C1424548

CM3103: High Performance Computing

Professor David Walker

---

# Programming with MPI

---

December 2, 2017

# Contents

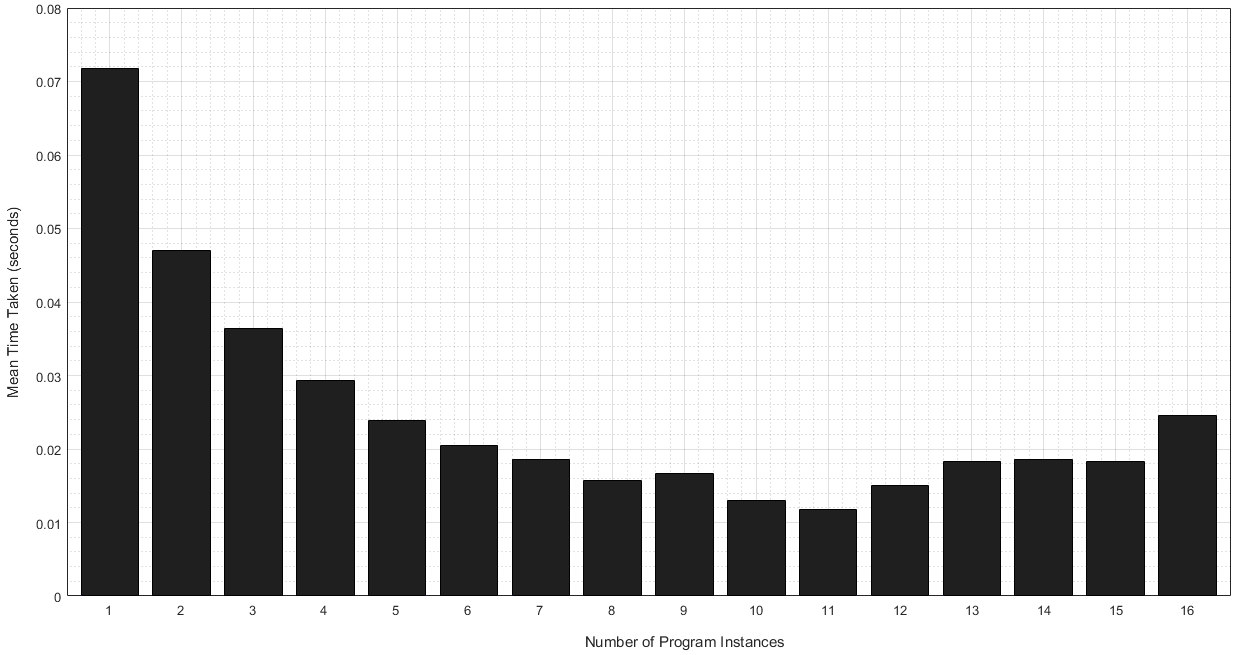
# 1 Environment and Methodology

I completed the MPI code on my windows PC at home, then used WinSCP to transfer the source to the Linux lab computers, and the PuTTY SSH client to remotely access my account. I then followed the instructions in *Running_MPI_In_Linux_Lab.pdf* (from Learning Central) to set-up the environment and compile the program.

Next, I wrote and executed a simple bash script to automatically run the program with different configurations, and collate the results. I set it up to run the program at instance counts from 1 to 16, for 100 iterations each, and calculate the mean outputted times. I ran multiple iterations to achieve higher accuracy, and to reduce noise from external factors.

Finally, I loaded the data into MATLAB and generated a simple bar chart, which is included in the next section. I chose a bar chart because the x-axis is discreet, but still allows you to see patterns over multiple bars.

# 2 Results



| N | Time (s) |
|---|---|
| 01 | 0.07174568 |
| 02 | 0.04695763 |
| 03 | 0.03638140 |
| 04 | 0.02929049 |
| 05 | 0.02383108 |
| 06 | 0.02051633 |
| 07 | 0.01862701 |
| 08 | 0.01567072 |
| 09 | 0.01664836 |
| 10 | 0.01298813 |
| 11 | 0.01174641 |
| 12 | 0.01497631 |
| 13 | 0.01834113 |
| 14 | 0.01860799 |
| 15 | 0.01832369 |
| 16 | 0.02462203 |

The graph above shows the mean time (y-axis) output by the code at different program counts (x-axis). The table on the left shows the exact values from the tests, with the $N$ column representing the number of program instances, and the *Time* column showing the mean outputted time in seconds.

### 2.1 Analysis

The overall U shape of the data, with a minimum at 11, illustrates two competing forces: The speed-up from splitting the workload, and the increasing overheads from communication between machines and other parts of the MPI library. The minimum then is the optimum balance between these two effects. The curve is also not totally smooth. For instance, 9 sticks up noticeably from the surrounding values, and there is no smooth ascent along 14, 15, 16. I will explore this more after showing my performance model.

