A parallel clustering algorithm with MPI Jacobi Iteration Method

Submitted By
Shuvradeb Saha - BSSE815
Suravi Akhter - BSSE827

Submitted To
Dr. B M Mainul Hossain
Associate Professor
Institute of Information Technology
University of Dhaka

Date: 30/04/18

Contents

1 Introduction **3**

2 Related Work **3**
2.1 Jacobi Iteration Method 3
2.2 The concept of MPI 4
2.3 MPI Functions 5
2.4 Message Passing Process of MPI 5

3 Methodology **5**
3.1 Jacobi Iteration Method 5

4 Experiment **6**
4.1 Experimental Environment 6
4.2 Data Sets 6
4.3 Experimental Results and Analysis 6

5 Discussion **7**
5.1 Performance Defects 7

6 Conclusions **8**

Acknowledgement **8**

1 Introduction

In this project we have implement Jacobi Iteration Method. Our project target is to minimize the execution time, overhead communication cost and message passing by implementing a parallel code for Jacobi method. Our goal is to develop this iteration method which minimizes communication and to measure the impact of communication on the performance of this method. There exists a lot of research regarding methods for solving systems of linear equations of the form Ax=b. Many algorithms from different math and computer science researchers are currently in place. Basically they are separated into two groups: direct and iterative methods. From the non practical application of Gaussian elimination as a direct method for solving systems Ax=b, iterative methods were widely studied from the scientists. There are lot of advantages and disadvantages between Gaussian elimination as an example of direct method and an iterative one like Jacobian [1]. The general framework of an iterative process is as simple as this: first, an initial assumption-solution is generated intuitively for the vector-solution . Then using this assumption the algorithm provides us with a possible solution x(0). Now the role of the solution x(1) becomes the input for next possible solution. This process goes repeatedly, providing an array of vector solution x(0), x(1), x(2) x(*), until we get into a satisfactory solution. As there is same computation many times we divide the work among some computers and make the computation faster than single computer.

2 Related Work

Related work including Jacobi Iteration Method and MPI are involved in this section. More precisely, an outline on Jacobi Iteration Method is summarized in Section 2.1, the concept of MPI is given in Section 2.2, brief introduction of MPI functions is given in Section 2.3, and the message passing process of MPI is described in Section 2.4.

2.1 Jacobi Iteration Method

When solving the systems of linear equations of the form:

$$\begin{cases} a_{11}x_1 + \cdots a_{1n}x_n = b_1 \\ \vdots \qquad\qquad \ddots \qquad\quad \vdots \\ a_{n1}x_1 + \cdots a_{nn}x_n = b_n \end{cases}$$

or Ax=b.....(*)

There exist two kinds of methods [2]: Direct methods, through which we obtain the solution of the system (*) after a finite and known number of steps and the error of these kind of methods is 0, therefore are also called exact methods. Iterative methods, which solving the system (*) produce an array of approximate values (vector-solution), which converges in some defined circumstances to the exact solution of the system. The Jacobi method is one of the oldest iterative methods which are engaged in solving the system (*). In order to describe the method procedure firstly we set the system in more proper form like the following:

$$\begin{cases} x_1 = -\dfrac{\sum_{j=2}^{n} a_{1j}x_j}{a_{11}} + \dfrac{b_1}{a_{11}} \\ \vdots \qquad\qquad \ddots \qquad\quad \vdots \\ x_i = -\dfrac{\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j}{a_{ii}} + \dfrac{b_i}{a_{ii}} \\ \vdots \qquad\qquad \ddots \qquad\quad \vdots \\ x_n = -\dfrac{\sum_{\substack{j=1 \\ j\neq i}}^{n-1} a_{nj}x_j}{a_{nn}} + \dfrac{b_n}{a_{nn}} \end{cases}$$

Or

$$x^{(k+1)} = Tx^{(k)} + c \qquad (1)$$

Hence, the formula in terms of its elements would look like the following:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}\, x_j^{(k)}\right), \quad i = 1, 2, \ldots, n. \qquad (**)$$

