# HPC4M Assignment 1

Andrew Cleary
andrew.cleary@ed.ac.uk

January 31, 2021

## 1 Parallelised Matrix Multiplier

### 1.1 High Level Overview of Code

In this exercise, I wrote some parallelised C++ code to multiply 2 square matrices together, $A$ and $B$. I made use of the vector C++ library to store the matrices as 1d vectors. The code takes in a single command line argument, $N$, which is the size of the $N \times N$ matrices that are to be multiplied together. This variable is also used in the definition of the matrices, given in the question.

The code handles the case when $N$ is a multiple of the number of processes used, $Nproc$. When $N = Nproc$, each process generates a single row of the matrix $A$. When $N = k \times Nproc$, the root process generates the first $k$ rows of $A$, while the second process generates the second $k$ rows, and so on. The root process generates the full matrix, $B$, as a 1d array and broadcasts it to all the other processes. Each processes performs the matrix multiplication with their respective row(s), which the root process then gathers into the final answer, $A \cdot B$. The root process then prints this out.

### 1.2 $N = Nproc$

Firstly, the code was written to handle the case when $N = Nproc$. As a test that the code was working, I ran it for $N = Nproc = 3$ on Cirrus. The result printed by the code was equal to

$$\begin{bmatrix} 75 & 68 & 43 \\ 204 & 184 & 116 \\ 387 & 348 & 219 \end{bmatrix}$$

which I checked by hand, and found to be correct. I then ramped up the values of $N$ and $Nproc$ to 36, which is the number of processes available on a single node. The full $36 \times 36$

result was too large to print fully here, but it was equal to

$$\begin{bmatrix} 352944 & 367710 & \dots & 34374 \\ 756432 & \ddots & \ddots & 72672 \\ \vdots & \ddots & \ddots & \vdots \\ 44548704 & 45783360 & \dots & 3709584 \end{bmatrix}$$

### 1.3 $N = k \times Nproc$

I then generalised the code to handle the case when the size of the matrix is larger than the number of processes that we have available to us. However, I still impose the condition that $N = k \times Nproc$, where $k$ is the number of rows each process has to multiply. The generalised code returned the same answer for $N = 36$ as above, when I set $Nproc = 6$, such that each process generated and multiplied 6 rows of $A$ with $B$, which shows that the generalised code still returns the correct results.

## 2 Parallelised Trapezoidal Rule

### 2.1 High Level Overview of Code

In this exercise, I wrote parallelised C++ code to implement the Trapezoidal rule in 2D. The code was based on the following formula:

$$\int_a^b \int_c^d f(x,y)dxdy \approx (b-a)(d-c)\frac{f(a,c)+f(a,d)+f(b,c)+f(b,d)}{4}$$

where I take the average of the 4 corners of the region we are integrating over, and multiply it by the area of the region, which is a natural extension of the 1d Trapezoidal rule. The code was first written in a serial fashion, and then it was generalised so that it could be run in parallel. The region of integration was divided between the processes in two different ways - in strips divided along the x-axis, and in squares of equal area.

The error used in this code is defined as the absolute error:

$$\text{Error} = \frac{|\text{Numerical} - \text{Analytic}|}{|\text{Analytic}|}$$

### 2.2 Error versus N

Firstly, I plotted how the numerical result varies with the resolution, $N$, using the serial case. I found that I had to set the resolution to at least $10^6$ for the numerical answer to be correct to 3 decimal points. Note that the analytic result equals 195.216, while the numerical result with $N = 10^6$ is 195.212. We see that the numerical simulation begins to stabilise nicely at $N = 10^5$.

