HW5: 50/50 very nicely done

Homework 4 – Mohammad Atif Faiz Afzal

HPC1

Date- 11/11/2014

Problem 1:

The aim is solve the Laplace equation in 2 dimensions

Laplace equation in dimension is given as

$$\Delta^2 \emptyset = \left(\frac{\partial^2}{\partial\, x^2} + \frac{\partial^2}{\partial\, y^2}\right) \emptyset = 0$$

The above Laplace can be solved numerically using 2nd order central difference approximation as shown below

$$\frac{\partial^2 \emptyset}{\partial\, x^2} = \frac{\emptyset(x+h) - 2\emptyset(x) + \emptyset(x-h)}{h^2}$$

Considering a uniform square grid of M points and using the above equation, the Laplace equation can be written as (i and j represent the position of the node on the grid of size M)

$$\frac{\partial^2 \emptyset}{\partial\, x^2} + \frac{\partial^2 \emptyset}{\partial\, y^2} = \frac{\emptyset(i+1,j) - 2\emptyset(i,j) + \emptyset(i-1,j)}{h^2} + \frac{\emptyset(i,j+1) - 2\emptyset(i,j) + \emptyset(i,j-1)}{h^2} = 0$$

$$\emptyset(i+1,j) - 2\emptyset(i,j) + \emptyset(i-1,j) + \emptyset(i,j+1) - 2\emptyset(i,j) + \emptyset(i,j-1) = 0$$

$$\emptyset(i,j) = \frac{\emptyset(i+1,j) + \emptyset(i-1,j) + \emptyset(i,j+1) + \emptyset(i,j-1)}{4}$$

In Relaxation method (Jacobi method), the above equation is solved by iteratively as shown below

$$\emptyset(i,j)^{n+1} = \frac{\emptyset(i+1,j)^n + \emptyset(i-1,j)^n + \emptyset(i,j+1)^n + \emptyset(i,j-1)^n}{4}$$

Where n is the iteration number

To reduce the number of iterations a modification called 'over relaxation' method [1] can also be used

$$\emptyset(i,j)^{n+1} = \left(\frac{\emptyset(i+1,j)^n + \emptyset(i-1,j)^n + \emptyset(i,j+1)^n + \emptyset(i,j-1)^n}{4}\right) w + (1-w)\emptyset(i,j)^n$$

Where w is a non-zero constant and should be optimized to get a better result.

Given boundary conditions are

$$\emptyset(x,0) = \sin(\pi x)$$

$$\emptyset(x,1) = \sin(\pi x) e^{-\pi}$$

$$\emptyset(0,y) = \emptyset(1,y) = 0$$

[1] http://bulldog2.redlands.edu/facultyfolder/deweerd/tutorials/Tutorial-RelaxationMethod.pdf

Part a

The serial code is written in C language and is attached in the Appendix 1. Intel (ICC) compiler is used for all the compiling done in this assignment.

To check if the code is working properly a 3D plot was generated. Figure 1 a and b shows the 3D plot generated for a grid of 10X10 points and 100X100 points. Figure 1 c shows the plot of the analytic solution for comparison (generated in Matlab and the code attached in Appendix 1).

