RYERSON UNIVERSITY

Faculty of Electrical and Computer Engineering
Department of Electrical and Computer Engineering
Program: Computer Engineering

Course Number	**EE 8218 – 011**
Section Number	**01**
Course Title	**Parallel Computing**
Semester/Year	Fall 2015

Instructor	**Nagi Mekhiel**

ASSIGNMENT No.	**02**

Assignment Title	**Introduction to MPI**

Submission Date	**November 03,2015**
Due Date	**November 03,2015**

Student Name	**Ismail Sheikh**
Student ID	Xxxx89867
Signature*	**M.Ismail**

(Note: Remove the first 4 digits from your student ID)

Objective:

This lab is about installing and getting familiarize with the MPI software. Similarly, visualize the performance improvement through parallel computing a program in a network of computers

Introduction:

Message Passing Interface (MPI) is a portable message-passing system used in many computer languages such as FORTRAN, C, C++ and Java. Further, MPI software, allows to run a computer program parallel in the network of computers to increase the computational power of a system. Parallel computing is really important for handling big data since the sequential algorithm (single processor) has many limitation when it comes to big data. Depending on data, they might take years to solve a problem. However, by diving the problem and solving it parallel in the network of computers helps solve the problem much faster.

This being said, there are certain thing one has to keep in mind for parallel computing. Such as for parallel computing the data has to be big and independent since the communication time is really high for the small or dependent data resulting in poor performance in the network of computer. Similarly, the data has to be divided in the network equally. If the data is not distributed equally, the process with minimum task will finish it quickly and stay idle until the rest of the processors finish their task which results in long overhead. In this report, these problem will be explicitly discuss and proven with the results.

Experiment:

In order to visualize and compare the performance of a parallel computing, two square matrices with the size of 1000, 3000, and 5000, 6000 respectively initialized. Further, both square matrices where multiplied together using sequential algorithm as well as parallel computing algorithm in the network of 4 and 6 computers.

The result of sequential algorithm is as follow:

Size of Matrices	**Computational Time (seconds) for sequential algorithm**
1000 x 1000	10.36 Seconds
3000 x 3000	339.92 Seconds (~5 Minutes)
5000 x 5000	1577.78 Seconds (~26.5 Minutes)
6000 x 6000	2523.55 Seconds (~42 Minutes)

Table 1 above represents the size of two square matrices used for the application and the time the system took to multiply those two matrices using sequential algorithm

Size of Matrices	Computational Time (seconds) in a network of 4 computers
1000 x 1000	3.44 Seconds
3000 x 3000	147.04 Seconds (~2.45 Minutes)
5000 x 5000	678.63 Seconds (~11.5 Minutes)
6000 x 6000	979.52 Seconds (~16.5 Minutes)

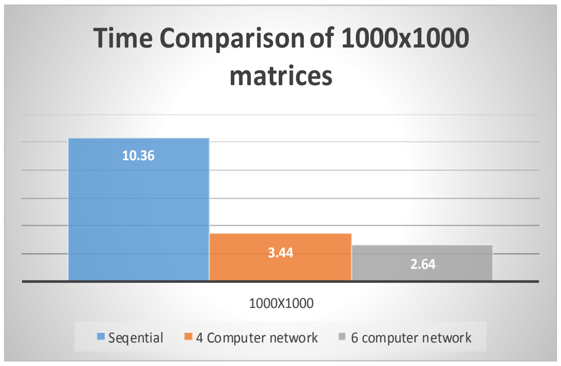
Table 2 above represents the size of two square matrices used for the application and the time the system took to multiply those two matrices in a network of 4 computers.

