HPC4M Assignment 3

Andrew Cleary
andrew.cleary@ed.ac.uk

February 22, 2021

1 Parallelised 2D Wave Equation Solver

1.1 High Level Overview of Code

In this exercise, I wrote some parallelised C++ code to solve the 2D wave equation:

$$u_{tt} = u_{xx} + u_{yy}$$

for $x, y \in (-1, 1), t \in (0, 1)$. I first wrote serial code first to solve this equation, using a central difference method for space and the leap-frog method for time, subject to some boundary conditions. Once the serial code was working, I parallelised the code using the halo-swapping technique, as there is overlap between the data required by each process.

Note that memory had to be dynamically allocated in the heap for this assignment as the arrays used were very large. It was particularly important that these memory was allocated as a single huge block for the parallel code, as opposed to having the sub-arrays scattered around the heap.

Three different methods of splitting up the grid u were explored:

1. Horizontal Strips
2. Vertical Strips
3. Equally-sized squares

After each time-step, this procedure was carried out for the strips code:

1. Every even-ranked process first sent the overlap at the end of its data, while every odd-ranked process received this data.
2. Then, this was inverted, so that every odd-ranked process sent the overlap at the end of its data, with the exception of the last process. And every even-ranked process received this data, with the exception of the root process.

3. Then, every even-ranked process sent the overlap at the start of its data, with the exception of the root process. And every odd-ranked process received this data.

4. Finally, this was inverted, so every odd-ranked process sent the overlap at the start of its data. And every even-ranked process received this data.

The procedure was similar for the squares code. I created a grid of processes, each with an index in the horizontal and vertical direction, like in Figure 1. Thus, I was able to use my code from the horizontal and vertical strips to do the halo-swapping, along with these process indices.

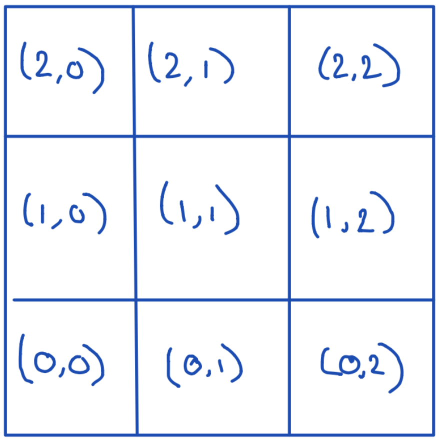

Figure 1: Labelling of processes in the squares parallel code

The reason the code was written in terms of even and odd-ranked processes, was because MPI_Ssend was used, so that every process has to wait until its message is received. We want to avoid a situation where every process is waiting for a single process to receive a message. Writing the code this way means that at worst, each process is just waiting for the neighbouring process to receive its message. This means that we avoid a domino effect kicked off by the receipt of the message at the final process.

The run-time of each of these three methods was examined, by plotting the strong scaling of the code. An estimate for the percentage of the serial part of the code, α was estimated from the strong parallel efficiency of all three methods. Then, the best and worst scaling methods were chosen and their respective weak scaling and weak speed up of these two methods was plotted.

1.2 Comparing Serial to Parallel Results

The serial code returned the following waves, at the times $t = 0, 0.33, 0.67, 1.00$:

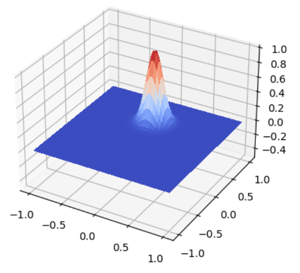
Figure 2: t = 0s

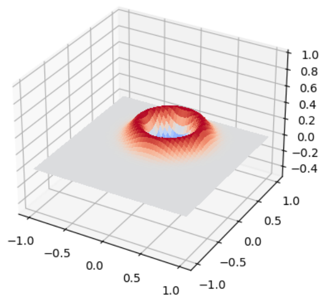
Figure 3: t = 0.33s

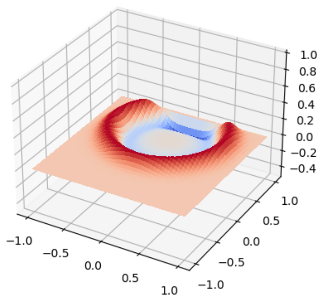
Figure 4: t = 0.67s

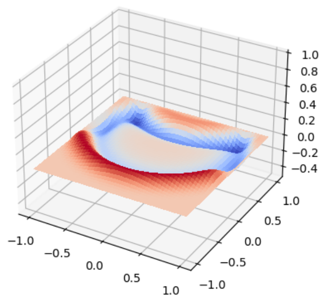
Figure 5: t = 1.00s

When I parallelised the code, I found the exact same solution from all three methods as from the serial code. This confirms that the parallelisation was done correctly.

1.3 Strong Scaling

Strong scaling was tested for all three methods. Note that writing time was omitted for this analysis. A quick discussion on the writing time for each of the three methods is included at the end of this report. A simple strong scaling plot can be seen in Figure 6. We can see that there is a great reduction in run-time for all three methods, but as we increase the number of processes, we also increase the overhead from sending and receiving the data, so eventually this improvement tapers off.

From Figure 6, we can clearly see that the vertical (Y) slices code performs considerably slower than the other two methods. The reason for this is the way that C++ stores data in an array. C++ is a row-major language, which means that the consecutive elements of a row reside next to each other. Thus, when we halo-swap the data after each process, it is much quicker to access and halo-swap elements along a row, instead of along a column. This accounts for the dramatic difference between the horizontal (X) and the vertical (Y) running times. The squares code is much closer to (but still slightly worse than) the horizontal (X) code as it is a mixture of both row and column halo-swapping.

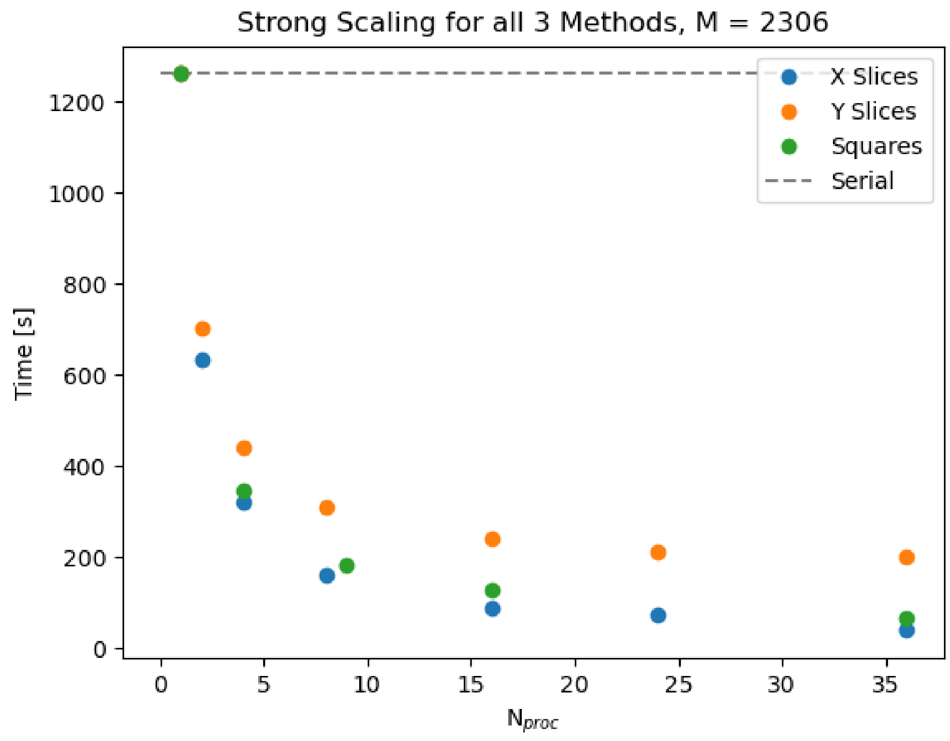

Figure 6: Strong Scaling for all three methods. An MxM grid was used.

