1 – Instructions & Introduction

This exercise requires me to work out the Pearson correlation coefficient of two data sets of equal size using OpenMP. 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 first part of this discussion will discuss the speedup of the parallel version to serial when using different parallel for scheduling, once we have the optimal scheduling we'll discuss how speedup varies with different values for threads and elements in the dataset. After, we'll make some comparisons between the MPI and OpenMP versions and show how speed up varies with different parameters in these different programs and then finally a conclusion section will document our overall findings from the project.

Three core code files have been submitted: **'mpi.c'** is a modified version of the first assessment which computes the task using only MPI and prints the speedup results. **'omp.c'** is the main code file for this project – it computes the task in openMP and computes speed up from it's parallel version against the serial and the MPI version by reading in the **'mpiresults.txt'** folder. The final code file is **"analysis.c"** this runs these programs iteratively to produce some averages that we can use for plotting graphs and diagrams in this report.

To compute this entire task there are a few main steps

1. First type in cmd 'mpicc –g mpi.c –std=c99 –o mpi -lm' which creates the executable from the code file for the mpi program.
2. Then to create an executeable for the OpenMP version type 'gcc –g –std=c99 –fopenmp –o omp omp.c –lm'
3. Compile this executable with './omp (data array size) (number of threads)' replacing the bracketed text with some numeric values which allows the user to run the program with dynamic values.
4. Finally use gcc to compile the analysis and run if that process needs to observed. (This wasn't fully completed due to bugs but I left it in to see what process I would have done if I had the time).

2.1 – Initialisation of the Algorithm

Before executing any of the core code we have a few steps set up that will need to be taken care of first. The '*convertUserInput*' and '*setNumberOfThreads*' functions deal with user input. The user will input the data size of the array and the amount of threads they want the OpenMP file to run on.

We use the argv/argc parameters of the core main function to manipulate this and convert them into numeric values. We must first check if any command line arguments have actually been submitted and if not we have code to detect this that will reset the numbers to some default values.

```
// Defaults
#define ARRAY_DEFAULT 5000000
#define THREADS_DEFAULT 16
```

Like in the first assessment noticeable speedup is observed when declaring the array over multiple processes so the next step of initialisation is to get a number for ELEMENTS_PER_PROCESS by using some division on the user defined array size and threads.

The only other things that are handeled here are the inline function declarations, global variables, library imports and constant declarations.

2.2 – Serial Implementation

The serial task is first handeled, this is very similar to the MPI approach, the only differences are the actual timing functions – now we use omp_wtime instead of the mpi functions. We have to core timers one initialisation timer which records the speed of initialising both of these arrays and the other times the calculation side of the algorithm.

To recap essentially we allocate size for the full array ***a*** and the full array ***b***, a for loop is iterated over to assign values of the user defined array size by assigning each element of the array ***a*** to *sin(i)* and assigning each element of array ***b*** to be *sin(i + 5)*.

```
// 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];
}
```

Once the arrays are set up the calculation timer starts and we compute an average for both arrays which is then used to compute the standard deviation and pearson correlation coefficient and the timer is stopped.

```
// Start calc timer
calculationTime_start = omp_get_wtime();

// Calculate means of the arrays
double meanA = totalSumA / ARRAY_SIZE;
double meanB = totalSumB / ARRAY_SIZE;

// Work out square distances from mean
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;
}

// Work out standard deviations
double standardDevA = sqrt(squareDistanceA / ARRAY_SIZE);
double standardDevB = sqrt(squareDistanceB / ARRAY_SIZE);

// Pearson correlation coefficient - covariance / stds
double pearsonCC = (productOfDifferences / ARRAY_SIZE) / (standardDevA * standardDevB);

// End calc timer
calculationTime_stop = omp_get_wtime();
```

The timings are then printed and stored to work with in the later stages of the program – we get three stored timing values the initialisation, calculation and summing these to get an overall time.

2.3 – Parallel Implementation

The parallel code uses the OpenMP library to perform parallel operations. This starts by taking the user input to define how many threads in the command line arguments.