good – verification

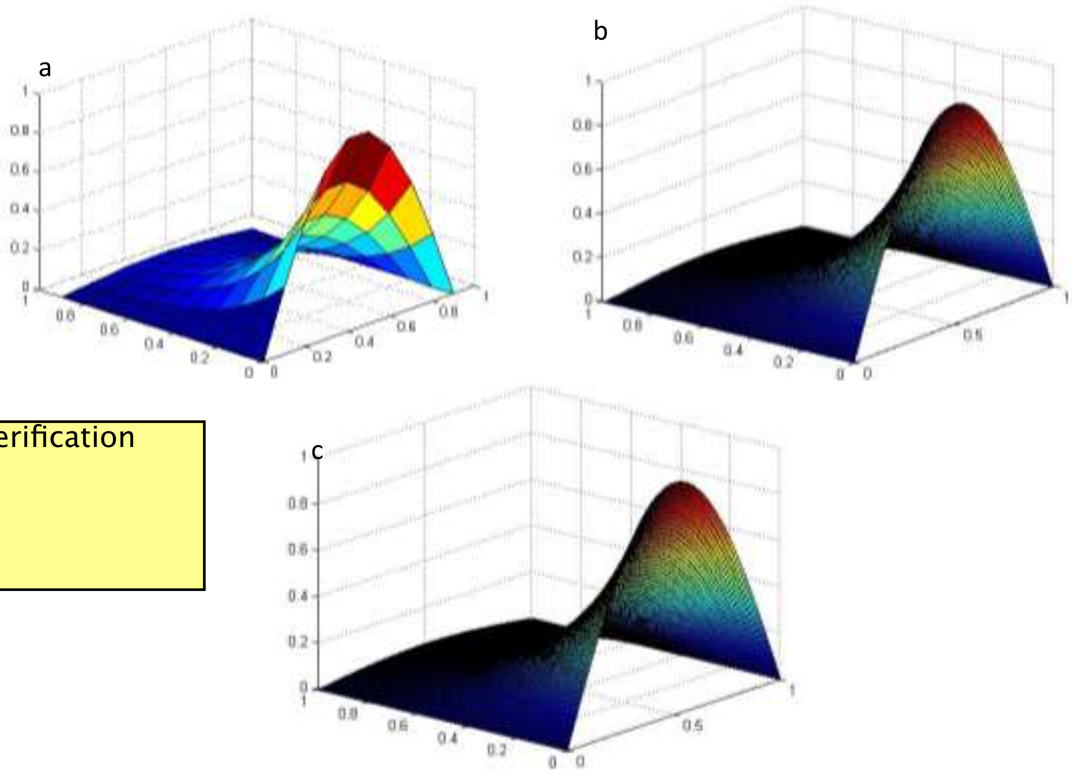

Figure 1: 3D plot for the solution of Laplace equation. Numerically solved using Jacobi iteration (a) generated for a grid of 10X10 points and (b) 100X100 points. (c) is analytic solution for comparison

Interesting observation. I copied the data points to a spread sheet and to my amuse I found that the data in the spreadsheet actually represents the 2D projection of the above plots. Check the figure 2 for the amusing pattern in the data in the spreadsheet.

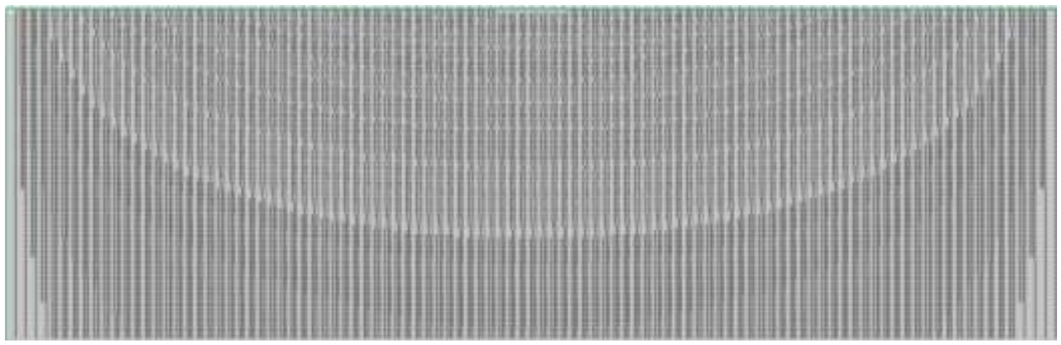

Figure 2: Pattern generated by the data points in the spreadsheet ☺

As a curiosity I changed the grid to a rectangular grid and solved iteratively. Apparently, the solutions I got do not agree with the actual solution. Thus, Jacobi iteration method does not work for non-square grids.

