# COMP528 Multi-Core and Multi-Processor Programming

Assignment 1: Computing the Pearson correlation coefficient

Maciej Lechowski

m.j.lechowski@liverpool.ac.uk

## 1 Description

Program calculating Pearson coefficient was written in C and tested on Linux (Ubuntu 14.04) with MPI installed (mpicc for MPICH version 3.0.4 ).

It consists of several source code (*.c) and header (*.h) files. In order to ease compilation, make file was prepared with all the commands needed to successfully produce executable file. Therefore, to compile the program a user needs to be in a directory with a "makefile" present and execute following commands:

```
$ make clean
$ make
```

After that, a number of *.o files should appear in the same folder, together with the executable file called `pearson`.

To run it conveniently, I have prepared a shell script (run_pearson.sh).

Command:

```
$ ./run_pearson.sh
```

makes the program run with default parameter n = 2. To specify number of processes type:

```
$ ./run_pearson N
```

where `N` is an integer denoting number of processes.

# 2 MPI_Send and MPI_Recv vs MPI_Bcast

First version of the program I have written used `MPI_Send` and `MPI_Recv` functions to exchange data between processes. As program prints amount of time needed to complete each step (ie., calculating mean, standard deviation and Pearson coefficient), I've noticed that when using `MPI_Send` and `MPI_Recv` functions time needed to exchange data tends to vary between program runs. Although usually it takes about 0.1 ms, every now and then it can raise to as high as 3-4 ms. That's why I've also written a version using `MPI_Bcast` function. It turns out it is more stable: rarely raises to a higher value than 0.1-0.15 ms. What this means is that using `MPI_Send` and `MPI_Recv` may be dangerous in some cases, as waiting for completion of those functions may become a bottleneck. Changing which functions should be used is available in `definitions.h` file: constant `USE_BROADCAST` set to 1 means that `MPI_Bcast` will be utilised during runtime.

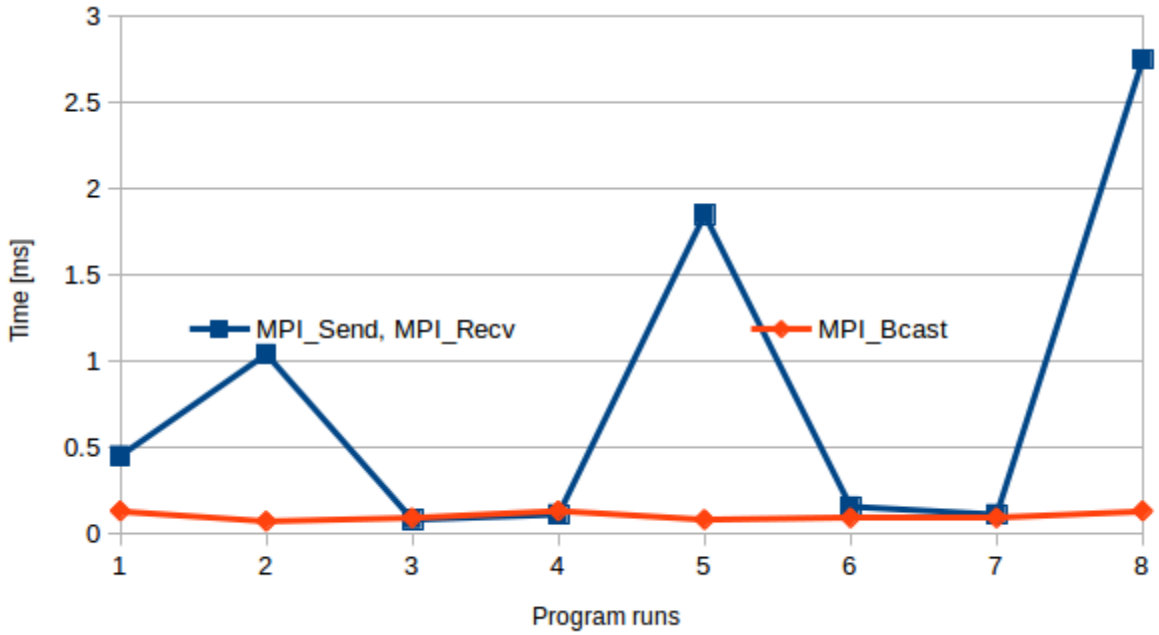


Figure 2.1. Two lines plotted on the chart visualise observations described above. Using `MPI_Bcast` tends to be more stable, whereas `MPI_Send` and `MPI_Recv` tend to have their execution time increased across different runs.

