Introduction

In this assignment, the Jacobi iteration method for solving a system of linear equation is going to be discussed. A parallel program will be introduced and compared with a sequential program to solve this type of equation using Jacobi method.

Problem Definition

The purpose of Jacobi iteration method is to solve the following system of linear equation:

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21x1} + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{array}$$

In this method, this system of equation would be solved by solving the following expression :

$$x_i = \frac{1}{a_{ii}}[b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij}x_j]; i=1,2,\dots,n$$

It starts with some initial values $x_i^{(0)}$ for x_i successively ; then the mentioned expression will be applied iteratively on x_i found in the previous stage; that is the x_is of step k+1 will be computed by substituting the x_is found in step k using the expression as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}[b_i - \sum_{j=1,j\neq i}^{n} a_{ij}x_j^{(k)}]; i=1,2,\dots,n$$

The termination condition for the iteration is satisfaction of the following expression which implies the convergence of the solution.

$$|x_i^{(k+1)} - x_i^{(k)}| \leq \varepsilon \, ; i = 0,1,\ldots,n$$

Where ε is a small number.

It must be considered that the Jacobi method does not converge for every problem; but it has been found that a sufficient condition for this method is as follows:

$$|a_{ii}| > \sum_{j=1,\, j \neq i}^{n} |a_{ij}|$$

As mentioned, this algorithm will be going to implemented in a sequential program as well as parallel program whose details is going to be discussed next.

<u>Implementation</u>

Sequential Implementation

The flow of the implemented sequential program is described in the following pseudocode in which the dimension of the problem(number of equations and variables x_i) is represented using variable *n*. The two dimensional array *a_b[][]* is used in this algorithm to store the coefficients a_{ij} as well as b_i(row *i* of the array *a_b* contains *n* coefficients of row *i* in the problem definition followed by b_i). The values of $x_i^{(k)}$ are stored in the array *Xi_old[]* and the values of $x_i^{(k+1)}$ are stored in the array *Xi_new[]*. Also, the variable *convergence_flag* is used for convergence check.

```
STEP 1: epsilon = 0.001;  convergence_flag = 0;   //used for convergence check
STEP 2: for (i=0; i<n; i++)
                Xi_old[i] = 0;                          //initialization
STEP 3: Repeat while (convergence_flag<n)
                convergence_flag = 0;
STEP 4:         for(i=0; i<n; i++)
                {
                        sum = 0;
STEP 5:                 for(j=0; j<n; j++)
                                sum += a_b[i][j]*Xi_old[j];
                        sum -= a_b[i][i]*Xi_old[i];

STEP 6:                 Xi_new[i] = (1/a_b[i][i])*(a_b[i][n]-sum);  //a_b[i][n] is  $b_i$

STEP 7:                 if( |Xi_old[i]-Xi_new[i]| <= epsilon )
                                convergence_flag++;
                }

STEP 8:         for(i=0; i<n; i++)
                        Xi_old[i] = Xi_new[i];
```

Time Complexity

The dominated step in this algorithm is STEP 5 which is executed n^2I times(there are ($n+n(n+3)+n)I = (n^2+5n)I$ *operations* in each iteration) where I is the number of iterations; therefor, assuming that the number of iterations is constant, this algorithm has a time complexity of $O(n^2)$ and is a polynomial time algorithm.