It does, actually – but nonuniform grids have to be factored into the relaxation equation, the one we are using above assumes a uniform grid size in both dimensions

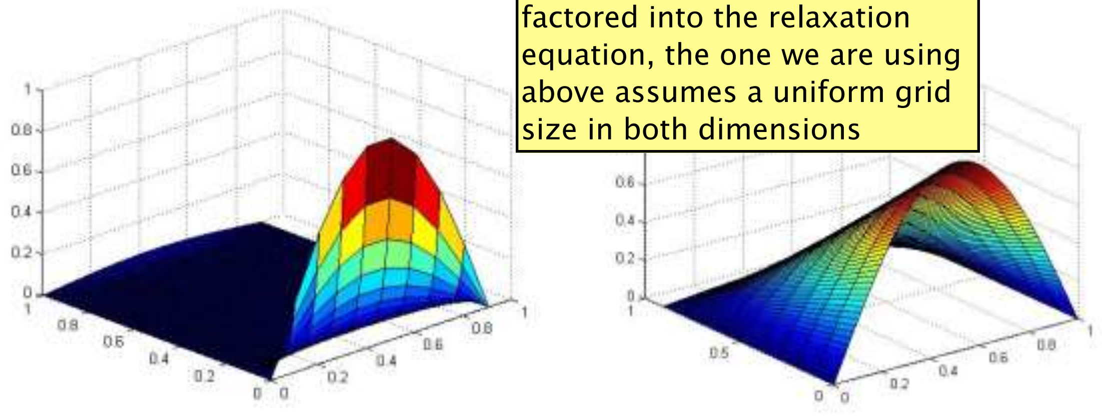

Figure 3: 3D plots generated for the solution on a rectangular grid of (a) 100X10 points and (b) 10X100 points

Part B

To plot the number of iterations that are required as a function of grid size, the grid size was varied from 10 to 640. The value of L2 norm is taken as 10^{-4}.

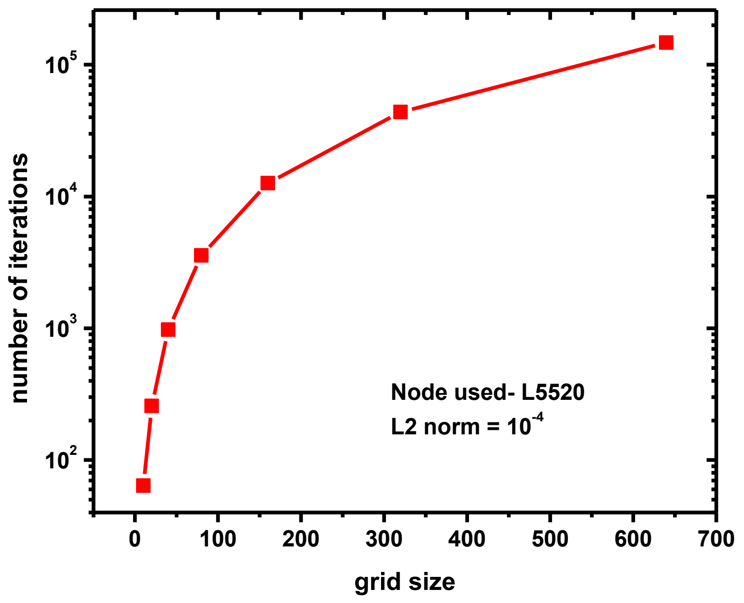

Figure 4: Number of iterations that are required as a function of grid size

Observations

1. It can be seen that the number of iterations vary non-linearly with the grid size.
2. In fact, the variation number of iterations is proportional to O(grid points squared)
3. The number of iterations were increasing when a smaller value of L2 norm was taken. For example, for a grid size of 200 and L2 norm=10^{-4} the number of iterations were 18923, whereas for the same grid size and l2 norm= 10^{-5} the number of iterations were 28162

Part C

The code for OpenMP implementation of the code is included in the Appendix 2. Two codes were written, one with a defined job distribution to the threads and other code where the decomposition is automatic. Both the codes work!

Part D

For this part, a 12 core node (E5645) available on CCR was used. The code was initially giving some abnormal results when all the 12 threads was being used, therefore Affinity option was used. Figure 4 shows the computation time as a function of no. of threads for a grid size of 1000X1000. The affinity used in this case is compact type.