# 3 Using MPI_Scatter

Initially, I've used `MPI_Scatter` to distribute arrays to each process. Then local value was computed, and after that the global value was obtained with `MPI_Reduce`. That approach turned out to be inefficient.

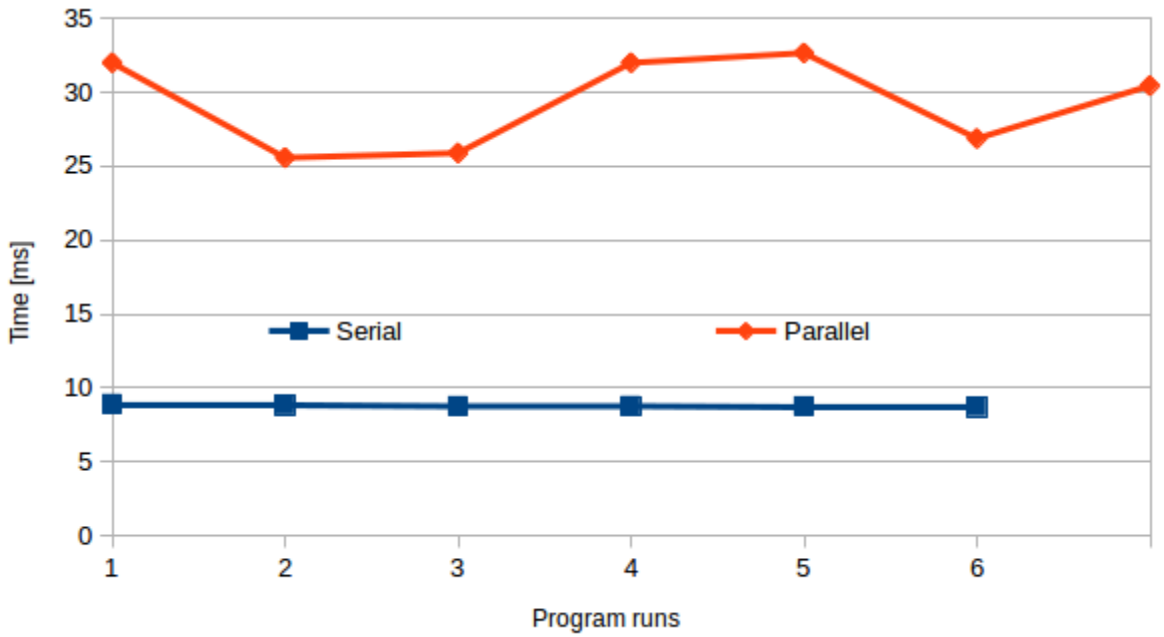


Figure 2.2. This graph presents results for both serial and parallel versions over several program runs. It uses an array with 1,000,000 values, `MPI_Bcast` function as well as `MPI_Scatter`/`MPI_Reduce`. Although it was tested on a laptop with a 2-core Intel i5 processor, lack of speed-up was also easily noticeable on the Chadwick cluster. More detailed information about execution in this approach are presented in tables 2.1 and 2.2 for serial and parallel version, respectively.