Formula (**) meanwhile represents the Jacobian iterations. This method assumes we have all the input values of x in the previous iteration (k). But, usually there are not given all the x values. What we can do is to make an initial prepossession for x and to generate another group of solutions for x from the equation (**), which in fact will represent the input values for the next iteration (k+1). After we have found the group of x values for the previous iteration we continue generating new

groups again and again until we arrive at an acceptable solution. These iterations produce an array of approximate values for the real solution of the system (*). With the word acceptable solution we will intend that group of solutions, respectively to the approximations, of x with the required accuracy.
If the vector-solution values are getting closer to the expected solution with the growth of iterations, then it is said that it converges to the solution of the system (*). In the majority cases the approximation array results with a good estimation for the values of x i.e. converges. Note that this algorithm is nonfunctional for all matrices. One of the wide exceptions is any matrix with any 0 in diagonal. We stress out the following requirements of the Jacobi method, in order for it to be functional:

1. All the members of the main diagonal of the matrix A must be nonzero (aij != 0 and i=1, 2...n).
2. This method requires a duplicate storage for the vector-solution x. This, because none of the members (elements) of x can be overwritten while all the x-elements are calculated for that iteration i.e. it is needed an array of current solution to manipulate with and another array, we will note it with x old, holding values of the previous iteration.
3. Components (elements) of the new iteration vector solution are not dependent of each other, hence can be computed at the same time. This identifies the high potential of the parallelization of Jacobi method.
4. The solution provided by the Jacobi method not always converges. It is ensured only within some specified circumstances, which will be discussed in the next subsection.

2.2 The concept of MPI

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.
MPI (Message Passing Interface) is a specification for the developers and users of message passing libraries. By itself, it is NOT a library but rather the specification of what such a library should be. MPI primarily addresses the message-passing parallel programming model where data is moved from the address space of one process to that of another process through cooperative operations on each process. [2]

Reasons for Using MPI:

1. Standardization - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

2. Portability - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

3. Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

4. Functionality - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

5. Availability - A variety of implementations are available, both vendor and public domain.

The standards of MPI are as follows [3]:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Bindings for FORTRAN 77 and C
- Environmental management and inquiry
- Profiling interface

2.3 MPI Functions

Although MPI is a complex and multifaceted system, we can solve a wide range of problems using just six of its functions! We introduce MPI by describing these six functions, which initiate and terminate a computation, identify processes, and send and receive messages [4]:

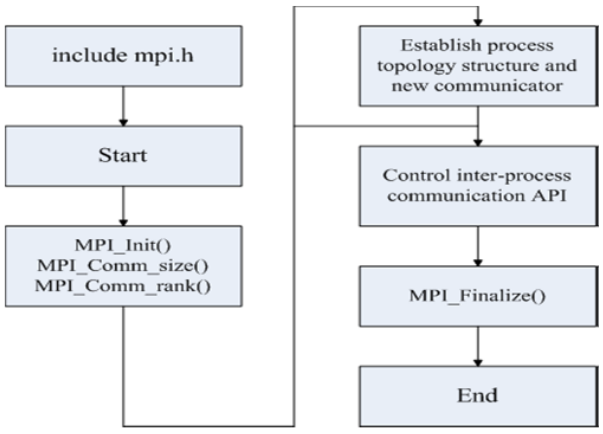
1. `MPI_INIT` : Initiate an MPI computation.
2. `MPI_FINALIZE` : Terminate a computation.
3. `MPI_COMM_SIZE` : Determine number of processes.
4. `MPI_COMM_RANK` : Determine my process identifier.
5. `MPI_SEND` : Send a message.
6. `MPI_RECV` : Receive a message.