To parallelise the array initialisation we make use of a parallel for reduce loop in OpenMP which allows us to sum across all threads. Each thread will create it's own local array but then the totals are summed so we can easily find the mean later on.

```
// For loop to generate the data arrays
#pragma omp parallel for reduction(+:totalSumA,totalSumB)
for (int i = 0; i < ELEMENTS_PER_PROCESS + remainder; i++) {
    a[i] = sin(localIteration + i);
    totalSumA += a[i];
    b[i] = sin(localIteration + i + 5);
    totalSumB += b[i];
}
```

To gather the mean we treat it as two separate "sections" in OpenMP meaning we can split the allocated thread resources to compute both the tasks in parallel. The first section would be to create the mean of A and the other to create the mean of B and a barrier is used to make sure both tasks are completed before moving on.

Following this we use another parallel for reduce loop to sum the square distances and product of their differences across all threads which is used to calculate the standard deviation and pearson correlation later on.

```
// Work out square distances from mean
#pragma omp parallel for reduction(+:distanceA,distanceB,squareDistanceA,squareDistanceB,productOfDifferences)
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;
}
```

From this the standard deviation tasks can also be treated as two sections to perform in parallel to find the std of A and B.

```
// Calculate means of the arrays
#pragma omp parallel sections num_threads(omp_get_num_threads())
{
    #pragma omp section
        standardDevA = sqrt(squareDistanceA / ARRAY_SIZE);
    #pragma omp section
        standardDevB = sqrt(squareDistanceB / ARRAY_SIZE);
}
```

Then from these values the perason correlation coefficient is computed. Note that the timers are handeled the same way as in the serial implementation.

2.4 – Speedup calculation

Now we have the timers for the serial and parallel algorithms we can use these to calculate speedup. The first task in this is to compute a ratio of speed up (serial / parallel) for each section so we can see where the speedup is gained/loss for the intial speedup/calculation speedup and overall speedup. As well as this we also compute percentage of speedup as in some cases it could be clearer than the ratio.

```
Serial Speedup: (Ratio and Speedup percentage)
Initialisation Speedup: 1.541999                35.149125%
Calculation Speedup: 1.719207                   41.833641%
Overall Speedup: 1.551675                       35.553523%
```

The figure above shows what a sample of this could look like. This was performed with 5000001 data elements on 2 threads.

The next task is to get outputs of the MPI version so we can make some more comparisons. This is done by using sprintf to run the Mpi version of the code but with the actual size of the array used in the OpenMP version so that the values are consistent. This is handeled in the ***mpiVersionExecute()*** code

```
void mpiVersionExecute()
{
    // Executes the MPI version
    if(omp_get_thread_num() == 0)
    {
        char cmd[100];
        sprintf(cmd, "mpiexec -n %d mpi %d", 8, ARRAY_SIZE);
        system(cmd);
    }
}
```

This mpi executable creates a file of the mpi's speed for initialisation, calculation and overall and stores that in a text file. This is then read in the OpenMP results section so we can produce the MPIs speedup showing the ratio and percentage speedup of the MPI version against the OpenMP version.

```
FILE* resultFile;
resultFile = fopen("mpiresults.txt", "r");
double timingArray[2];

for(int i = 0; i < 3; i++)
{
    fscanf(resultFile, "%lf", &timingArray[i]);
}

mpi_speedupRatio_init = timingArray[0] / p_initializationTime;
mpi_speedupRatio_calc = timingArray[1] / p_calculationTime;
mpi_speedupRatio_overall = timingArray[2] / p_overallTime;
```

This can then be printed to the terminal to intuitively see the speedup again:

```
MPI Speedup: (Ratio and Speedup percentage)
Initialization Speedup: 0.944953                -5.825410%
Calculation Speedup: 1.203991                   16.942900%
Overall Speedup: 0.967464                       -3.363020%
```

