EEE4120 Practical 4

1st Thomas Greenwood
GRNTHO006
UCT
Cape Town, South Africa
GRNTHO006@myuct.ac.za

2nd Charles Portman
PRTCHA018
UCT
Cape Town, South Africa
PRTCHA018@myuct.ac.za

***Abstract*—Message Passing Interface (MPI) is a common tool used in C programming to facilitate the communication between parallel processing clusters. When parallel processing takes place each process runs on a separate memory space, meaning the way in which the data is distributed will greatly affect the efficiency of processing and the program.**

I. Introduction

To investigate the way in which the data is distributed and how this effects processing, this a program using C++ and MPI will be compared to a golden measure of Matlab. The comparison will be between speed of bubble sort algorithm, sorting various size matrices

A. Distribution of work

To maximise the speed up available for parallelism, work should be evenly distributed between the parallelised sections. In this instance, this distribution can be done by finding the size of an array and the number of workers, splitting the array into equal chunks and distributing each chunk to a parallel worker. However, in cases where the array cannot be perfectly divided between the workers. If there is an array that cannot be equally distributed, will result in leftover the work being done by the master worker. This results in a slightly uneven distribution of work between the processors. This is unavoidable without limiting the number of parallel processors or size of array used.

B. Processing data

Data can then be processed independently by each slave and the master, each running the bubble sort algorithm. After which it is reconstructed by the master.

II. Methods

Programs written in C++ and Matlab where compared. Both programs used a bubble sort to sort matrices columns of various sizes. To ensure a fair test both items had to sort the same matrix from a .csv file. The time taken for both programs to run was recorded and used to calculate speed up with:

$$SpeedUp = \frac{TimeTakeninMatlab}{TimeTakeninC++}.$$

```
for(int j = 0; j < chunksize+leftover; j++)
{
```

Fig. 1. Master Iteration

```
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

Fig. 2. NumTasks Message

A. Problem Partitioning

To divide the data into subsections for each worker to operate on, the array is divided equally between the workers. In the case where the size of the array is not divisible by the number of workers, any unequal chunk left is processed by the master worker. This is shown in figure 1 and in figure 10, where the unequal distribution to the master can be seen.

Additionally, the approach of separating any remaining unequal chunk of data to the master worker ensures that processing is as efficiently as possible done among available slaves. While this may introduce a slight imbalance in workload distribution, it minimizes time to partition and reconstruct the data. To indicate to the slave the size of data it is operating on we use a single message sent first in a fixed format. Relevant code snippet is shown in figure 2, figure 3 and figure 4.

Once the slaves have finished processing on the data, it needs to be sent to the master for reconstruction.This can be done by iterating over the slaves. This is shown in code snippet figure 5.

Complete code snippets are shown in figure 7, figure 6, figure 8 and figure 9.

```
chunksize = (arraySize / numtasks);
```

Fig. 3. Chunk Size Partitioning

```
for (int i = 0; i < chunksize; i++)
{
```

Fig. 4. Chunk Size Iteration

```
for (int i = 0; i < chunksize; i++)
{
```

Fig. 5. Master Reconstruction

```
if (taskid == MASTER)
{
    double ti = MPI_Wtime();
    std::vector<double> allTimes(arraySize);

    //sending the data to be sorted
    offset = 0;
    for (dest = 1; dest < numtasks; dest++) {
        //sending the offset
        MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
        /*Havent got these working yet, I think for this to work will need to do a 1xX array - luss
        MPI_Send(&arraySize, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
        MPI_Send(&chunksize, 1, MPI_INT, dest, 3, MPI_COMM_WORLD);
        MPI_Send(&matrix[offset][0], chunksize, MPI_INT, dest, 4, MPI_COMM_WORLD);
        */
        offset+=chunksize;
    }
    //Master doing its section of the work, sorting its chunk and an additional leftover
    std::cout << "Master is busy doing work \n";
```

Fig. 6. Master 1

```
int numtasks, taskid, rc, dest, offset, tag1, tag2, source, chunksize, leftover, arraySize;
double time;


MPI_Init(&argc, &argv);
double startTime = MPI_Wtime();
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Status status;

tag2 = 1;
tag1 = 2;

/*Getting Data from the csvFile */
std::string readFile = argv[1];
std::string writeFile = argv[2];