Description of each function is given below-

- **MPI_INIT(int *argc, char ***argv)**: Initiate a computation. argc, argv are required only in the C language binding, where they are the main program arguments.
- **MPI_FINALIZE()**: Shut down a computation.
- **MPI_COMM_SIZE(comm, size)**:
  IN comm communicator.
  OUT size process id in the group of comm (integer).
- **MPI_SEND(buf, count, datatype, dest, tag, comm)**:
  IN buf address of send buffer (choice).
  IN count number of elements to send (integer >= 0).
  IN datatype data type of send buffer elements (handle).
  IN dest process id of destination process (integer).
  IN tag message tag (integer).
  IN comm communicator.
- **MPI_RECV(buf, count, datatype, source, tag, comm, status)**:
  Receive a message
  OUT buf address a receive buffer (choice)
  IN count size of receive buffer elements (integer>=0).
  IN datatyp e data type of receive buffer elements (handle).
  IN source process id of source process, or MPI_ANY_SOURCE (integer).
  IN tag message tag or MPI_ANY_TAG (integer).
  IN comm communicator.
  OUT status status object (status).

2.4 Message Passing Process of MPI

MPI is a mainly established to divide work among some computers and minimize the calculation or computing time of data operation. If we divide the tasks among some computers then one computer need to communicate with other computer to give data and receive data. For this reason MPI is based on the message passing, whose function is to exchange information, coordinate and control the implementation steps with the definition of program grammar and semantics in the core library by sending messages between the concurrent execution parts.
All MPI program include mpi.h header, by including this header in a program we can use all library function which we need to write a MPI code. We initialize all process by MPI_Init() function and after finishing tasks we terminate all process by MPI_Finalize() function.
The parallel program design flow of message passing process is shown in the following figure:[5]



3 Methodology

In this section we propose an algorithm based on MPI a parallel implementation of Jacobi iteration method.

3.1 Jacobi Iteration Method

We implement Jacobi iteration method in MPI clustering by partitioning data among processes. The main idea of the project is to solve linear equation in message passing interface environment. All initialization is implemented by the `MPI_Init` function, which is the first call of MPI program. It is the first executable statement of the MPI program. To start the MPI environment, it means the beginning of the parallel codes. The `MPI_Finalize` function is the last call of MPI program, and it ends the running of MPI program. It is the last executable statement of MPI program, otherwise, results of the procedure is unpredictable. `MPI_Finalize` symbolizes the end of the

parallel codes.
In first step we read matrix A and B from file and evenly divide input data (matrix A and B) among processes. In our algorithm parallelism is implemented by the data parallelism.
Algorithm for serial implementation of Jacobi Iteration Method is given below-

Algorithm 1 Serial Algorithm of Jacobi Iteration Method

```
1: Input Matrix A(n*n) and Matrix B(n*1)
2: while convergence not reached do
3:   for i=1 step until n do
4:     for j=1 step until n do
5:       if i ≠ j then
           end if
6:     end for
7:   end for
8: end while
```

Algorithm for parallel implementation of Jacobi Iteration Method is given below-

Algorithm 2 Parallel Algorithm of Jacobi Iteration Method

```
1: Input Matrix A(n*n) and Matrix B(n*1)
2: Read N objects from the file
3: Partition N data objects evenly among all processes,
   and assume that each process has N' data objects
4: Broadcast data from all processes when rank is 0
5: Calculate x^0,x^1...x^* for all processes
6: Gather result from all processes
7: MPI_Finalize // finish the procedure
```

Following is the code of divide same row amount of data among all processes-
Following is the code segment where we implement parallel code for Jacobi Iteration method -

4 Experiment

4.1 Experimental Environment

The hardware platform which we used run our MPI code has 4GB RAM, Linux operating system, quad-core processor, Hard Disk 1TB etc. In experimental platform we use gcc compiler to compile our code. In command line for compiling program we write this "mpicc -o jacobi jacobi.c"command in terminal. mpicc and mpiCC (mpic++ is a synonym for mpiCC provided for file names that do not support case-sensitive file names) are convenience wrappers for the local native C and C++ compilers. For executing our MPI code we write this "mpirun -np 4 ./jacobi" where 4 stands for number of process and jacobi is for the object file name. In cluster, for excuting our MPI code we write this "mpirun -np 12 -hosts bsse827, bsse815, bsse841, bsse810" where bsse827 is master PC and others are clients.[6]

4.2 Data Sets

As we did not find huge amount of data sets for our project for this reason we make our own data set which is generated by a code. After executing the code we find data set contains matrix A(n*n) and matrix B(n*1). In the data generation code we generate random number in a specific range.

