1 – Instructions & Introduction

This exercise requires me to work out the Pearson correlation coefficient of two data sets of equal size using MPI. These data sets are given in variables ***a*** and ***b*** in the program and are of type ***double***, these are dynamically created with sin(i) and sin(i + 5). This program will execute the correlation coefficient calculation serially on one process and then in parallel across multiple processes and the report will compare the speed of these results and explain the reasoning behind this result.

The attached file for this exercise is '**pearsonCorrelation.c**' and can be run by first entering the directory this file is stored in and entering the following command line instructions

1. 'mpicc –g –std=c99 –o pearsonCorrelation pearsonCorrelation.c' which creates the executable from the code file.
2. 'mpiexec –n 5 pearsonCorrelation 2000000' this will run the program, note '5' is the number of processes and can be chosen by the user and '2000000' is just the number of data elements in each array. (**IMPORTANT!**)

2.1 – Serial Implementation

The program has been split up into an *initialisation* and *calculation* stage.

Initialisation:

Initialisation is the stage where we get the data ready for calculation. Simply here we start a timer for the process, assign the memory for the dynamic arrays and run a for loop to assign elements of the arrays with *sin(i)* for ***a*** and *sin(i + 5)* for ***b*** and keep a sum of all elements in each array. Note that all of this code is run on a single process which is process zero:

```
// For loop to generate the data arrays
for (int i = 0; i < ARRAY_SIZE; i++) {
    a[i] = sin(i);
    totalSumA += a[i];
    b[i] = sin(i + 5);
    totalSumB += b[i];
}
```

Figure 1.1 – Serial Array Initialisation

Calculation:

We can use these total sum variables of each array to calculate the mean by dividing by the array size supplied in the command line arguments supplied by the user and then run a second for loop to work out the square differences between the elements and their arrays mean to work out standard deviation and then use the differences from the mean for each element in both arrays to allow us to easily calculate the Pearson correlation coefficient from this and the calculated standard deviations. Note we also use another timer in this section to work out the time taken for both initialisation and calculation *(This code starts from line 65 in the supplied code file).*

```
double meanA = totalSumA / ARRAY_SIZE;
double meanB = totalSumB / ARRAY_SIZE;
for(int i = 0; i < ARRAY_SIZE; i++) {
    distanceA = a[i] - meanA;
    distanceB = b[i] - meanB;
    squareDistanceA += distanceA * distanceA;
    squareDistanceB += distanceB * distanceB;
    productOfDifferences += distanceA * distanceB;
}
double standardDevA = sqrt(squareDistanceA / ARRAY_SIZE);
double standardDevB = sqrt(squareDistanceB / ARRAY_SIZE);
```

Figure 1.2 – Serial Calculation

2.2 – Parallel Implementation

Initialisation:

Parallel works for any number of processes and elements, we do this by calculating a variable for how many elements a process should have by integer dividing the array size by the number of processes and storing a remainder if necessary. Then in the initialisation for loop the code runs for how many elements the process should have and giving the last process the remainder elements if any exist. We use a **MPI_Barrier** statement to synchronise these processes to make sure the initialisation timer is right. Originally I used the recommended implementation of initialisting the arrays on one process but this led to much slower results, this is discussed this in the results section.

```
int localIteration = world_rank * ELEMENTS_PER_PROCESS;
int remainder = ARRAY_SIZE % ELEMENTS_PER_PROCESS;
if (world_rank != world_size - 1) {
    for (int i = 0; i < ELEMENTS_PER_PROCESS; i++) {
        a[i] = sin(localIteration + i);
        b[i] = sin((localIteration + i) + 5);
        totalLocalSumA += a[i];
        totalLocalSumB += b[i];
    }
} else {
    for (int i = 0; i < ELEMENTS_PER_PROCESS + remainder; i++) {
        a[i] = sin(localIteration + i);
        b[i] = sin((localIteration + i) + 5);
        totalLocalSumA += a[i];
        totalLocalSumB += b[i];
    }
}
```

Figure 1.3 – Parallel Array Initialisation

Calculation:

For each process we work out their local mean over the entire data set and then using the **MPI_Allreduce** functions we can sum these values to get a total mean.

```
MPI_Allreduce(&localMeanA, &globalMeanA, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&localMeanB, &globalMeanB, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

We can then perform the same calculation as serial but using this approach that accounts for remainder. We don't need to account for the local iteration value now as each processes has their own version of the array. So we can essentially use the same code as **Figure 1.2** here but include an else statement like **Figure 1.3** to account for remainder.

We can then use **MPI_Reduce** to sum the square distances and product of differences to get the standard deviations and Pearson correlation coefficient. To do these final calculations only one process needs to handle it for print so we use process zero again.