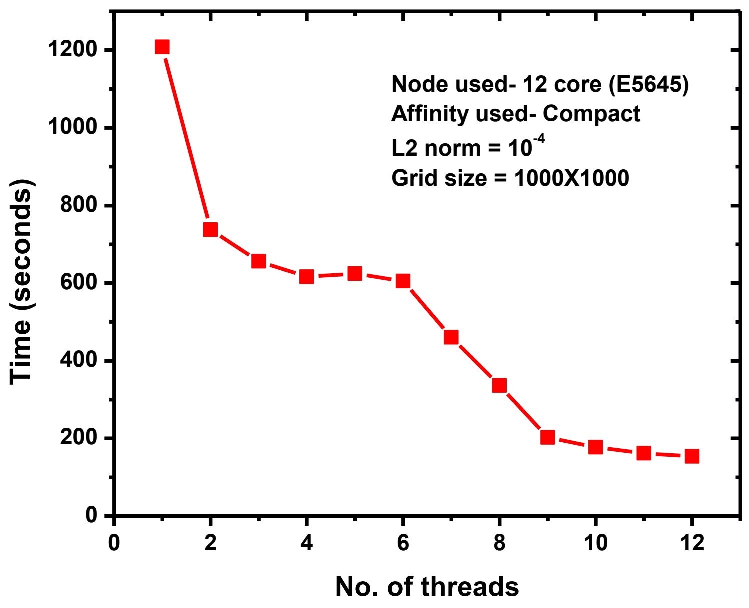

Figure 5. Computation time as a function of no. of threads for a grid size of 1000X1000

The code was also executed for 500X500 grid size (figure 6). This time two different affinity options: compact and scatter were implemented. It can be seen in the plot that the performance of compact affinity is not good when the no. of threads is 4-8. The is probably because when compact affinity is used the jobs are distributed in closer threads which have similar L3 cache which might be small for the big matrices that are being used. That is why scatter performance is better as in scatter affinity the jobs are distributed in threads having different L3 cache. It can also be seen in scatter affinity that the performance of even number of threads is better when compared to odd number of threads.

very good.

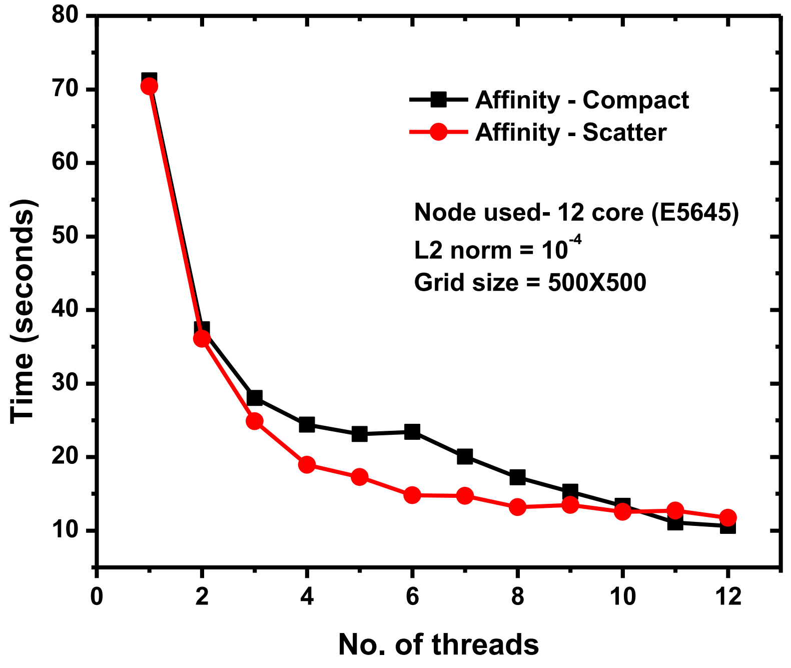

Figure 6. Computation time as a function of no. of threads for a grid size of 500X500

Parallel speedup is calculated by the formula

$$Parallel\ Speedup = \frac{Time\ for\ sequential\ running}{Time\ for\ running\ on\ N\ Threads}$$

For Sequential time, the serial code written in part 1 was used and executed on the same node (E5645). Parallel efficiency is calculated by dividing the parallel speedup with the no. of threads used.