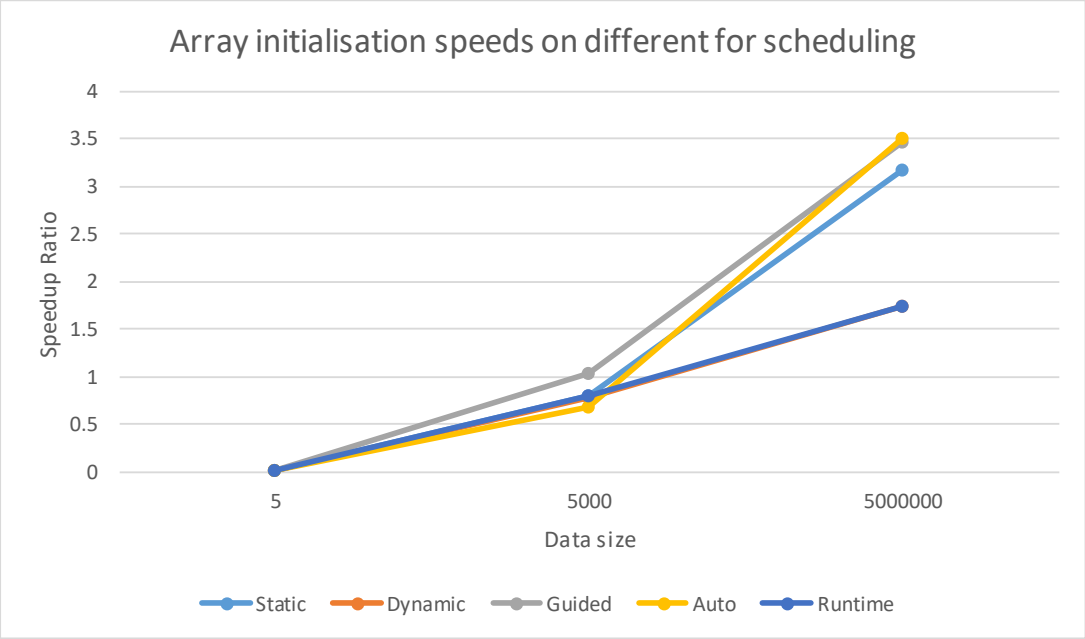
3.1 – Results/Experiments: For scheduling

Before discussing the results we need to ensure that the for loops in the parallel algorithm are running efficiently. In OpenMP there are many variations of scheduling so this section will show which are more powerful in this context.

To show this we ran the algorithm on all of the scheduling techniques over some different array sizes and averaged out the speed over some iterations. From this we constructed a table to show the results. (NOTE: Originally I tried to do this programmatically in ***analysis.c*** but a lot of errors came up and I didn't have the time to get it working – instead I had to do this manually hence the only few variations of data elements)

	5	5000	5000000
Static	0.022099	0.806369	3.176189
Dynamic	0.019718	0.782506	1.742544
Guided	0.021199	1.031541	3.460828
Auto	0.02532	0.687565	3.509148
Runtime	0.025093	0.803032	1.735841

