=0; i<=N+1; i++)
{
    for (j=0; j<=N+1; j++)
    {
        printf("%f  ", C_current[i][j]);
    }
    printf("\n");
}*/

/*for (i=0; i<= N+1; i++)
{
    for(j=0; j<=N+1; j++)
    {
        RDfile << C_current[i][j] << setw(16);
    }
    RDfile << endl;
}*/

} /* End else statement */
} /* End if (T>T_end) statement */

printf("\n\nTime used for this process is:%f on processor %d\n ", timel-time0,p_rank);
printf("\n\nNo of processors used in this process is %d\n\n ", p);

MPI_Finalize();
return 0;

} /*End of main()*/

```

```

/* -----
Name: Leng-Feng Lee
Course: High Performance Computing I
Instructor: Dr Patra
Description: Solve  $A_p = B$  equation using preconditioned
            Conjugate Gradient Method.

Last Modify: 19 DECEMBER 2005.

CODE FOR HOMEWORK 5 PROBLEM 2: CG-METHOD USING NEW SPARSE MATRIX
STORAGE SCHEME-COMPRESSED DIAGONAL STORAGE- SERIAL CODE.

-----*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#define N 1280

int i, j, k;
double alpha=0.0, beta=0.0;
double A[3][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<3;j++)
            A[j][i]=0.0;
    }
}

```

```

/* Original Sparse matrix */
/*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;
      }
    }
  }
}
*/

/* New sparse storage matrix A*/
A[0][0]=0;
A[1][0]=2;
A[2][0]=-1;
A[0][N-1]=-1;
A[1][N-1]=2;
A[2][N-1]=0;
for (j=1; j<N-1; j++)
{
  A[0][j]=-1;
  A[1][j]=2;
  A[2][j]=-1;
}
/*for (i=0;i<3;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("The D vector is");
for (i=0; i<N; i++) printf(" %f  \n",D[i]);*/

/*printf("inner1 is %f\n\n",inner1);
printf("\n\ninner2 is %f",inner2);*/

/*----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]);*/

  /* Here is the Matrix-Vector Multiplication */

```

```

for (j=0; j<N; j++)
{
    for (i=0; i<3; i++)
    {
        W[j]=A[i][j]*D[j+i-1]+W[j];
    }
}

/*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  $A_p = B$  equation using preconditioned
            Conjugate Gradient Method.
Note: This is the Parallel solution for Homework #5.
Last Modify: 15 December 2005.
Note:
1. This code compiled successfully on u2.ccr.buffalo.edu
2. Use N number of processor to compile, N is the number of the A
   matrix (NxN) size.

CODE FOR HOMEWORK 5 PROBLEM 2: CG-METHOD USING NEW SPARSE MATRIX
STORAGE SCHEME-COMPRESSED DIAGONAL STORAGE- PARALLEL CODE.
-----*/

```

```

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

#define N 12

double A[3][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<3; j++)
        A[j][i]=0.0;
}

/* New sparse storage matrix A*/
A[0][0]=0;
A[1][0]=2;
A[2][0]=-1;
A[0][N-1]=-1;
A[1][N-1]=2;
A[2][N-1]=0;
for (j=1; j<N-1; j++)
{
    A[0][j]=-1;
    A[1][j]=2;
    A[2][j]=-1;
}
/*----- 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-middle row 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;*/                               /* size of rows each processor will process */
  /*local_i_ini  = p_rank*local_N;*/                     /* starting row for each processor */
  /*local_i_max  = p_rank*local_N + local_N;*/           /* ending row for each processor */
  /*local_jmax   = N;*/                                   /* each column hav the same size as original NxN*/
  for (i=0;i<local_N;i++) local_W[i] = 0;               /* reinitialize for each iteration */

  /* Local Matrix-vector Multiplication -Sparse Storage Scheme */
  for (j = 0; j < local_N; j++)
  {
    for (i=0; i<3; i++)
    {
      local_W[j] =A[i][j+p_rank*local_N]*D[j+i+p_rank*local_N-1]+local_W[j];
    }
    /*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;
}

```