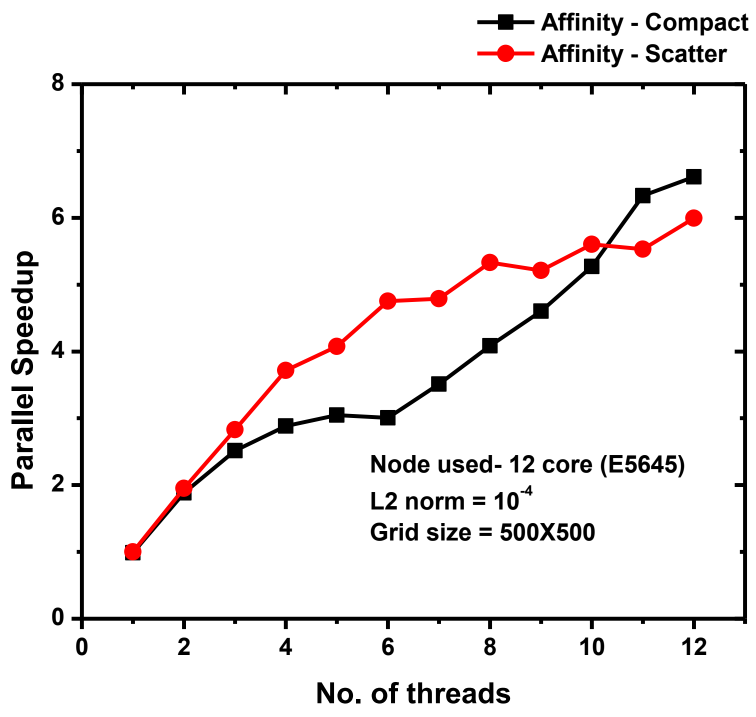

Figure 7. Parallel speedup as a function of no. of threads for a grid size of 500X500

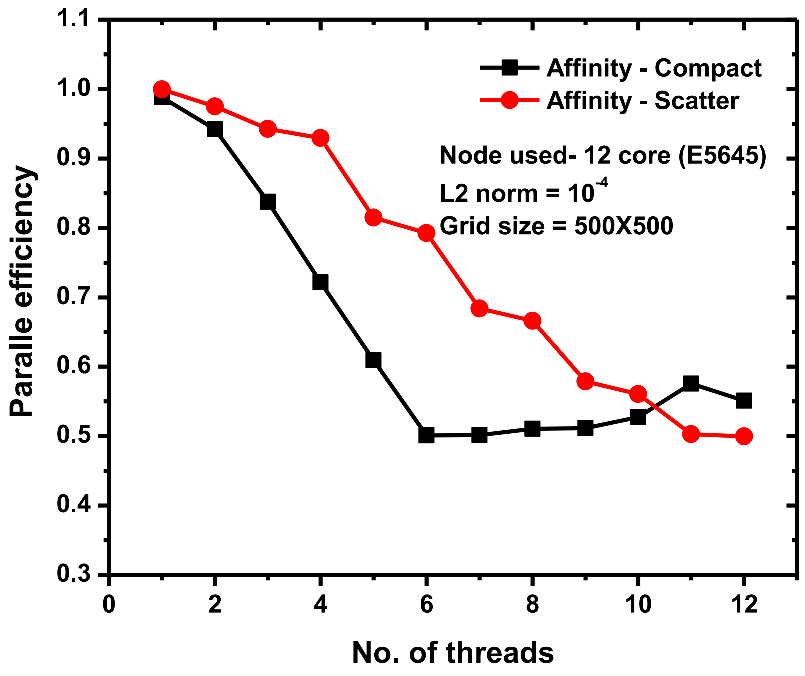

Figure 8. Parallel efficiency as a function of no. of threads for a grid size of 500X500