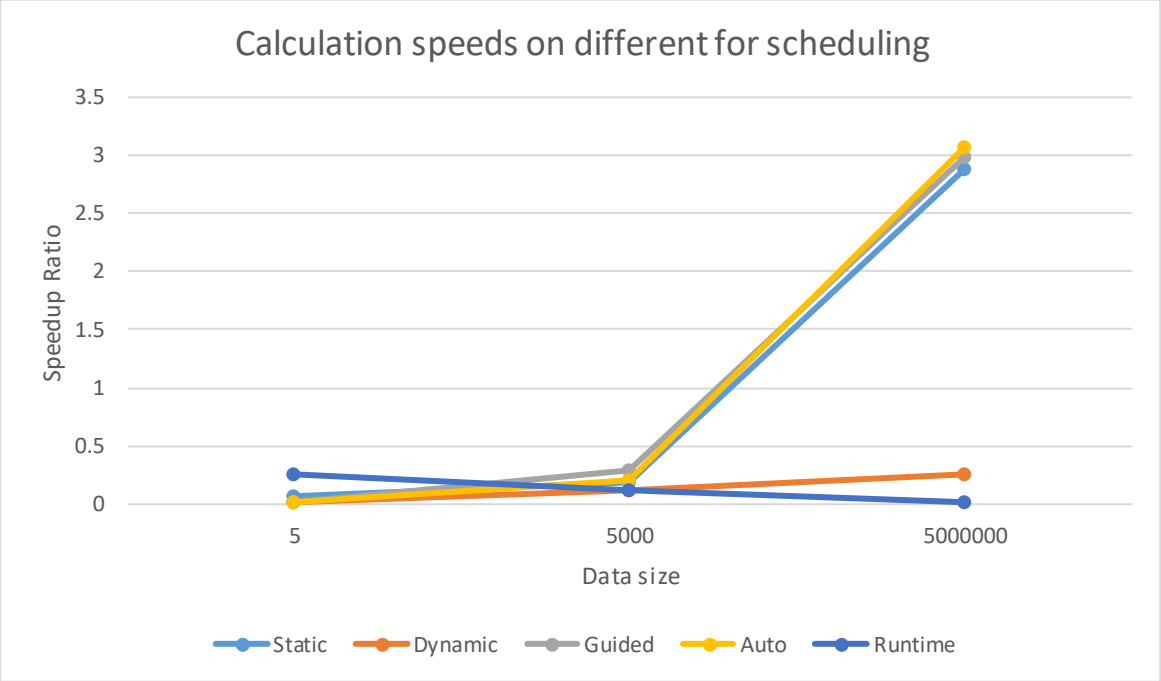
This inherently isn't very useful just showing the averages so we plotted this to see the relationships. The first plot shows the speedup (vertical) of different scheduling for initialisation of the data arrays.



It's obvious that there isn't a clear "best" method, different scheduling works well at different data array sizes. For working on data at all sizes then Guided would be the best but auto seems to be the best performing at high levels but the worst performing at low levels. For this reason we'll use the "Guided" schedulng which is similar to dynamic where the chunk changes size as needed but instead the chunk starts off large and decreases to load imbalance between iterations [1].

We can now see if these trends continue into the calculation for loop:

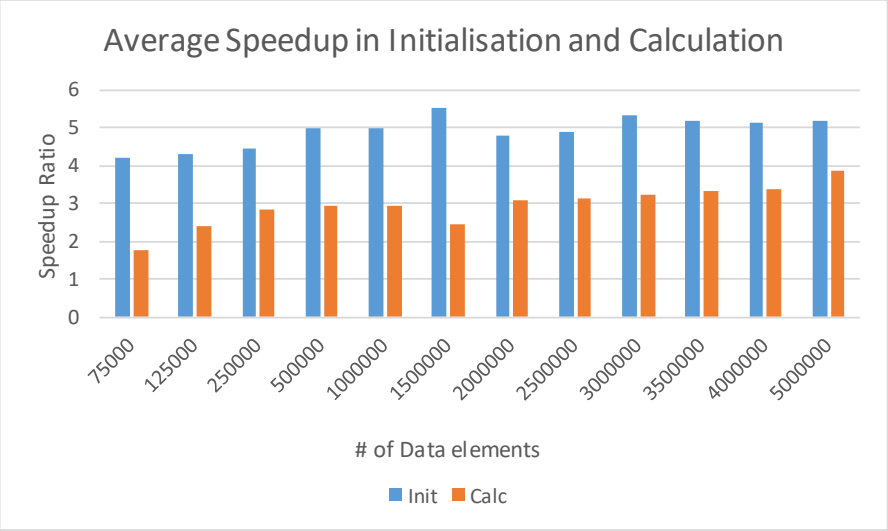
	5	5000	5000000
Static	0.057012	0.193198	2.88906
Dynamic	0.014143	0.121871	0.259422
Guided	0.01558	0.284506	2.987398
Auto	0.017373	0.210239	3.06421
Runtime	0.262343	0.115709	0.014712



We can see that again auto and guided generally seem to be the best choices, but maybe suprisingly the static version is now much higher performing than the initialisation step. Again we will use guided for this though as it seems to give a higher performance at all levels. The runtimes performance seems to be a bit of an anomaly early on as its performance decreases throughout.