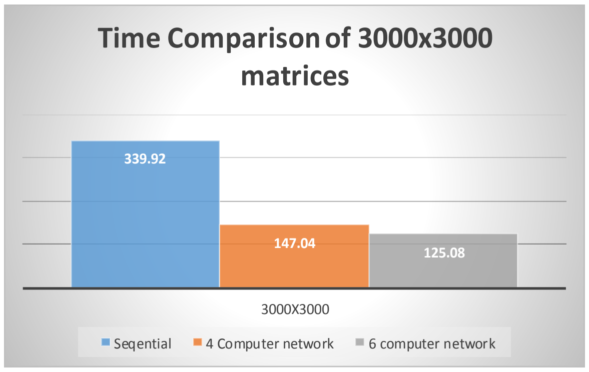
Size of Matrices	Computational Time (seconds) in a network of 6 computers
1000 x 1000	2.64 Seconds
3000 x 3000	125.08 Seconds (~2.1 Minutes)
5000 x 5000	525.63 Seconds (~9Minutes)
6000 x 6000	625.6 Seconds (~10.5 Minutes)

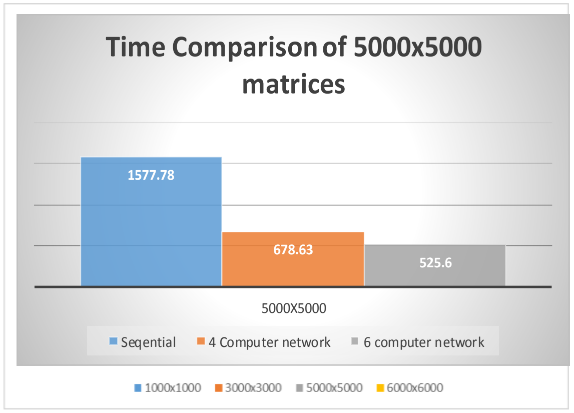
Table 3 above represents the size of two square matrices used for the application and the time the system took to multiply those two matrices in a network of 6 computers.

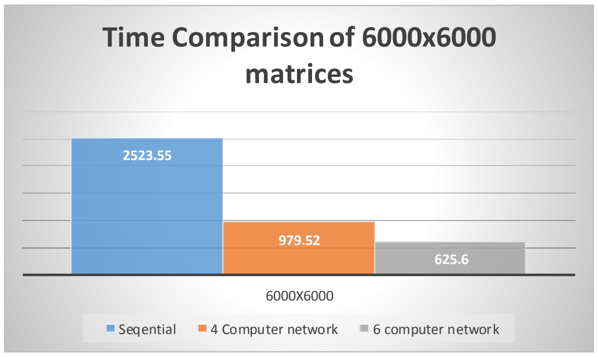
Result Comparison:

Further, as the size of the matrices becomes bigger and bigger there are more values to execute and sequential algorithm waits for the first commands to complete before executing the next command which results in a really slow response. Similarly, the parallel algorithm divide the workload into network of computers and execute at the same time. After completing the process, they send their result to main host computer where the main computer combines the results in much faster way.

