ASSESSING THE PERFORMANCE OF MPI WHEN CALCULATING THE PEARSON CORRELATION COEFFICIENT

COMP528 - ASSIGNMENT 1

The Pearson Correlation coefficient is a measure of linear correlation between two variables and is defined as:

$$\rho(X,Y) = \frac{\mathbf{Cov}(X,Y)}{\sqrt{\mathbf{Var}(X)\mathbf{Var}(Y)}}.$$

The above formula requires the following calculations: $\bar{x}$ and $(x_i - \bar{x})$ and similarly for y. Hence an algorithm using this approach must iterate through each array twice.

The formula can be rearranged, such that only one iteration of each array is required:

$$\rho(X,Y) = \frac{\sum xy - \frac{\sum x \sum y}{n}}{\sqrt{\sum x^2 - \frac{(\sum x)^2}{n}}\sqrt{\sum y^2 - \frac{(\sum y)^2}{n}}},$$

where n is the number of elements in the array.

Using this formula, an algorithm would need to calculate: $\sum x, \sum y, \sum xy, \sum x^2, \sum y^2$.

Program structure - See Pearson1.c for code.

The first step of the program is to create two arrays for X and Y and populate them with values such that: $x[i] = \sin(i)$ and $y[i] = \sin(i+2)$. This is a serial algorithm, only performed by process 0.

Serial Program

(1) Use a for statement to iterate through X and Y to calculate $\sum x$, $\sum y$, $\sum xy$, $\sum x^2$, $\sum y^2$

(2) Calculate the Pearson correlation coefficient with the above formula

Parallel Program

(1) Use `MPI_Scatter` to send a section of the X and Y arrays to each process. Each process stores these in two new arrays: `local_x` and `local_y`. The size of these sections is equal to: the size of the X array / number of processes.

(2) Each process calculates the five summations (listed above) for their section of the array

(3) Use `MPI_Reduce` to calculate the summations of the entire arrays

(4) Calculate the Pearson correlation coefficient. This is only performed by process 0, and hence is in serial.

Initial observation

The size of the array sent to each process is calculated as $\frac{the\ size\ of\ the\ array}{number\ of\ processes}$. When the number of processes is not a factor of the size of the array, some elements of the array are excluded from the calculation.

For example, assume the size of the array is 10 and the number of processes is 6. The size of the array sent to each process will be: $\frac{the\ size\ of\ the\ array}{number\ of\ processes} = \frac{10}{6} \Rightarrow 1$, as this an integer variable. Each process will receive one element of the array and hence 4 elements are excluded from the calculation.

Note: for this particular exercise (e.g.calculating the Pearson correlation coefficient where $x[i] = \sin(i)$ and $y[i] = \sin(i+2)$ on an array of size $2,000,000$), excluding just a handful of elements from the calculation does not affect the overall result. This is because the points plot an ellipse (given the cyclical nature of $\sin(i)$) and hence the correlation quickly converges to the value -0.416 as the size of the array increases.

I have explored two methods to resolve this issue:

(1) Increasing the length of X and Y such that the size of the arrays are divisible by the number of processes, assigning the additional elements with a value of 0.

(2) Using the `MPI_Scatterv` function

Method 1 - See Pearson2.c for code.

If the number of processes is not a factor of the array size, we increase the size of the array so that it is divisible by the number of processes. For example, assume the size of the array is 10 and the number of processes is 6. The array is increased to size 12. These 2 additional elements are given the value of 0. Now when the array is scattered, each process receives 2 elements.

When we calculate the Pearson correlation at the end of the program, we set n = the original array size, and hence the calculation remains unchanged.