Table 2.1. Table includes detailed information about the execution time of each step in the **serial** algorithm.

| | **Mean [ms]** | **Standard deviation [ms]** | **Pearson coefficient [ms]** | **Overall [ms]** |
|---|---|---|---|---|
| Run #1 | 2.76 | 3.71 | 2.27 | 8.74 |
| Run #2 | 2.80 | 3.73 | 2.35 | 8.88 |
| Run #3 | 2.81 | 3.70 | 2.29 | 8.80 |

| | | | | |
|---|---|---|---|---|
| Run #4 | 2.76 | 3.71 | 2.27 | 8.74 |
| Run #5 | 2.83 | 3.67 | 2.25 | 8.75 |
| Run #6 | 2.86 | 3.57 | 2.30 | 8.73 |
| Run #7 | 2.73 | 3.70 | 2.25 | 8.68 |
| **Average** | **2.79** | **3.68** | **2.28** | **8.76** |

Table 2.2. Table includes detailed information about the execution time of each step in the **parallel** algorithm. Clearly, computing mean and standard deviation takes bulk of the overall execution time.

| | **Mean [ms]** | **Standard deviation [ms]** | **Pearson coefficient [ms]** | **Broadcast 1 [ms]** | **Broadcast 2 [ms]** | **Overall [ms]** |
|---|---|---|---|---|---|---|
| Run #1 | 14.97 | 15.37 | 1.41 | 0.13 | 0.13 | 32.01 |
| Run #2 | 13.81 | 10.20 | 1.43 | 0.07 | 0.07 | 25.58 |
| Run #3 | 13.49 | 10.85 | 1.40 | 0.09 | 0.07 | 25.90 |
| Run #4 | 14.97 | 15.37 | 1.41 | 0.13 | 0.13 | 32.01 |
| Run #5 | 15.28 | 15.75 | 1.47 | 0.08 | 0.09 | 32.67 |
| Run #6 | 12.34 | 12.20 | 1.49 | 0.09 | 0.76 | 26.88 |
| Run #7 | 13.89 | 14.81 | 1.59 | 0.09 | 0.08 | 30.46 |
| **Average** | **14.11** | **13.51** | **1.46** | **0.10** | **0.19** | **29.36** |

# 4 Optimizing parallel algorithm

As tables 2.1 and 2.2 clearly show, parallel algorithm created in this manner does not produce any speed-up neither on a laptop nor on a cluster.

That's why I've changed my approach in design of the parallel algorithm. Instead of scattering array with values, program calculates indices for each process. Then every process computes only its part of the whole array. After that, `MPI_Reduce` function is used to obtain single, global value.

**Example 4.1**

If input array consisted of 12 values and there were 4 processes available, each process would receive 3 values. Indices of the array would start at 0, 3, 6 and 9 for processes 1, 2, 3 and 4, respectively. The idea is visualised on Figure 4.1.