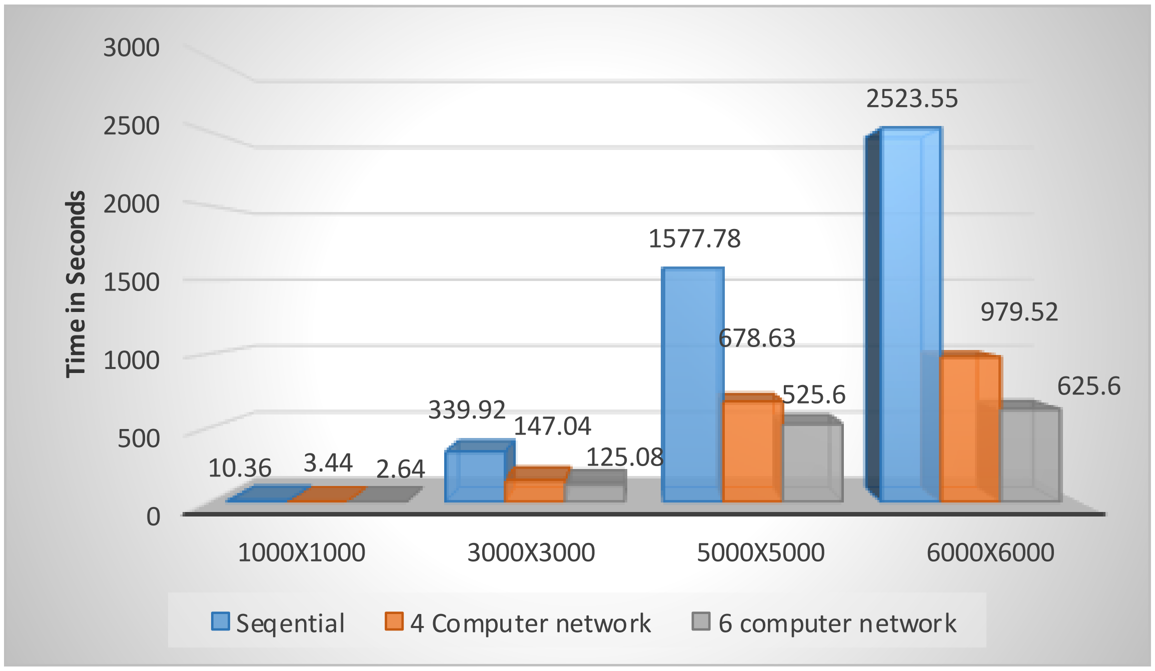








Hence, the graph below represents the computation time for all matrices, 1000x1000, 3000x3000, 5000x5000 and 6000x6000.

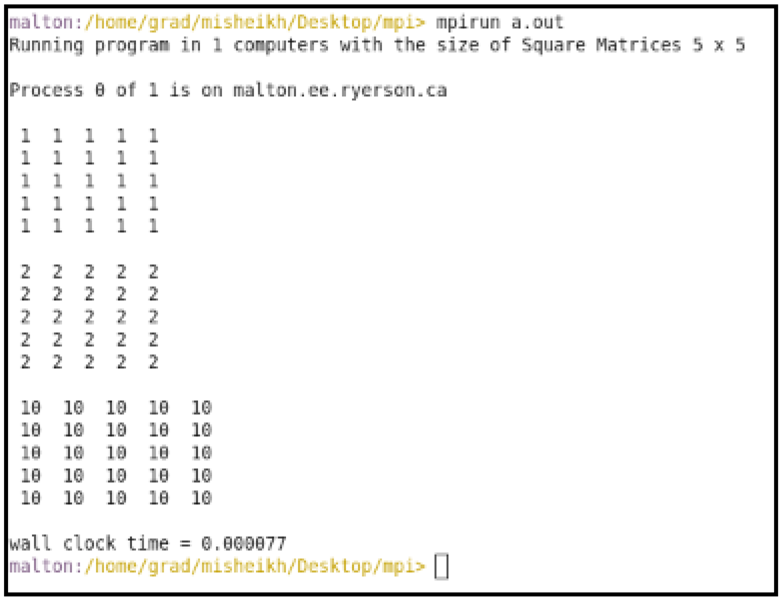


Conclusion:

Based on the above results we see that in parallel computing, even by using 4 computers network or 6 computers network, we still don't receive 4 times or 6 times faster response. That could be the reason for many different options. First, in this lab, we only used maximum size of matrix to be 6000x6000 which is still not big enough. Also, it could also be the result of network speed since we are communicating with a network of computers. In order to achieve a better result, we have to use faster network speed with Fiber optics.