3.2– Results: Parallel Performance

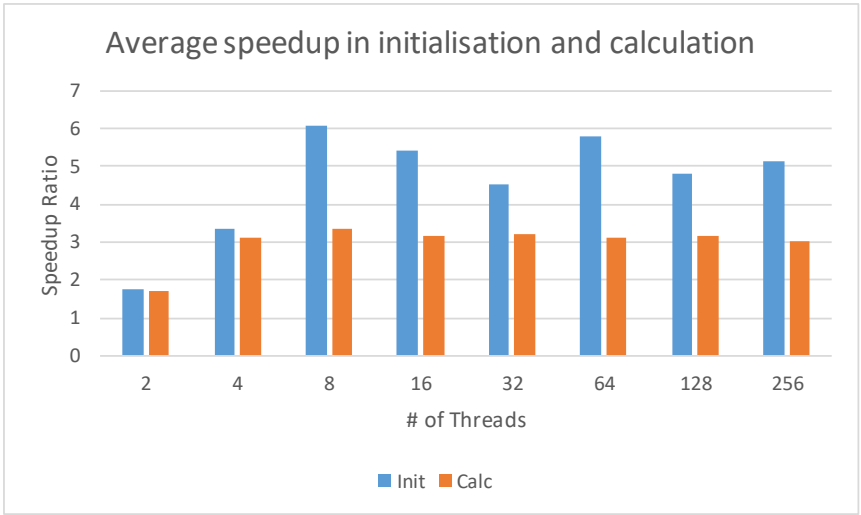
Now we have optimal performing for loops we can look at how the parallel performance compares against the serial. To do this we took our optimal serial code and compared at different levels of the parallel code. The results are recorded in ratios so anything below one would mean there is a negative speedup and anything above one means the parallel is faster.



	Init	Calc
75000	4.236014	1.773071
125000	4.3096	2.424887
250000	4.464774	2.838592
500000	5.022305	2.937361
1000000	5.022305	2.937361
1500000	5.543908	2.447532
2000000	4.788444	3.088779
2500000	4.892352	3.131044
3000000	5.357555	3.226941
3500000	5.183969	3.32712
4000000	5.130464	3.3927
5000000	5.180821	3.895383

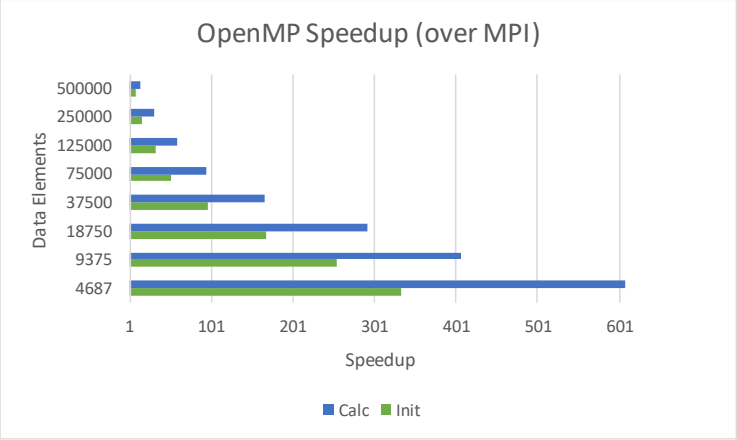
We used the same strategy of taking averages at different levels to plot. It's easy to see no matter how many elements that the parallel is faster from the graph, there is an exception to this ignored in the graph. Anything below 5000 data elements results in the serial being faster, but the parallel array initialisation is always faster. By looking at the results we can see that generally the more elements there are the greater the parallel performance is, which is what we should expect. We can also see that the intialisation makes much more of a factor than the calculation speed up but that both speed up considerably.

Now we want to take into account the number of actual threads used by the program so we perform the same experiment but sticking at 5 million data elements and differing the threads. These results show at all thread levels parallel performance increase but we also see a mirror in our conclusion of the MPI program. The peak performing number of threads seems to be the exact number of hardware threads available on the computer (8 here) and also tested on computers with 4 hardware threads.