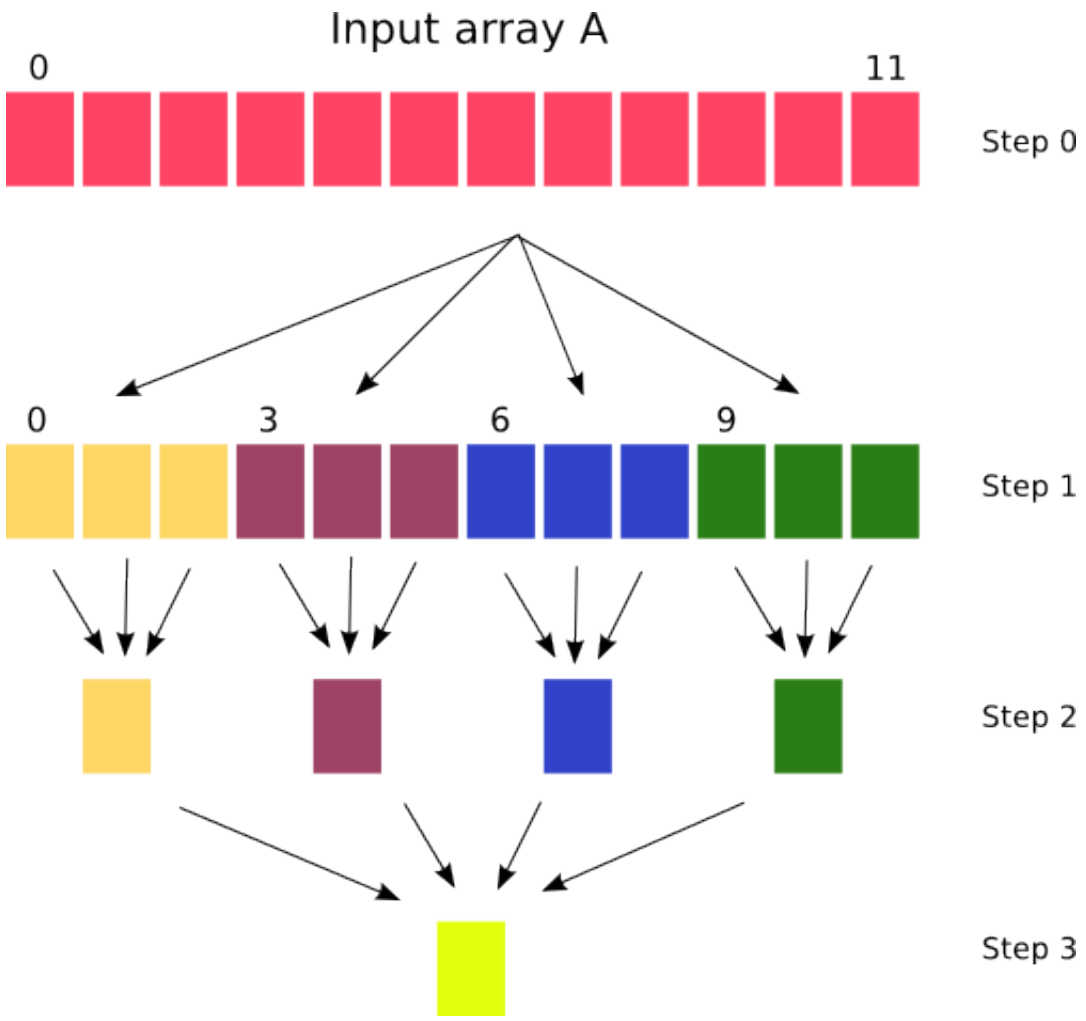


Figure 4.1. Diagram presents distribution of the input array (pink rectangles, step 0) to four processes (data that will be computed by each process are denoted by different colours, step 1). Then, each process needs to finish its computations (step 2). In the end, results from each process are reduced to a single result (step 3).

As it turns out, this approach brings speed-up even on a personal laptop. In source code I've attached, computing mean and standard deviation in this manner can be obtained by setting constant `USE_PARALLEL_INDICES` to 1 (in `definitions.h` file).