Appendix 1

C code for serial implementation of Jacobi iteration to solve the Laplace equation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

int main (int argc, char *argv[])
{
 int a,i,j,k,p,m,n,iter;
 iter=100000000;
 double sum,conv,b,r,w;
 w=.99;
 sum=0.0;
 struct timeval ti,tf;
 long double pi;
 pi= 3.14159265359;

 for (p=10;p<=100000;p=p*2)
  {
   m=p;n=p;
   double **fi_i = (double **) malloc(sizeof (double *) * m);
   double **fi_f = (double **) malloc(sizeof (double *) * m);

   for (i=0;i<m;i=i+1)
          {
           fi_i[i]=(double *) malloc(sizeof (double) * n);
           fi_f[i]=(double *) malloc(sizeof (double) * n);
          }
   /* boundary conditions and initializing all other grid points to zero value */

   for (i=0;i<m;i=i+1)
          {
           for (j=0;j<n;j=j+1)
            {
             fi_i[i][j]=0.0;
             fi_f[i][j]=0.0;

             if (i==0)
                {
                 fi_i[i][j]=sin(pi*(j/((double)n-1)));
                 fi_f[i][j]=sin(pi*(j/((double)n-1)));
                }
             if (i==(m-1))
                {
                 fi_i[i][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                 fi_f[i][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                }

             /* printf("%d\t",j); */
```

```
                /* printf("%f\n",fi_i[i][j]);  */
               }
             /* printf("%d\n",i); */
            }
gettimeofday(&ti,0);

for (k=1;k<=iter;k=k+1)
            {
             for (i=1;i<m-1;i=i+1)
               {
                for (j=1;j<n-1;j=j+1)
                     {
                      /* Relaxation method */
                      /* fi_f[i][j]=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0; */

                      /* using over relaxation method */
                      r=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0;
                      fi_f[i][j]=r*w+(1-w)*fi_i[i][j];

                      sum = sum + (fi_f[i][j]-fi_i[i][j])*(fi_f[i][j]-fi_i[i][j]);

                     }
               }

             /* assiging new fi values  */
             for (i=1;i<m-1;i=i+1)
               {
                for (j=1;j<n-1;j=j+1)
                     {
                      fi_i[i][j]=fi_f[i][j];
                     }
               }
             /* Calculating L2 norm */
             conv=sqrt(sum);

             if (conv < 0.0001)
               {
                break;
               }
             /* printf("Iteration number %d\t",k); */
             /* printf("convergence = %.10f\n",conv); */

             sum=0.0;
            }
gettimeofday(&tf,0);
double timeelapsed = ((tf.tv_sec - ti.tv_sec)*1000000 + (tf.tv_usec - ti.tv_usec))/1000;

printf(" %d\t",m);
printf(" %f\n",timeelapsed);

/* for (i=0;i<m;i=i+1) */
/*          { */
/*            for (j=0;j<n;j=j+1) */
```

```
        /*          { */
        /*            printf("%f\t",fi_i[i][j]); */

        /*          } */
        /*        printf("%c\n",' '); */
        /*      } */
        }
}
```

Matlab code used to generate 3D plots for analytic solution