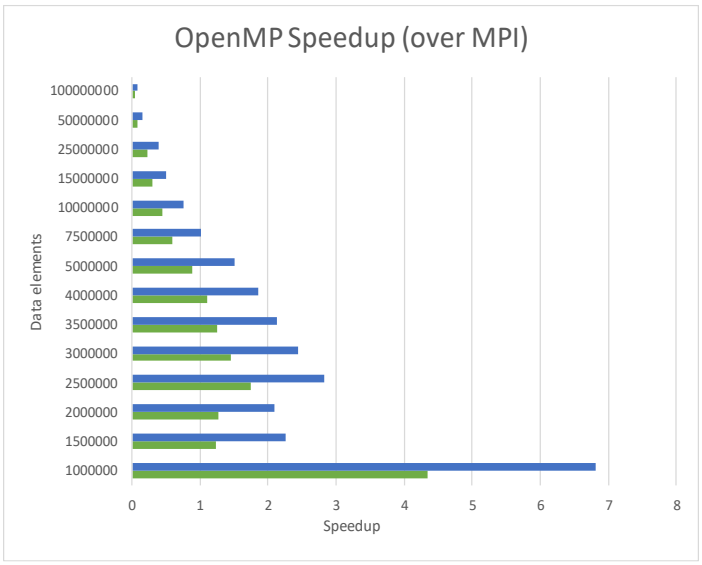


3.2– Results: Comparison with MPI

Because we execute our original MPI source within the OpenMP source code we can easily compare the speedup the OpenMP program has over the MPI. The first assignment required that we define the array over one process but this will give the same results as the graphs above as the initialisation would be serial so using this traditional way with scatter and gather the OpenMP will have much faster initialisation as shown. Instead I'll use the modified MPI program for comparison that defines the array over multiple processes and see which can most efficiently perform the dynamic initialisation task. As well as this we'll look at the speedup in the actual calculation. Note that these comparisons are made over the linux dedicated server with the optimal number of threads in both versions over the same amount of data elements in each recording.



Remembering that anything over 1 shows that OpenMP has a speedup (is faster than the MPI) then this first graph looks hopeful for OpenMP. It shows drastic speedup with a relatively small amount of elements 4687-500000 but we can definitely see the more elements there are that its getting slower. So we can now explore this relationship further.



Now this second graph shows something interesting, when we use 5 million data elements the initialisation in MPI actually starts to get speed up. The MPI starts to get faster, this trend also continues with the calculation algorithm. Any time we use elements over ten million the MPI starts to speed up in initialisation and calculation. And using 100 million we get some significant speedup.

This is an interesting relationship we can make some conclusions from but maybe as a general rule we should handle the data array initialisation with MPI as it sees speedup at lower levels of elements than the calculation algorithm does.

4– Conclusions

The clear conclusion in both cases of MPI and OpenMP is that when working with relatively large data sets paralleism can improve efficiency in all areas of processing. The speedup at all levels besides very low number of data elements in both reports shows this.

We've also gathered that in the case of OpenMP that the number of hardware threads on the PC seems to give the best performance when setting the number of threads the program will run on – this is similar to the conclusion for MPI which said the same for the number of processes.

More interestingly we can conclude that both platforms are powerful computation but that OpenMP and MPI have their advantages and disadvantages. OpenMP is relatively simple to build with – it can be easily placed in a serial program and have some good performing parallel programs and for the more simple and less computation based tasks this seems to be the quickest way to implement a powerful program, it also outperforms MPI at a smaller level of parallelism. On the other side OpenMP seems to fall off in performance when working with very large datasets and large computation whereas MPI keeps being powerful, MPI is less natural to program with and has to undergo more consideration when developing a program rather than just plugging it into serial code but it seems to provide very powerful results that can deal with the high end of big data. We backed this up in the report by showing that OpenMP held greater performances with up to 10million data elements but anything after that then MPI outperformed OpenMP and still outperformed it on a very high scale. Relating to that we can say both platforms are powerful but OpenMP does not have as much scalability as MPI.