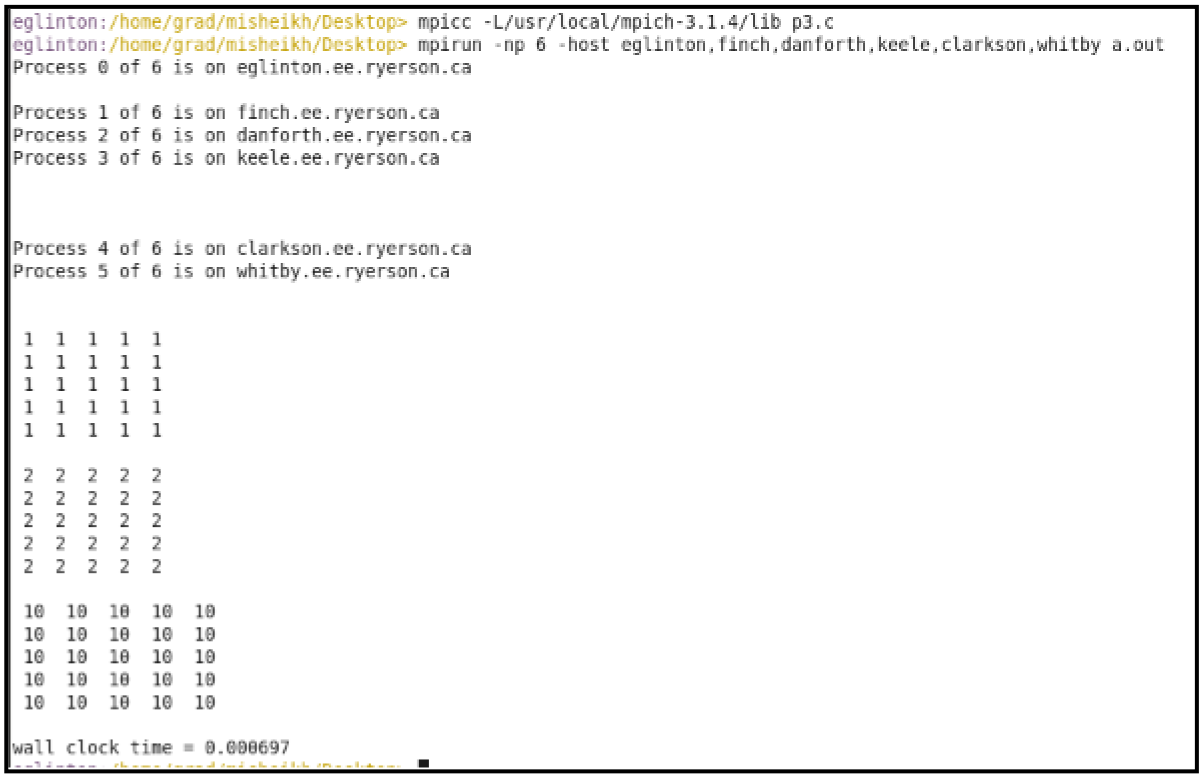
However, we do see from the above graph that as the matrix size is increases, the time difference in sequential, 4 computer network and 6 computer network increases.

Appendix A:
Program output used to verify that the program works



```
malton:/home/grad/misheikh/Desktop/mpi> mpirun a.out
Running program in 1 computers with the size of Square Matrices 5 x 5

Process 0 of 1 is on malton.ee.ryerson.ca

 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1

 2  2  2  2  2
 2  2  2  2  2
 2  2  2  2  2
 2  2  2  2  2
 2  2  2  2  2

 10  10  10  10  10
 10  10  10  10  10
 10  10  10  10  10
 10  10  10  10  10
 10  10  10  10  10

wall clock time = 0.000077
malton:/home/grad/misheikh/Desktop/mpi> 
```

Program Output for 1 computer



```
eglinton:/home/grad/misheikh/Desktop> mpicc -L/usr/local/mpich-3.1.4/lib p3.c
eglinton:/home/grad/misheikh/Desktop> mpirun -np 6 -host eglinton,finch,danforth,keele,clarkson,whitby a.out
Process 0 of 6 is on eglinton.ee.ryerson.ca

Process 1 of 6 is on finch.ee.ryerson.ca
Process 2 of 6 is on danforth.ee.ryerson.ca
Process 3 of 6 is on keele.ee.ryerson.ca



Process 4 of 6 is on clarkson.ee.ryerson.ca
Process 5 of 6 is on whitby.ee.ryerson.ca


 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1

 2  2  2  2  2
 2  2  2  2  2
 2  2  2  2  2
 2  2  2  2  2
 2  2  2  2  2

 10  10  10  10  10
 10  10  10  10  10
 10  10  10  10  10
 10  10  10  10  10
 10  10  10  10  10

wall clock time = 0.000697
```