Parallel Implementation

To implement Jacobi iteration method in a parallel manner, C/OpenMPI was exploited. “n+1” number of processes was planned to be utilized where n is the dimension of the problem equation; n number of those processes, each having access to one row of the coefficients of the equation, computes one of the variables x_i in the same fashion the sequential algorithm would do. Since all the $x_i^{(k)}$ s are needed in iteration (k+1), each process will send its corresponding computed x_i to other processes. The pseudocode of the implemented program is introduced next. In this algorithm, the process 0 will read the input file and send the coefficients to the other processes; after finishing the computations, those processes will send back their final results to process 0 who will display them in turn(There are n+1 total number of processes needed). Here, *a_b_complete[][]* array is used by process “0” to store the coefficients and b_i . Each process has its own version of array *a_b[]* in which it store its corresponding coefficients. As same as the serial algorithm, *Xi_old[] and Xi_new[]* arrays are used by each process to store the result of iteration k and k+1 respectively. Also, *convergence_flag* is used to check the convergence criterion.

```
STEP 1: epsilon = 0.001;  convergence_flag = 0;   //used for convergence check

STEP 2: MPI initializations

STEP 3: if( this is process “0” ) do the followings

STEP 4:        read the “inputFile”and put its contents in a_b_complete[][]

STEP 5:        for (i=1; i<=n; i++)
                      MPI_Send(a_b_complete[i-1][] to process “i”);

STEP 6:        for (i=1; i<=n; i++)
                      MPI_Receive(Xi[i-1] from process “i”);
                      print(Xi[i-1]);                  //Final results


STEP 7: else , do the followings                       //Process 1 to n

STEP 8:        MPI_Receive( a_b[] from process “0”);

STEP 9:        for (i=0; i<n; i++)
                      Xi_old[i] = 0;                          //initialization

STEP10:        Repeat while (convergence_flag==0)

                      i = my_rank – 1;
                      sum = 0;
STEP11:               for(j=0; j<n; j++)
                             sum += a_b[j]*Xi_old[j];
STEP12:               sum -= a_b[i]*Xi_old[i];

STEP13:               Xi_new[i] = (1/a_b[i])*(a_b[n]-sum);

STEP14:               if( |Xi_old[i]-Xi_new[i]| <= epsilon )
                             convergence_flag = 1;
                      else
                             convergence_flag = 0;

STEP15:               Xi_old[i] = Xi_new[i];

STEP16:               MPI_Barrier(slave_processes /*process 1 to n*/); //waiting others
STEP17:               MPI_Allreduce(convergence_flag with MIN)//find the min of all
                                                               //    convergence_flag s.
STEP18:               for(j=1; j<=n; j++)
                              if(j != my_rank)
                                    MPI_Send(Xi_old[i] to process “j”);
STEP19:                             
STEP20:                             MPI_Receive(Xi_old[j-1] from process “j”);

STEP21:        MPI_Send(Xi_new[i] to process “0” );
```

Time Complexity

The execution time of this algorithm can be computed as follows:

Computation Time for processes 1 to n in STEPs 9, 11, 12, 13, 14, 15 respectively are:
$n+n+1+1+1+1 = 2n + 4$

Communication Time in STEPs (5,8), (6,21), 17, (19,20) respectively are:
$n(t_{start} + nt_{data}) + n(t_{start} + t_{data}) + n(t_{start} + t_{data})I + (n-1)(t_{start} + t_{data})I = (4n-1)t_{start} + (n^2+3n-1)t_{data}$

where I is the number of iterations.

Therefor, the *computation complexity* is $O(n)$ and the *communication complexity* is $O(n^2)$; which makes the overall complexity $O(n^2)$.

Comparing complexities of the sequential and parallel algorithms, we can imply that in terms of computation,

Speedup factor: $$\frac{t_s}{t_p} = \frac{n^2+5\text{n}}{(2\text{n}+4)+(4\text{n}-1)t_{start}+(n^2+3\text{n}-1)t_{data}}$$

Considering that in this algorithm we utilized n processes, the speedup factor, ignoring communication times, is almost n. However, we can not ignore communication time in real world; as in experiments the sequential program responded much faster that the parallel one.

Sequential Program

```
#include <stdio.h>

#define epsilon 0.0001

main(int argc, char* argv[])
{
     int n;
     float *Xi_old,*Xi_new;//Xi in each iteration
     short convergence_flag;

     int i,j;
     FILE *inputFile;

     inputFile = fopen ("inputFile.txt","r");
     fscanf(inputFile, "%d", &n);


     short **a_b = (short**)malloc(n*sizeof(short*));//Coefficients a[i,j] followed by
                                                                          b[i]
     for(i=0; i<n; i++)
     {
        a_b[i] = (short*)malloc((n+1)*sizeof(short));
        for(j=0; j<n; j++)
        {
               fscanf(inputFile, "%d", &a_b[i][j]);//a
        }
        fscanf(inputFile, "%d", &a_b[i][n]);//b
     }
     fclose(inputFile);


     Xi_old = (float*)malloc(n*sizeof(float));
     Xi_new = (float*)malloc(n*sizeof(float));
     for(j=0; j<n; j++)
     {
        Xi_old[j] = 0;//Initialization
     }

     //Computations
     do
     {
        convergence_flag = 0;
        for(i=0; i<n; i++)
        {
               float sum = 0;
               for(j=0; j<n; j++)
               {
                      sum += a_b[i][j]*Xi_old[j];
               }
               sum -= a_b[i][i]*Xi_old[i];

               Xi_new[i] = (1.0/a_b[i][i]) * (a_b[i][n]/* b[i] */-sum);

               if(fabs(Xi_old[i]-Xi_new[i]) <= epsilon)
                      convergence_flag++;
        }

        for(i=0; i<n; i++)
        {
              Xi_old[i] = Xi_new[i];
        }
     }while(convergence_flag<n);



     printf("\n");
     for (i = 1; i <n+1; i++)
     {
        printf(" X%d = %f\t", i, Xi_new[i-1]);
     }
     printf("\n\n");

     free(a_b);
     free(Xi_old);
     free(Xi_new);
}
```