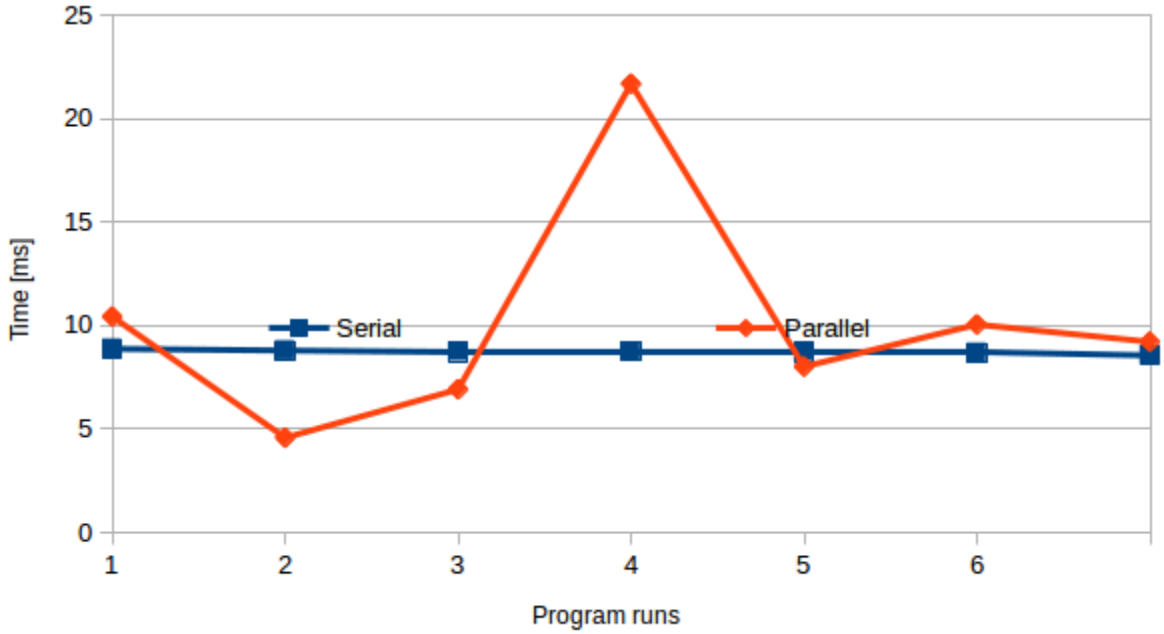


Figure 4.2. Chart presents time needed to obtain results using both serial and parallel algorithm (in the version discussed above). In this case 4 processes were used. At times, almost 2x speed-up was achieved (run #2). Running 4 processes in this case resulted in inconsistent behaviour (4th run finished after over 20 ms).

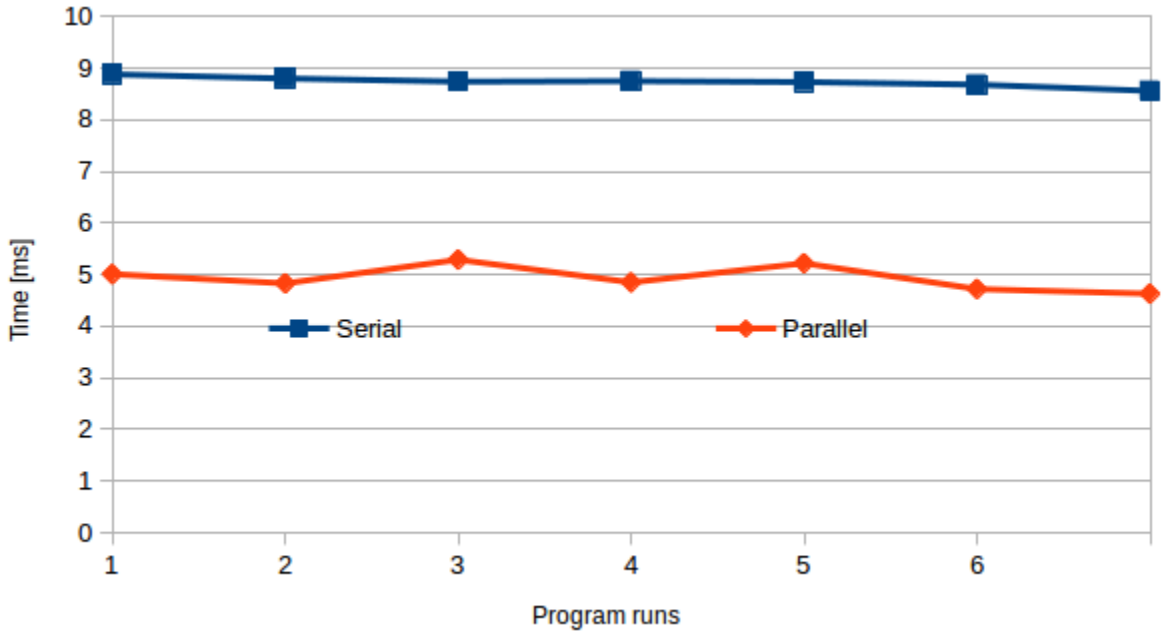


Figure 4.3. It turns out that best speed-up is achieved when running the program with

2 processes (which is logical, as the computer is equipped with Intel i5 processor with two cores). As shown above, time of the parallel algorithm in this case is stable and the program finishes in about 5 ms, or roughly half the speed of the serial algorithm.

# 5 Measuring time of the execution on the Chadwick cluster

I have measured time needed for computation on the Chadwick cluster.

Unsurprisingly, speed-up rate in this case was larger. Best results were observed when running the application with 8 processes.