4.3 Experimental Results and Analysis

In our experiments, the time cost is the key performance. The I/O time and clustering time is calculated in Jacobi Iteration Method.

Speed Up for Matrix(500*500)			
Process	Single	2 PC	4 PC
4	2.8497	0.8081	0.6909
10	0.076	0.1279	0.3350
25	0.034	0.04601	0.0823

Table 1: Table of speed up for data size 250000

From the above table we found that speed up is highest in single computer for 4 process. Speed up is high in 4 computer for 10 process. Again speed up is highest for 25 process is high in 2 computer. This occurred because of overhead communication and weak internet connection.

Speed Up for Matrix(1000*1000)			
Process	Single	2 PC	4 PC
4	3.2652	1.5227	0.7868
8	0.3902	1.3249	1.1748
20	0.1526	0.1948	0.3275

Table 2: Table of speed up for data size 1000000

From the above table we found that speed up is highest in single computer for 4 process. Speed up is high in 4 computer for 8 and 20 process. This occurred because of overhead communication and weak internet connection.

Speed Up for Matrix(5000*5000)			
Process	Single	2 PC	4 PC
4	3.4316	1.1334	1.1974
8	1.4608	1.8625	1.40216
20	0.1526	1.0134	1.04625

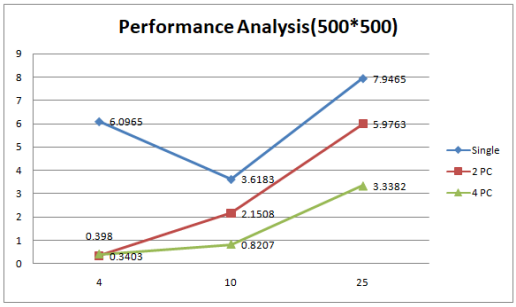
Table 3: Table of speed up for data size 25000000

From the above table we found that speed up is highest in single computer for 4 process. Speed up is high in 2 computer for 8 processes. And for 4 computer speed up is high for 20 processes.

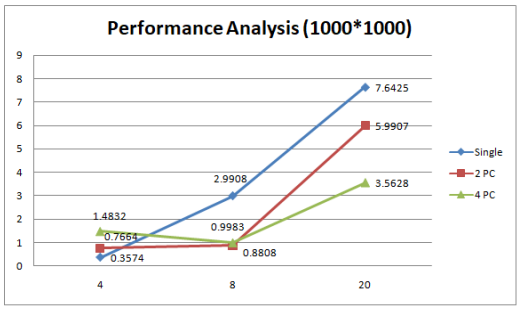
Speed Up for Matrix(10000*10000)			
Process	Single	2 PC	4 PC
4	2.2839	1.0995	1.0279
8	1.0982	1.3	1.0774
16	0.5324	0.6451	1.0768

Table 4: Table of speed up for data size 100000000

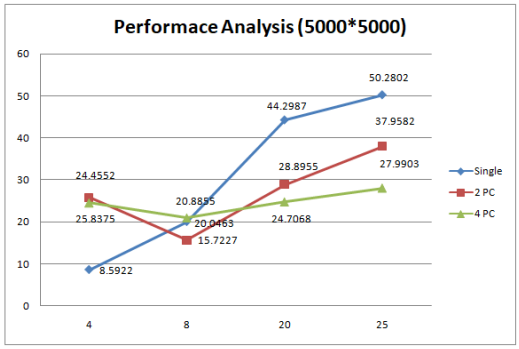
From the above table we found that speed up is highest in single computer for 4 process. Speed up is high in 2 computer for 8 processes. For 16 processes speed up is high in 4 computer.



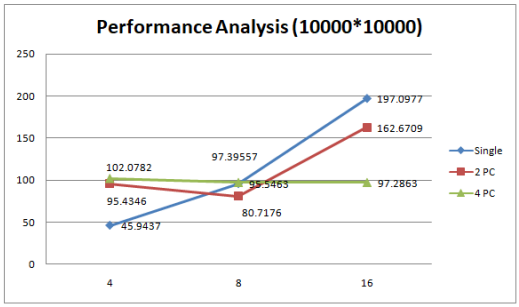
From this figure we find that if we increase number of process then at a critical point clustering in 4 computers and 10 processes perform better. For 4 processes, performance is increased in 2 computer and 4 computer clustering because of overhead communication. In 25 processes, performance is increased for 4 computer clustering because execution time is decreased. In all cases performance is increased for 4 computers.