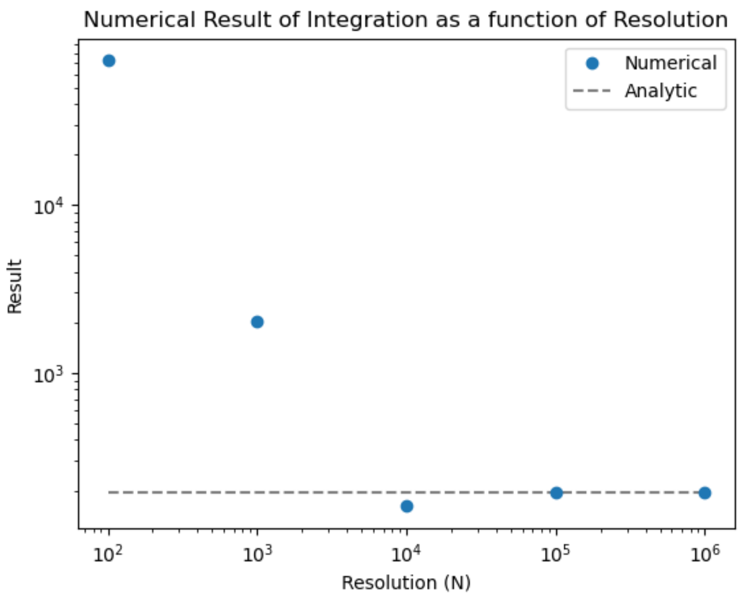

Figure 1: Plot showing how the numerical result compares with the analytic result for different resolutions, $N$.

From Figure 2, we see that the absolute error improves dramatically with the resolution, $N$. Note that the computation time required to go above this resolution was too large for this assignment, but we see that the error improves with an inverse power law as we increase the resolution.

## 2.3 Time improvement with Parallelisation

Note that the result for the integral returned by the parallelised code was exactly the same as the result returned by the serial code, confirming that the parallelisation was done correctly. From the analysis in the previous subsection, I decided that the maximum resolution before rounding errors come in, which can be achieved in reasonable computation time, is $N_{\text{max}} = 10^5$. I then ran parallel simulations with an increasing number of processes. There are constraints on the number of processes due to the division of the region into squares and strips. The number of processes has to equal an square number, which divides into $N_{\text{max}}$. Thus, I ran it for $N_{\text{proc}} = 4, 16, 25, 100$. Here, we compared the running time for the strips and squares methods compared to the serial code. This was plotted as a function of $N_{\text{proc}}$ in Figure 3.

From Figure 3, we see that the slope of the log-log plot is approximately equal to -1, which means we have an inverse relationship between running time and $N_{\text{proc}}$. This makes good sense as if we double the number of processes, we would expect the running time to be halved. We also see that the difference in running times between the strips and squares methods is negligible. To investigate this further, I also plotted the difference between the

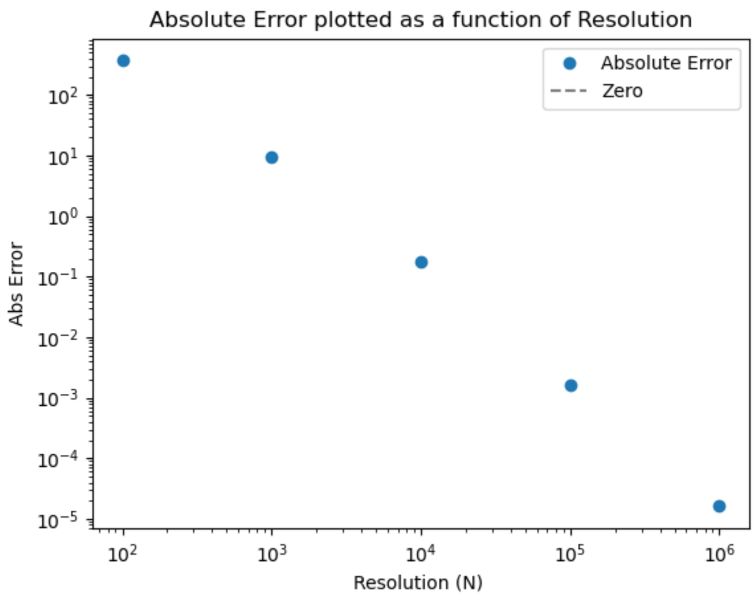

Figure 2: Plot showing how the absolute error improves with the number of processes employed, $N$.