//trying to just send the data in main instead of this

matrix = csvToVec(filename:readFile);
arraySize = matrix.size();
chunksize = (arraySize / numtasks);
leftover = (arraySize % numtasks);
```

Fig. 7. Initialization

```
//reciving the data
for (int i = 1; i < numtasks; i++) {
    source = i;
    MPI_Recv(&time, 1, MPI_DOUBLE, source, tag2, MPI_COMM_WORLD, &status);
    allTimes[i] = time;
}
allTimes[numtasks + 1] = (MPI_Wtime() - startTime) * 1000;
//writingData to csv
vecToCsv(&:allTimes, & filename:writeFile);
```

Fig. 8. Master 2

```
if (taskid > MASTER) {
    double ti = MPI_Wtime();
    MPI_Recv(&offset, 1, MPI_INT, MASTER, tag1, MPI_COMM_WORLD, &status);
    /*
    MPI_Recv(&arraySize, 1, MPI_INT, MASTER, 2, MPI_COMM_WORLD, &status);

    MPI_Recv(&chunksize, 1, MPI_INT, MASTER, 3, MPI_COMM_WORLD, &status);
    chunksize = (arraySize / numtasks);
    MPI_Recv(&matrix[offset][0], chunksize, MPI_INT, MASTER, 4, MPI_COMM_WORLD, &status);*/


    std::cout << "Worker Number " << taskid << " is trying its best... \n";