```
[X,Y] = meshgrid(0:.01:1);
Z = sin(X .* pi).*exp(-Y.*pi);
surf(X,Y,Z)
```

Appendix 2

C code for OpenMP implementation for the code in part A. This code does a defined decomposition of jobs for the specific threads

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

int main (int argc, char *argv[])
{
 int i,j,k,m,n,iter;
 iter=2000000;
 m=500;n=500;
 double sum,conv,r,w;
 w=.99;
 sum=0.0;
 long double pi,ctime1,ctime2;
 pi= 3.14159265359;

 double **fi_i = (double **) malloc(sizeof (double *) * m);
 double **fi_f = (double **) malloc(sizeof (double *) * m);

 for (i=0;i<m;i=i+1)
  {
   fi_i[i]=(double *) malloc(sizeof (double) * n);
   fi_f[i]=(double *) malloc(sizeof (double) * n);
  }
 int nthreads;

 for (nthreads=1;nthreads<=12;nthreads++)
  {
   for (i=0;i<m;i=i+1)
          {
           for (j=0;j<n;j=j+1)
            {
             fi_i[i][j]=0.0;
             fi_f[i][j]=0.0;

             if (i==0)
                 {
                  fi_i[i][j]=sin(pi*(j/((double)n-1)));
                  fi_f[i][j]=sin(pi*(j/((double)n-1)));
                 }
             if (i==(m-1))
                 {
                  fi_i[i][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                  fi_f[i][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                 }
            }
```

```
                    }

        ctime1 = omp_get_wtime(); /* using OpemMP time */
        omp_set_num_threads(nthreads);

        for (k=1;k<=iter;k=k+1)
                    {
     #pragma omp parallel default(shared) private(i,j,r) reduction (+:sum)
                     {
                      const int tid = omp_get_thread_num();
                      const int a = (tid)*(m-2)/nthreads+1;
                      const int b = (tid+1)*(m-2)/nthreads+1;

                      for (i=a;i<b;i=i+1)
                        {
                            for (j=1;j<n-1;j=j+1)
                              {
                                /* fi_f[i][j]=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0; */

                                r=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0;
                                fi_f[i][j]=r*w+(1-w)*fi_i[i][j];

                                sum = sum + (fi_f[i][j]-fi_i[i][j])*(fi_f[i][j]-fi_i[i][j]);
                              }
                        }

                      for (i=a;i<b;i=i+1)
                        {
                            for (j=1;j<n-1;j=j+1)
                              {
                                fi_i[i][j]=fi_f[i][j];
                              }
                        }

                     }

                     conv=sqrt(sum);

                     if (conv < 0.0001)
                       {
                         break;
                       }
                     sum=0.0;
                    }
        ctime2 = omp_get_wtime();
        double time=(ctime2-ctime1)*1000; /* calculate time in milli seconds */

        printf("%d\t",nthreads);
        printf(" %f\n",time);

      }

}
```

C code for OpenMP implementation for the code in part A. This code does automatic decomposition of jobs for the specific threads

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

int main (int argc, char *argv[])
{
 int a,i,j,k,m,n,iter;
 iter=20000;
 m=200;n=200;
 double sum,conv,b,r,w;
 w=0.99;
 sum=0.0;
 long double pi,ctime1,ctime2;
 pi= 3.14159265359;

 double **fi_i = (double **) malloc(sizeof (double *) * m);
 double **fi_f = (double **) malloc(sizeof (double *) * m);

 for (i=0;i<m;i=i+1)
  {
   fi_i[i]=(double *) malloc(sizeof (double) * n);
   fi_f[i]=(double *) malloc(sizeof (double) * n);

  }
 int nthreads=4;

 for (i=0;i<m;i=i+1)
  {
   for (j=0;j<n;j=j+1)
           {
            fi_i[i][j]=0.0;
            fi_f[i][j]=0.0;

            if (i==0)
             {
              fi_i[i][j]=sin(pi*(j/((double)n-1)));
              fi_f[i][j]=sin(pi*(j/((double)n-1)));
             }
            if (i==(m-1))
             {
              fi_i[i][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
              fi_f[i][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
             }

           }
  }

 omp_set_num_threads(nthreads);

 ctime1 = omp_get_wtime(); /* using OpemMP time */
```

```
    for (k=1;k<=iter;k=k+1)
     {

  #pragma omp parallel default(shared) private(i,j,r)
       {

  #pragma omp for schedule (static) reduction (+:sum)
                    for (i=1;i<m-1;i=i+1)
                     {
                       for (j=1;j<n-1;j=j+1)
                         {
                             /* fi_f[i][j]=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0; */

                             r=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0;
                             fi_f[i][j]=r*w+(1-w)*fi_i[i][j];

                             sum = sum + (fi_f[i][j]-fi_i[i][j])*(fi_f[i][j]-fi_i[i][j]);
                         }
                     }

  #pragma omp for schedule (static)
        for (i=1;i<m-1;i=i+1)
                    {
                      for (j=1;j<n-1;j=j+1)
                        {
                          fi_i[i][j]=fi_f[i][j];
                        }
                    }
       }

       conv=sqrt(sum);
       printf("Iteration number %d\t",k);
       printf("convergence = %.10f\n",conv);

       if (conv < 0.0001)
                    {
                      break;
                    }

       sum=0.0;

     }
    ctime2 = omp_get_wtime();
    double time=(ctime2-ctime1)*1000; /* calculate time in milli seconds */

    printf("thread = %d\t",nthreads);
    printf("time=  %f\n",time);
}
```