Program Output for network of 6 computers

Appendix 2:
Program code:

```
#include "/usr/local/mpich-3.1.4/include/mpi.h"
#include <stdio.h>
#include <math.h>
#define sizeOfMatrix 1000
int matrix1[sizeOfMatrix][sizeOfMatrix];
int matrix2[sizeOfMatrix][sizeOfMatrix];
int result[sizeOfMatrix][sizeOfMatrix];
int row, colum;
int    n, myid, numprocs;
int tempArray[sizeOfMatrix * sizeOfMatrix];
int myid, numprocs, temp;
double startwtime = 0.0, endwtime;
int    namelen, ierr, icount;
int i, j, k;
int columNumber = 0;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Status status;

void initialize();
void display();

int main(int argc, char *argv[]) {
       MPI_Init(&argc, &argv);
       MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
       MPI_Comm_rank(MPI_COMM_WORLD, &myid);
       MPI_Get_processor_name(processor_name, &namelen);

       for (i = 0; i < numprocs; i++);
       printf("Running program in %d computers with the size of Square Matrices %d x %d \n\n",
i, sizeOfMatrix, sizeOfMatrix);

       printf("Process %d of %d is on %s\n", myid, numprocs, processor_name);
       printf("\n");
       initialize();
       if (myid == 0)
              startwtime = MPI_Wtime();

       for (row = 0; row < sizeOfMatrix; row++) {
              for (colum = 0; colum < sizeOfMatrix; colum += numprocs) {
                     for (k = 0; k < sizeOfMatrix; k++) {
                            if (myid == 0)
                                   result[row][colum] += matrix1[row][k] * matrix2[k][colum];
                            else
                                   tempArray[columNumber] += matrix1[row][k] * Matrix2[k][colum];
                     }
                     columNumber++;
              }
       }
       if (myid == 0) {
              int i, columns, j, counter;
              for (i = 1; i < numprocs; i++)
              {
                     counter = 0;
                     MPI_Recv(&tempArray, columNumber, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                     for (row = 0; row < sizeOfMatrix; row++) {
                            for (j = i; j < sizeOfMatrix; j += numprocs) {
```

```
                                        result[row][j] = tempArray[counter];
                                        counter++;
                                }
                        }
                }
        }
        else
                MPI_Send(&tempArray, columNumber, MPI_INT, 0, 0, MPI_COMM_WORLD);


        if (myid == 0) {
                // display();
                endwtime = MPI_Wtime();
                printf("wall clock time = %f\n", endwtime - startwtime);
        }

        MPI_Finalize();
        return 0;
}
void initialize() {
        for (row = 0; row < sizeOfMatrix; row++) {
                for (colum = 0; colum < sizeOfMatrix; colum++) {
                        // matrix1[row][colum] = 1;
                        //matrix2[row][colum] = 2;
                        matrix1[row][colum] = rand() % 100;
                        matrix2[row][colum] = rand() % 100;
                }
        }
}

void display() {
        int counter = 0;
        for (counter = 0; counter < 3; counter++) {
                for (row = 0; row < sizeOfMatrix; row++) {
                        for (colum = 0; colum < sizeOfMatrix; colum++) {
                                if (counter == 0)
                                        printf(" %d ", matrix1[row][colum]);
                                else if (counter == 1)
                                        printf(" %d ", matrix2[row][colum]);
                                else if (counter == 2)
                                        printf(" %d ", result[row][colum]);
                        }
                        printf("\n");
                }
                printf("\n");
        }

}
```