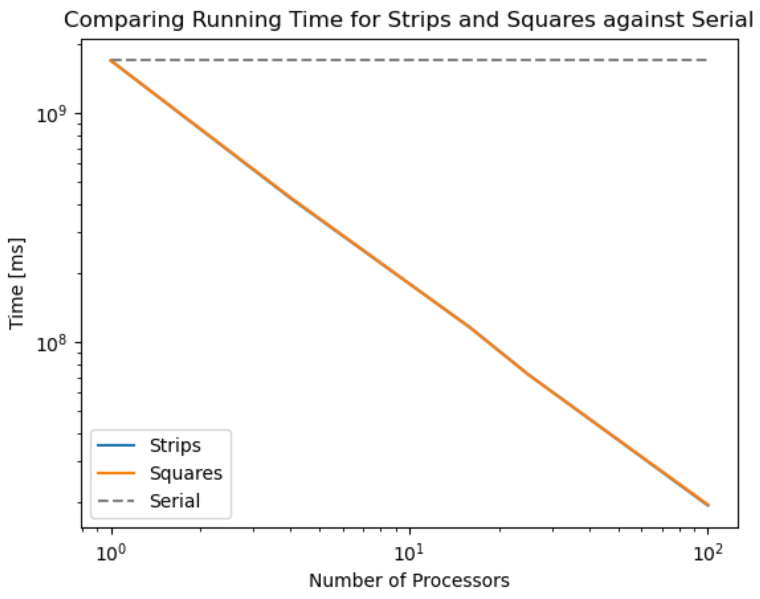

Figure 3: Plot showing how the running time decreases with the number of processes employed, $N$.

running times of the strips and squares methods. In Figure 4, a positive difference implies that the square method is faster at that number of processes. From this plot, we see that on the whole, the methods are very similar, with a strange increase in speed for the squares

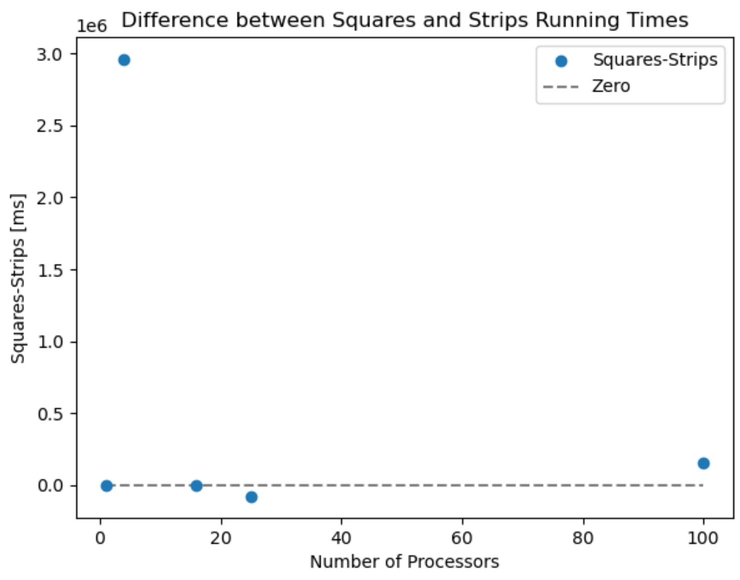

Figure 4: Plot showing how the squares running times minus the strips running time varies with the number of processes employed, $N$.

method for $N_{\text{proc}} = 4$. It is difficult to make any conclusive claims based on this graph though. A safer claim is to say that there is no obvious reason to use either method, except that the strips method is slightly easier to code.