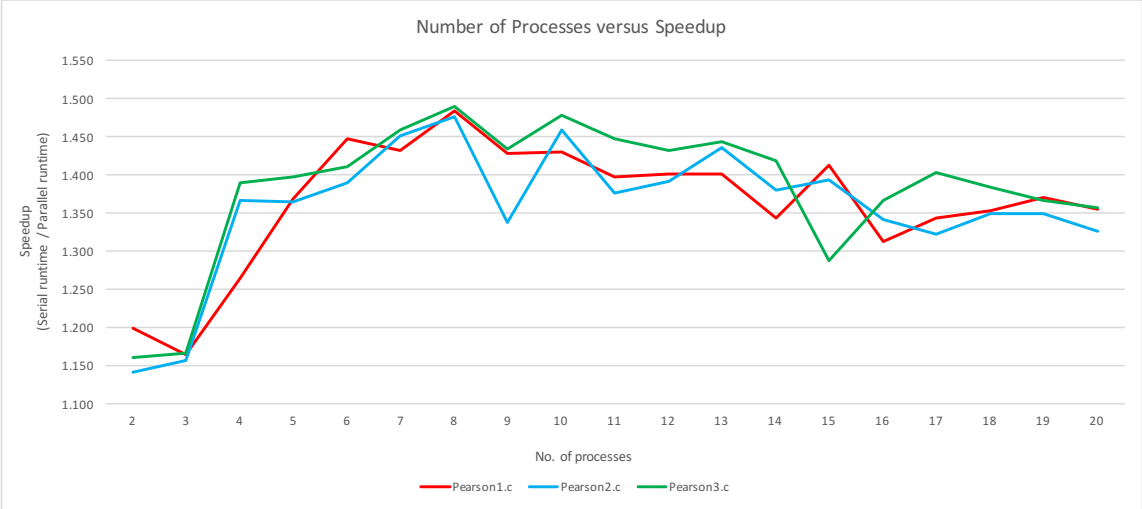
Method 2 - See Pearson3.c for code.

The `MPI_Scatterv` function allows different sized chunks of the array to be sent to the processes. The function receives two arrays: one with the number of elements to send to each process and one with the point along the array to start sending the data (referred to as the displacement).

Results

Speedup

Each program was run 10 times with the number of processes set from between 2 to 20. The graphs included in this report use the average value over the 10 runs. The computer used was a MacBook Pro with i7 processor.



The graph shows that the maximum speedup is achieved when 8 processes are implemented, with a value of 1.490.

The speedup increases up until this point and then decreases slightly as the number of processes is increased up to 20. We would not expect to see this fall in speedup, but instead expect the speedup to plateau after reaching its maximum. This is likely to be an indication of the limitations of the computer being used, than the actual speedup which could be obtained by the algorithm.

There are two notable falls in the speedup: program Pearson2.c, 9 processes and program Pearson3.c, 15 processes. Again these are likely to be due to a reduced level of resource available at the runtime as opposed to the actual speedup of the algorithms.

Over all, the algorithm in Pearson3.c gives a marginally higher speedup compared to the other programs.

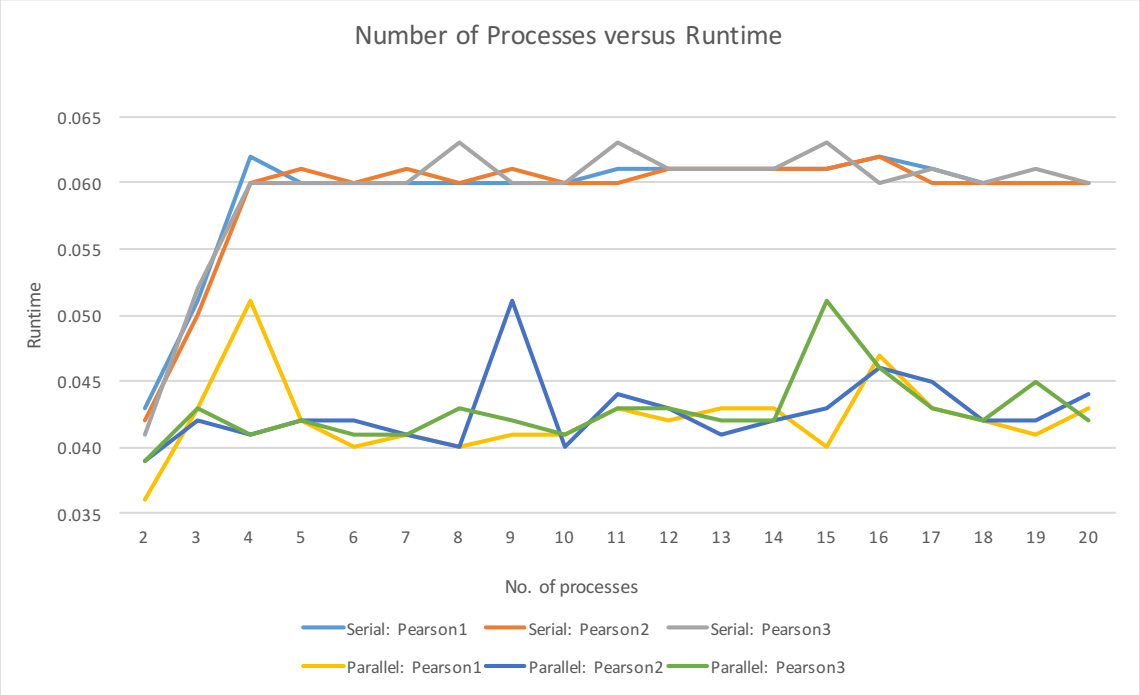
Amdahl's Law states that the theoretical speedup can be calculated as:

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)},$$

where n = number of processes and B = the fraction of the algorithm that is serial. This could suggest that 38% of the algorithm was run in parallel (based on the maximum speedup obtained: 1.490 with 8 processes).

Runtime

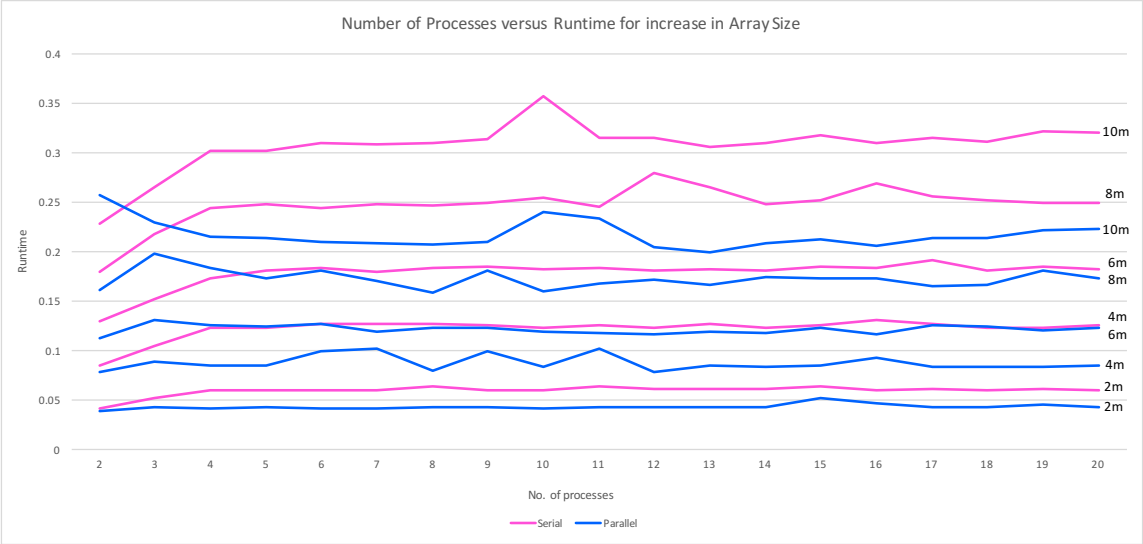
This graph shows the runtime of the serial and parallel codes:



These results show a slight anomaly in the data. Notice that the serial runtime increases from 2 processes to 4 processes, and then remains fairly constant; when we might expect the run time to be constant over all numbers of processes. Perhaps this is due to running a serial algorithm inside a program set up to run in parallel (e.g. the MPI library is called and multiple processes assigned). This would be an interesting point to investigate further.

Increasing the array size

I also ran the Pearson3.c code increasing the size of the X and Y arrays by 2,000,000 up to 10,000,000. The graph below shows the runtime results:



These results demonstrate that as the array size increases, the parallelisation is more effective. For example, it is quicker to process an array of size 10m in parallel, than 8m in serial.

It is interesting to note that in general the runtime time does not reduce after around 5 processes. Perhaps this is due to the computer being used, or perhaps that at this point the grain size is optimised.

Conclusion

My results showed that the Pearson3.c program gives a marginally higher speedup, however the runtime across the three programs is fairly similar. I also found that the benefits of using a parallel algorithm increased when the size of the dataset increased.

Further investigations could include: understanding the anomaly in the serial code runtime for processes 2 to 4, and running the programs on different computers, with more cores and processing power, to see if this produces different results.