## 3 Performance Model

To create an accurate performance model I used MATLAB's Curve Fitting Tool. This tool allows you to enter a function of x with unknown constants, and have it optimise the unknown values to yield the curve that most closely fits your data.

After some experimentation, I settled on a custom equation $t = an^2 + bn + c + \frac{d}{n}$, where $a, b, c, d$ are constants to be filled in, $t$ is the time taken, and $n$ is the number of program instances. This combines a standard quadratic equation and $\frac{d}{n}$.

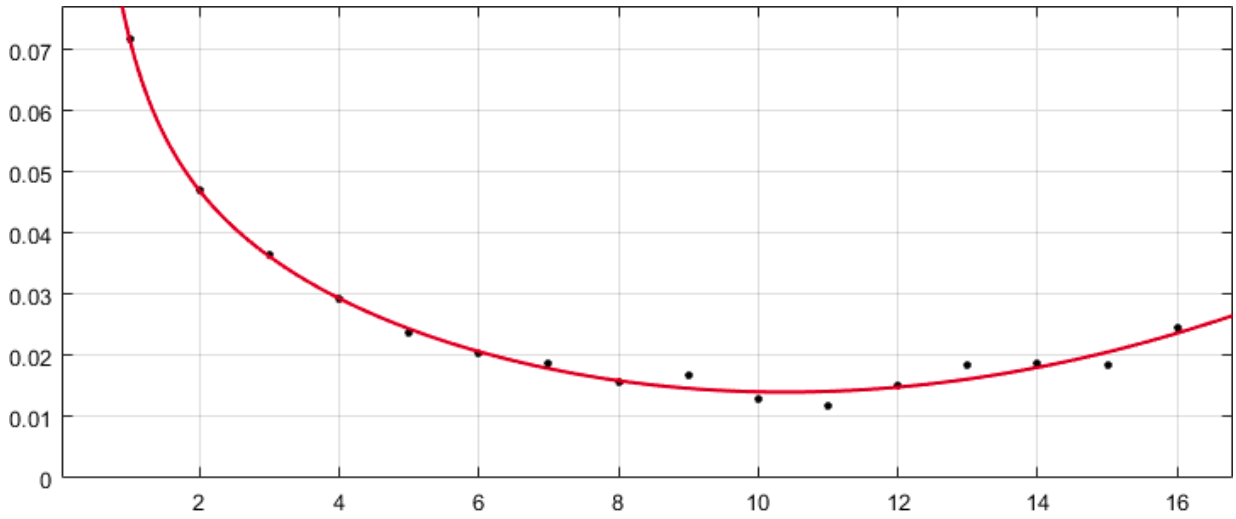
My starting point was the $\frac{d}{n}$ term, because MPI is literally dividing the workload over $n$ processors. This term on its own actually yielded a very good fit for the first half of the data, but it could not model the turnaround after 11 instances where the overheads start outweighing the benefits. I tried subtracting various different curve functions to model the performance loss from overheads, and in the end found that a simple quadratic fit best.

After the MATLAB optimisation, I obtained the following model:

$$t = 0.000283n^2 - 0.005505n + \frac{0.04015}{n} + 0.03681$$

*Where $t$ is the time taken (s), and $n$ is the number of program instances*

The graph below shows the curve of the model, along with my actual data. Time taken (in seconds) is along the y-axis, and number of program instances is along the x-axis. It is worth noting that you can only actually get integer numbers of program instances, I displayed the model as a continuous line to illustrate its overall shape better and differentiate it from the points.




## 3.1 Analysis

First of all, I am surprised that I could find a model that so closely fit the data. I still cannot be sure if the model can be used to accurately extrapolate, or if I have over-fit it to the data, and is only locally correct.

Towards the right half of the graph, a pattern emerges, with the real data beginning to zig-zag about the model line. I think this more complex pattern is linked to unevenness in the square-like topologies achievable with different numbers of nodes, and how well the 2D image can be divided amongst those topologies. I can't link it to any specific numbers however. All I can say is that even numbers of program

instances seem to perform more predictably than odd numbers. Also, there's no direct correlation between the squareness of the topology and efficiency, the actual relationship appears to be more subtle than that.

# 4 Conclusions

First of all, as expected, there were large performance gains achieved by dividing the workload between multiple machines. This is a relationship of diminishing returns however, and at some point, the overheads will begin to outweigh the benefits.

Writing code for a distributed memory system was an interesting contrast from the thread-based paradigms I have worked with in the past. It is clear to see the down-sides compared to shared memory systems, however it is also interesting to consider the increase in scalability, where the communication system becomes the only technical bottleneck to otherwise infinite potential for stacking machines.

This task was also a good C exercise for me. I have not written much C code in the past, and this is the most involved I have been with things like pointer arithmetic. Overall this task has been a good learning experience for me.