    for (int i = 0; i < chunksize; i++)
    {
        matrix[offset + i] = bubbleSortRow(& arr:matrix[offset+i]);
    }
    double time = (MPI_Wtime() - ti)*1000;
    MPI_Send(&time, 1, MPI_DOUBLE, MASTER, tag2, MPI_COMM_WORLD);
```

Fig. 9. Slave

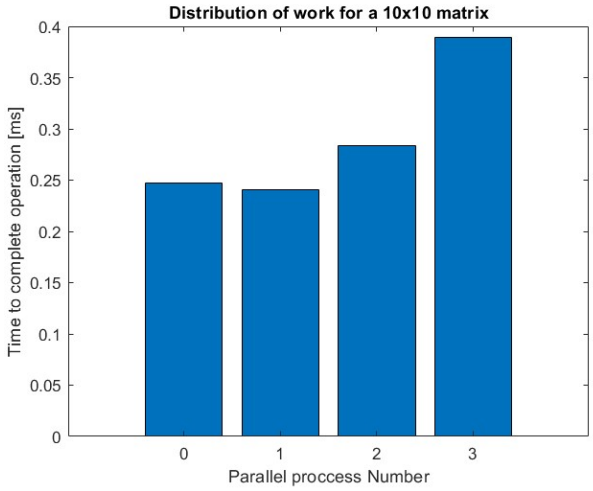

Fig. 10. Distribution 100x100

Full code is available at:
https://github.com/Thom-the-ass/EEE4120F_Pracs/tree/68ddf37a6f2193c2910a7d5eea2f68849c2fb72f/Practical4Part3

B. Experiments

In this section we exploring experimentation and runtime of the parallelisation with a 100x100 array, a 1000x1000 array and a 10 000x10 000 array. We can view how the workload is distributed over the 4 workers for each array size. These results are recorded and plotted in the results section.

We can also compare the speed up vs. Matlab parallelisation as the size of the array to be sorted increases through these values. As the size of the array increases we expect the MPI parallelisation to improve in performance over the Matlab code.

III. Results and Discussion

The runtime of the parallelisation with a 100x100 array, a 1000x1000 array and a 10 000x10 000 array distributed over the 4 workers is shown in figure 11, figure 12 and figure 10.

We can visualise the speed up vs. Matlab as the size of the array to be sorted increases through these values in the graph figure 13.

When this is compared to the speed up we achieved previously with a median filter, comparison shown in figure 14.This clearly shows that the results achieved using explicit parallelisation for a median filter yield a significantly greater gain when compared to the bubble sort algorithm, even over larger data sizes.

IV. Conclusion

The MPI parallelisation when using a bubble sort algorithm provides speedup over the Matlab golden measure for larger datasets but not significant speedup when compared to other algorithms previously tested such as the median filter.

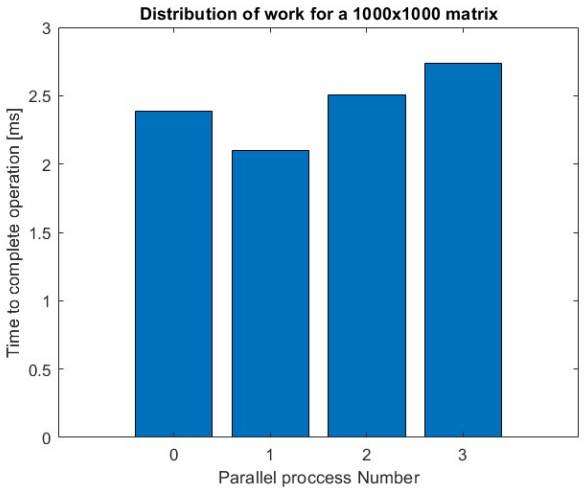

Fig. 11. Distribution 1000x1000

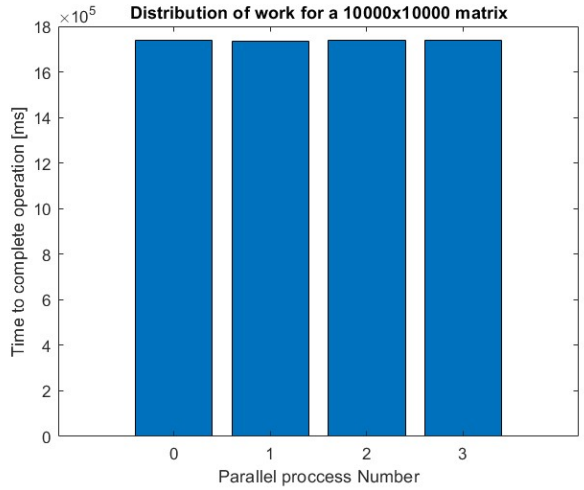

Fig. 12. Distribution 10000x10000

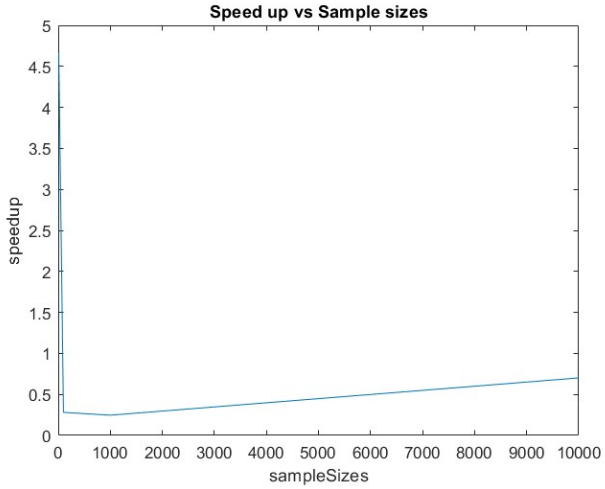

Fig. 13. Bubble Sort Speed Up

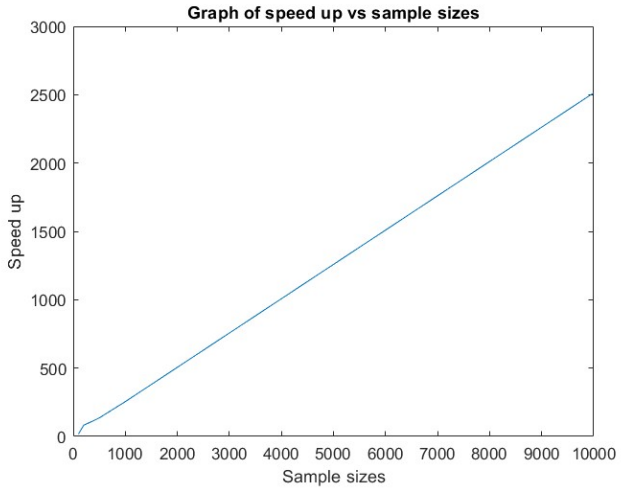

Fig. 14. Median Filter Speed-Up

In conclusion, while the parallelization of the bubble sort algorithm using MPI shows a speedup, especially for larger array sizes, further optimizations may be needed to yield significant performance gains. Additionally, the comparison with other parallelized algorithms highlights the shortcomings in the speedup achieved using the MPI parallelisation when compared to Matlab as the golden measure.

REFERENCES

[1] Ke Chen, Peter J. Giblin, A. Irving, “Mathematical Explorations with MATLAB, June 1999.