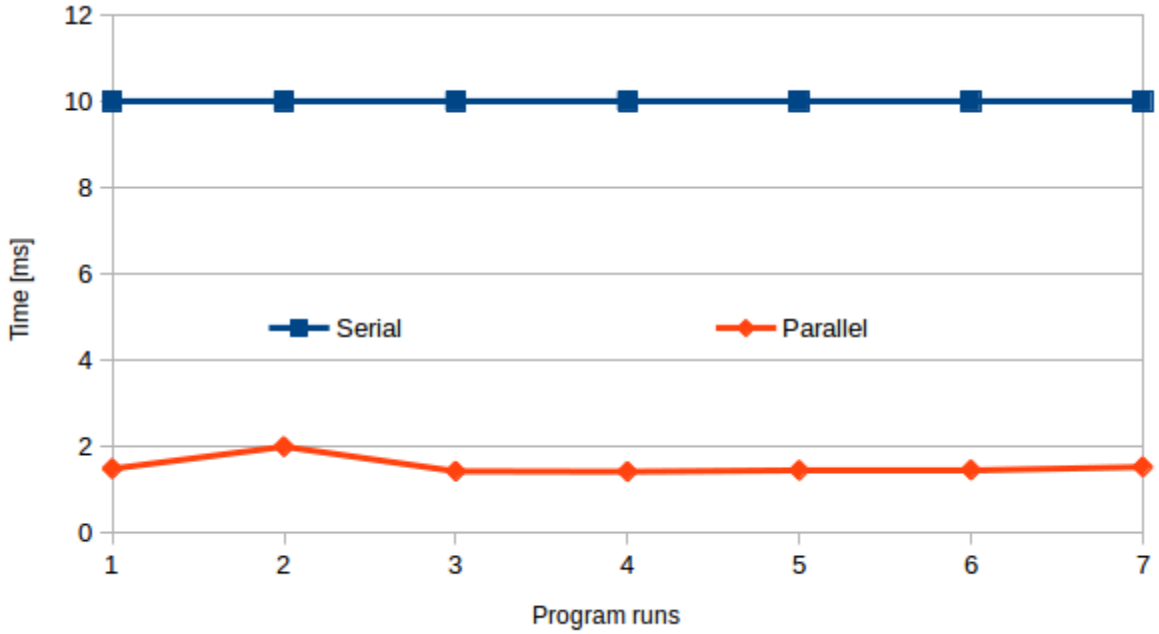


Figure 5.1. Graph shows results of running application on the Chadwick cluster with 8 processes. Parallel algorithm consistently finished executing under 2 ms. The serial algorithm, strangely enough, was usually set at 10 ms.

Even greater speed-ups were achieved when computing larger arrays. Figures 5.2 and 5.3 show execution times of array consisting of 100,000,000 values using 8 and 16 processses, respectively.