```
MPI_Reduce(&squareDistanceA, &standardDevA, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&squareDistanceB, &standardDevB, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&productOfDifferences, &sumOfDifferences, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if(world_rank == 0)
{
    standardDevA = sqrt(standardDevA / ARRAY_SIZE);
    standardDevB = sqrt(standardDevB / ARRAY_SIZE);
    double pearsonCC = (sumOfDifferences / ARRAY_SIZE) / (standardDevA * standardDevB);
```

Figure 1.4 – Parallel Calculation

3 – Experiments and Results

It isn't true to say that one method is faster than each other definitively and instead need to do some more experiments to get a clear answer

Number of Data Elements:

The total number of elements in the data arrays makes a huge impact on the speedup of the parallel version. For the designated 2 million elements with 5 processes we can see that the parallel program makes a big increase in speed.

```
================================================================================
                          COMP528-1: Parallel computation of Pearson correlation coefficient using MPI
================================================================================
Student Number: 200945941  Full Name: Jack Alan Taylor  Module: COMP528 Multi-Core and Multi-Processor Programming  Lecturer: Alexei Lisitsa
--------------------------------------------------------------------------------

Processes: 5    Elements per process: 400000     Remainder: 0

----------------------SERIAL IMPLEMENTATION----------------------
Array Length: 2000000, Pearson Correlation Coefficient: 0.283662
A: mean = 2.760418e-07, std = 0.707107
B: mean = 3.073144e-07, std = 0.707107
Initialisation completed in 0.673006
Calculation completed in 0.036913
Overall completed in 0.709919

----------------------PARALLEL IMPLEMENTATION----------------------
Array Length: 2000000, Pearson Correlation Coefficient: 0.283662
A: mean = 2.760418e-07, std = 0.707107
B: mean = 3.073144e-07, std = 0.707107
Initialisation Time: 0.142517
Calculation Time: 0.009871
Overall Time: 0.152388