From this figure we find that clustering in multiple computer performance less than single process. For 8 processes, performance is increased in 4 computer clustering. In 20 processes, performance is increased for 4 computer clustering because execution time is decreased.



From this figure we find that if we increase number of process then at a critical point clustering in 4 computers and 20 processes perform better. But it is not true in every time that if we increase number of process then performance will increase. It is also occurred for overhead communication and weak internet connection. For 4 processes, performance is increased in 4 computer clustering. In 20 and 25 process, performance is increased because execution time is decreased. But for 4 processes performance is increased for single computer for the reason of minimum execution time.



From this figure we find that if we increase number of process then at a critical point clustering in 4 computers and 16 processes perform better. For 4 processes, performance is increased in single computer clustering because of overhead communication. In 16 processes, performance is increased for 4 computer clustering because execution time is decreased. But for 4 processes performance is increased for single computer for the reason of minimum execution time.

5 Discussion

5.1 Performance Defects

From the performance analysis figure which is given in experiment part of the report, we can see that for 500*500 matrix size it took more time than serial execution. Serial execution took 0.275s where parallel execution for 4 process took 0.3403s for 4 computer clustering execution, 0.398s for 2 computer clustering execution and 6.0965s for single computer parallel execution. From this result we find out that parallel execution is not always better than serial execution if data size is not very huge.

For matrix size 1000*1000, we can see that serial execution took 1.167s and parallel execution took less time than serial execution. When we execute the code for parallel processing in 4 processor it took 0.3574s in 4 computer clustering execution, 0.7664s in 2 computer clustering execution and 1.4832s for single computer processing. Again execution time is increase when we execute the program in 8 process because of overhead communication. So we can conclude that performance is not always improve if we increase number of processor. After a critical it will not improve.
For matrix size 5000*5000, parallel execution performs better than serial execution. Serial execution took 29.485s and parallel execution for 4 process took 8.5922s for single computer parallel execution, 24.4552s for 2 computer clustering execution and 25.8375s for 4 computer clustering execution.
And same about for matrix size 10000*10000. After a certain number of processor if we increase the number of processor then performance will not always improve. That because of overhead communication and also for slow internet connection.

6 Conclusions

We have proposed Jacobi iteration method implementation using Message Passing Interface(MPI) and also implement a serial version of Jacobi iteration method code to check out the improvement and performance. The performance of the algorithm depended on two key factors - the efficiency and scalability of the implementation, and the effect of asynchronous on the number of iterations taken to converge both of which vary with the number of cores used. In future we want to improve our skills in MPI and implement a better solution to solve the linear equation in less time than serial execution.

Acknowledgement

At first, we would like to thank almighty for helping us to complete the report of the project which is on A parallel clustering algorithm using MPI - Jacobi Iteration Method. We would like to express our deepest gratitude to all those who provided us the support and encouragement to complete this project. We are grateful to our course teacher, Dr. B M Mainul Hossain, Associate Professor, Institute of Information Technology, University Of Dhaka, for giving us guidelines about how we can prepare this report. We are grateful to the Institute of Information Technology for giving us the opportunity to do such a project.

References

[1] W. R. Fraser, Gaussian Elimination vs Jacobi Iteration, Project Report, 21 November 2008.

[2] https://computing.llnl.gov/tutorials/mpi

[3] Y. Aoyama, J. Nakano, RS/6000 SP: Practical MPI Programming, International Technical Support Organization, August 1999.

[4] http://www.mcs.anl.gov/itf/dbpp/text/node96.htm

[5] JOURNAL OF COMPUTERS, VOL. 8, NO. 1, JANUARY 2013

[6] MPI tutorial: Run MPI In Cluster http://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/