Parallel Program

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

#define epsilon 0.0001

main(int argc, char* argv[])
{
   int n, my_rank, p;
   float *Xi_old,*Xi_new;//Xi in each iteration
   short *a_b;
   int tag;
   short convergence_flag;
   MPI_Comm slave_comm;
   MPI_Group world_group, slave_group;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);


   MPI_Comm_group(MPI_COMM_WORLD, &world_group);
   int master_ranks[] = {0};
   MPI_Group_excl(world_group, 1, master_ranks, &slave_group);
   MPI_Comm_create(MPI_COMM_WORLD, slave_group, &slave_comm);

   //Checking whether there are correct number of processes needed.
   FILE * inputFile = fopen ("inputFile.txt","r");
   fscanf(inputFile, "%d", &n);
   if((n+1)!=p)
   {
        printf("There must be %d processes (= problemDimention + 1). Terminating
                           process %d!\n",n+1,my_rank);
        fclose(inputFile);
        MPI_Finalize();
        exit(0);
   }
```

```
        fclose(inputFile);


        if(my_rank == 0)
        {
             int i,j;
             FILE *inputFile;

             inputFile = fopen ("inputFile.txt","r");
             fscanf(inputFile, "%d", &n);


             short **a_b_complete = (short**)malloc(n*sizeof(short*));//Coefficients a[i,j]
                                                                     //followed by b[i]
             for(i=0; i<n; i++)
             {
                 a_b_complete[i] = (short*)malloc((n+1)*sizeof(short));
                 for(j=0; j<n; j++)
                 {
                     fscanf(inputFile, "%d", &a_b_complete[i][j]);//a
                 }
                 fscanf(inputFile, "%d", &a_b_complete[i][n]);//b
             }
             fclose(inputFile);

             //Distributing coefficients
             for(i=1; i<p; i++)
             {
                 tag = 1;
                 MPI_Send(&n, 1, MPI_INT, i, tag, MPI_COMM_WORLD);

                 tag = 2;
                 MPI_Send(&a_b_complete[i-1][0], n+1, MPI_SHORT, i, tag, MPI_COMM_WORLD);
             }

             //Collecting final Xi for all i
             float *Xi;
             Xi = (float*)malloc(n*sizeof(float));
             tag = 4;
             for (i = 1; i <p; i++)
             {
                 MPI_Recv(&Xi[i-1], 1, MPI_FLOAT, i, tag, MPI_COMM_WORLD, &status);
             }printf("\n");
             for (i = 1; i <p; i++)
             {
                 printf(" X%d = %f\t", i, Xi[i-1]);
             }
             printf("\n\n");



             for(i=0; i<n; i++)
             {
                 free(a_b_complete[i]);
             }
             free(a_b_complete);
             free(Xi);
        }

        else
        {
             int i;

             tag = 1;
             MPI_Recv(&n, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
             a_b = (short*)malloc((n+1)*sizeof(short));
        tag = 2;

        MPI_Recv(a_b, n+1, MPI_SHORT, 0, tag, MPI_COMM_WORLD, &status);

        Xi_old = (float*)malloc(n*sizeof(float));
        Xi_new = (float*)malloc(n*sizeof(float));
        for(i=0; i<n; i++)
        {
            Xi_old[i] = 0;//Initialization
        }

        //Computations
        i = my_rank-1;
        do
        {
            float sum = 0;
            int j;
            for(j=0; j<n; j++)
            {
                sum += a_b[j]*Xi_old[j];
            }
            sum -= a_b[i]*Xi_old[i];

            Xi_new[i] = (1.0/a_b[i]) * (a_b[n]/* b */-sum);

            if(fabs(Xi_old[i]-Xi_new[i]) <= epsilon)
                convergence_flag = 1;
            else
                convergence_flag = 0;

            Xi_old[i] = Xi_new[i];


            MPI_Barrier(slave_comm);


            MPI_Allreduce (&convergence_flag, &convergence_flag, 1, MPI_SHORT, MPI_MIN,
               slave_comm);//Convergence check

            tag = 3;
            for(j=1; j<p; j++)
            {
                if(j != my_rank)
                {
                  MPI_Send(&Xi_old[i], 1, MPI_FLOAT, j, tag, MPI_COMM_WORLD);
                  MPI_Recv(&Xi_old[j-1], 1, MPI_FLOAT, j, tag, MPI_COMM_WORLD,&status);
                }
            }
        }while(convergence_flag==0);
        tag = 4;
        MPI_Send(&Xi_new[i], 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);

        free(a_b);
        free(Xi_old);
        free(Xi_new);
    }

      MPI_Finalize();
}
```