----------------------SPEEDUP RESULTS----------------------
Initialisation Speedup: 0.530489                78.823814%
Calculation Speedup: 0.027042                   73.258742%
Overall Speedup: 0.557531                       78.534452%
```

Figure 1.5 – Program Results

Focusing on the lower "speedup results" section we can see an overall speedup of 78.53% but if we compile the same program with 200 elements we get -33411.04% speedup so this is massively slower but both computations are still pretty fast in computer time. **Figure 1.6** shows this relationship well, as the number of array elements increases the time taken increases for both parallel and serial but at higher elements the serial program increases the speed taken greatly even though it's faster than the parallel with a low number of elements.

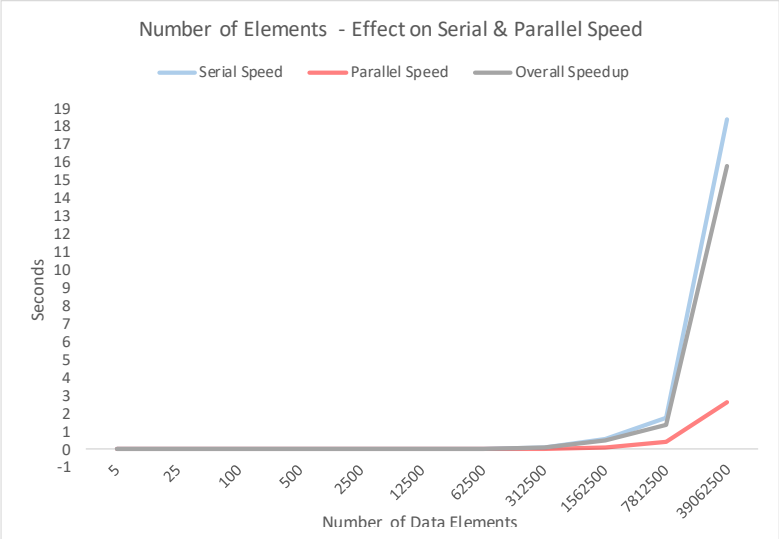

Figure 1.6 – Speedup for # of elements

This behavior is explainable, in the serial program one process has to perform all the calculations and initialisation but in the parallel program, processes share the task of initializing the array, calculating the mean, standard deviation and components for the Pearson coefficient.

I believe this behavior is the main reason the program speeds up, the highest amount of time in the serial application is spent on initialisation, this is obvious as the sin() function is called twice for each iteration slowing down the program massively, doing this in parallel where each process only handles a few sin function calls is a massive reason for the speedup.

The figure below provides evidence that this is the case. It can be seen that the speed up in initialisation outnumbers the calculation speed up. So the majority of our speed up comes from our initialization mechanism

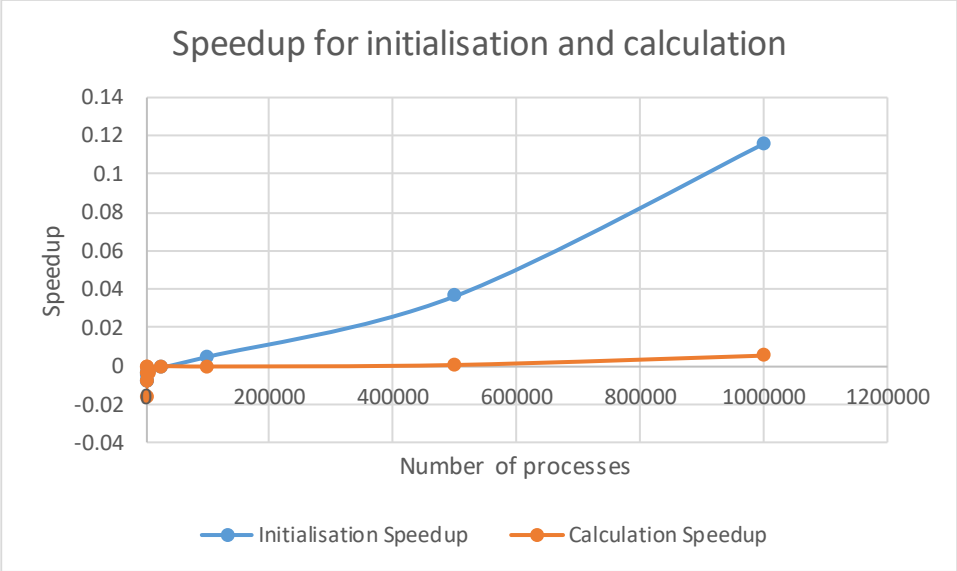

Figure 1.7 – Speedup for initialization and calculation

We speak about this further later in the report but this is the main reason why running the code the way initialization was recommended resulted in little to no speedup, because initialization would essentially be done the same way on one process with copies of full arrays on each process.

Number of Parallel Processes:

In the previous section we can see that due to multiple processes running tasks in parallel, that generally the program speeds up too. What number of processes would be optimal? Similar results were found no matter the data array size so the one graph below is enough to analyse this.

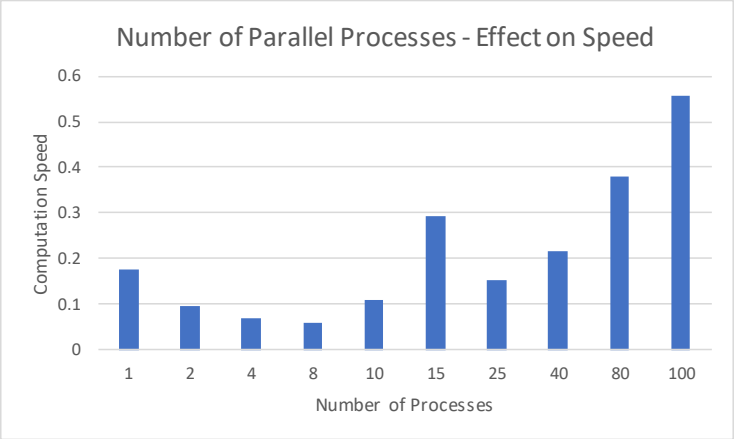

Figure 1.8 – Speedup for # of processes

We can see that we want a relatively small number of processes to gain optimal speed, but there is an interesting underlying analysis here. The number of hardware threads on the local CPU on the computer seems to be the optimal number depending on what computer you are using. This was tested on machine with 4 and 8 hardware threads.

Initialisation Methodology:

Originally as recommended in the specification we tried to initialise the array on one process but like the results show in the number of processes section it is much more efficient to split this initialisation so each process has their individual copies of the data arrays according to the number of data elements it should hold which is worked out programmatically according to the array size. Switching to this improved speed.

The inherent parallelism made this faster but there is another reason, computing MPI_Scatter and MPI_Gather on a large number of processes leads to a considerable overhead as copies of the full data array are needed for each which could be considered a limitation of MPI as it is designed for multiple systems.

Calculation methodology:

A small speedup came from changing how we computed the averages, originally we passed the sum of the arrays to one process which computed the mean for each which was relatively fast and still resulted in speedup but we changed this by computing a local mean on each process and summing that together with MPI_Reduce which increased the speedup.

This approach could possibly be taking to calculate standard deviations and overall Pearson coefficient rather than just calculating on the reduced result on one process.

Final Words:

In conclusion we can see from the above discussion that the parallel is generally faster with a "high" number of elements in the data array but slower when it's a low number, the number of processes is the inverse of this around the number of 4-8 seems to give the fastest results but it's all dependent on the processor threads itself but when we use a high number of processes it's slowed considerably. It's also shown how splitting up tasks like the initialization and mean calculation improves speed giving an insight of how to improve this further by parallelizing further the final standard deviation and Pearson correlation passes.

The program isn't written to read from a file specifically but through tests the program can compute an explicitly defined array of any elements and any size and gain an accurate correlation coefficient.