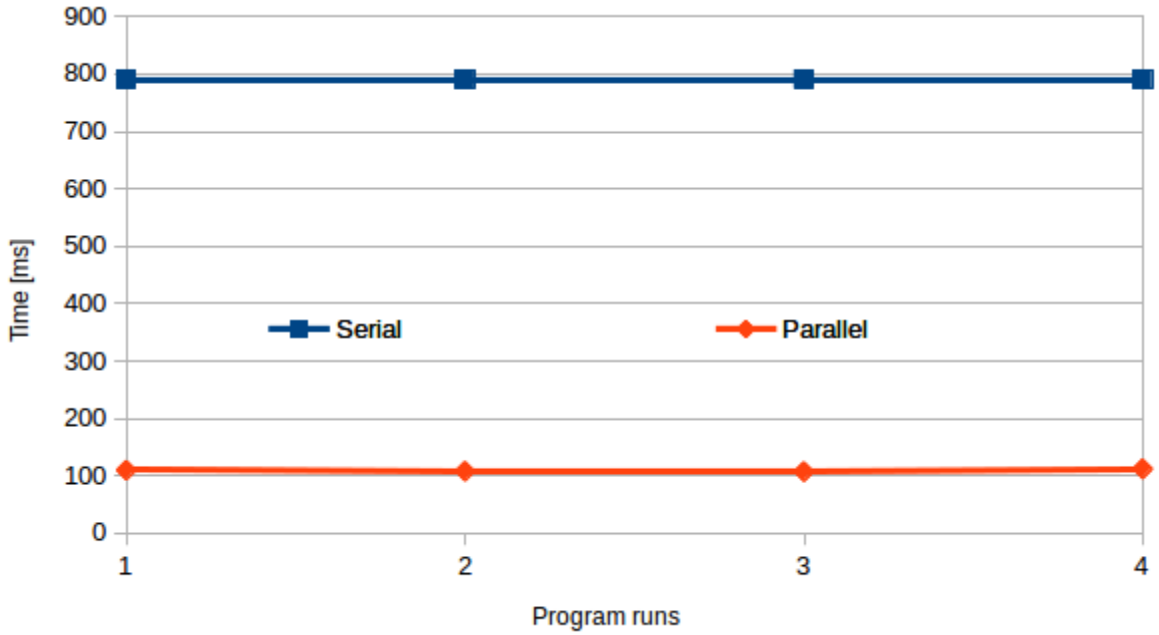


Figure 5.2. Graph shows times of several program executions on Chadwick, when utilising 8 processes. Again, results tend to be most stable when the amount of processes is equal to the number of cores in the processor. Speed-up in this case roughly eightfold.

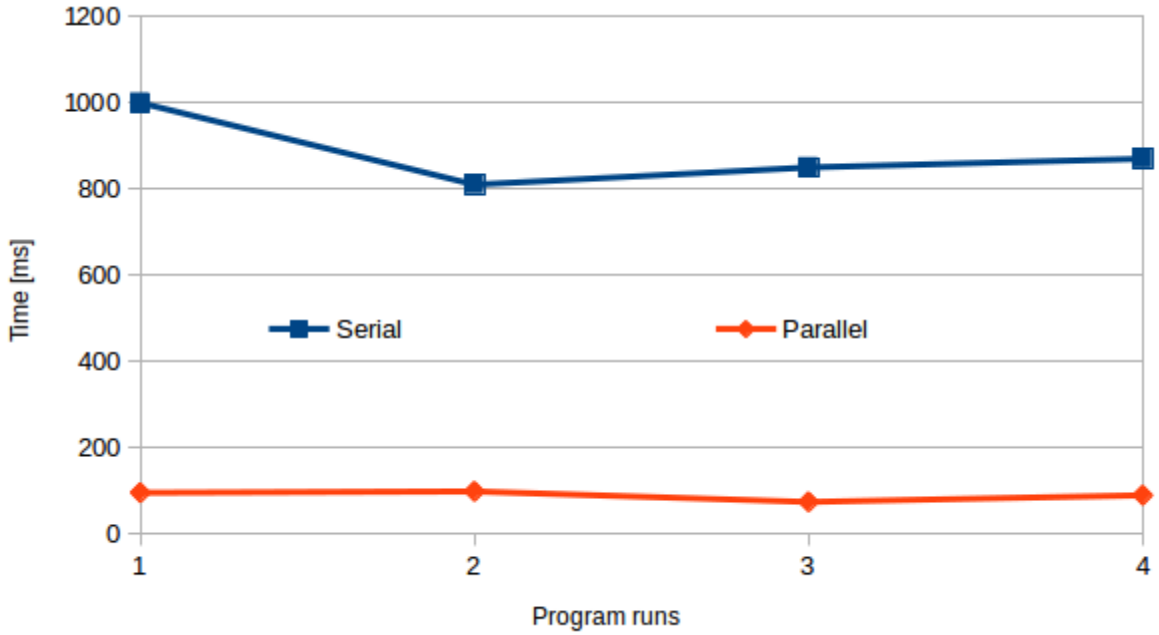


Figure 5.3. This chart is the same as 5.2, except in this case 16 processes were used. Although the speed-up is more or less similar, results tend to slightly less consistent.

# 6 Conclusions

In the previous sections I have described a method of achieving a speed-up when computing Pearson coefficient. My first approach, which involved the use of `MPI_Scatter`, proved to be inefficient (no speed-up was observed).

However, as discussed in section 4, I was able to optimise the parallel algorithm to achieve substantial decrease of execution time. Dividing the input array among different processes brought noticeable speed-up on a personal computer as well as a cluster.