1.3.1 Parallel Efficiency

Another nice piece of analysis we can do is compute the parallel efficiency in the strong scaling case. This is defined as

$$E(N,P) = \frac{T(N,1)}{T(N,P) \times P}$$

where N is the size of the problem (kept constant) and P is the number of processes used. A plot of this against the number of processes can be seen in Figure 7.

We can also use Amdahl's law to extract an estimate for the fraction, α, of the code that is completely serial:

$$E(N,P) = \frac{1}{\alpha P + (1-\alpha)}$$

I fitted this curve to the parallel efficiency plot, and an estimate for α for all three methods can be seen in Figure 7. As expected from the discussion on row-major languages, we see that the horizontal code has the best estimate for α, while the vertical code has the worst. In fact, $\alpha \approx 0.01$ means that only 1% of the horizontal code is serial, which is quite a good parallelisation.

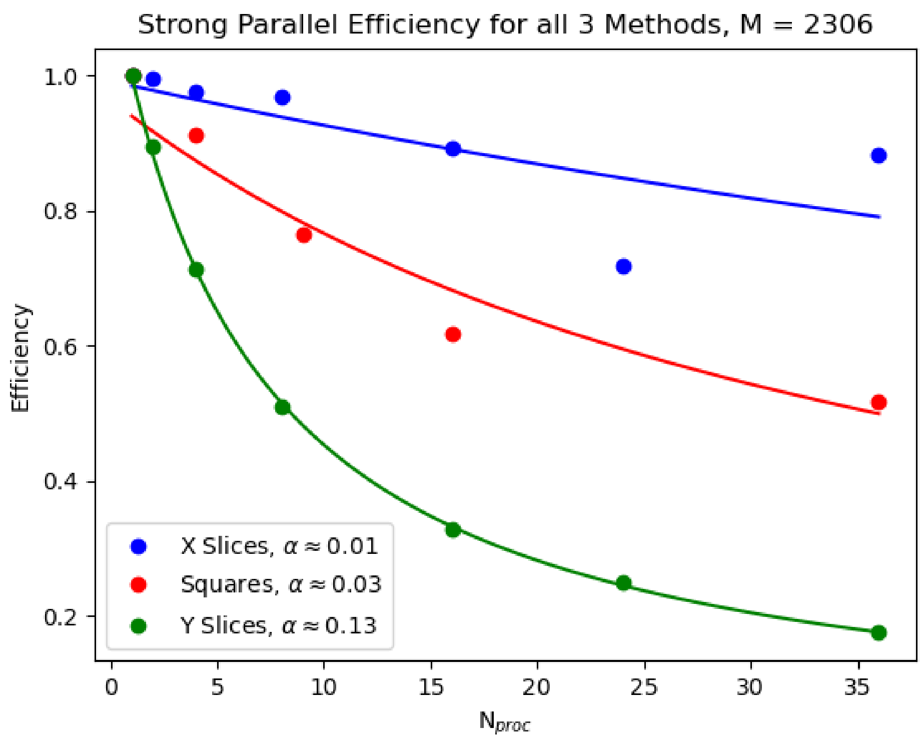

Figure 7: Parallel Efficiency for strong scaling for all three methods. An MxM grid was used.

1.4 Weak Scaling

As per the assignment, further analysis was done for the best method of parallelisation (horizontal slices) and the worst method (vertical slices). Thus, I plotted the weak scaling of both of these methods. This involves keeping the serial part of the code constant, while scaling the parallel part of the code with the number of processes used. In this way, the work done by each process is constant. I tried to do this by making sure that processes always worked on the same size grid, independent of how many processes were used. This meant that I had to increase the size of the overall grid (MxM) with the number of processes used. A plot of the weak scaling can be seen in Figure 8

We see the trend in Figure 8 that we expect. Ideally, this plot is a straight line, but in practice, we introduce communication overheads with more processes. However, we still see that these overheads are introduced much more slowly with the horizontal slices, as expected.

1.4.1 Parallel Speed Up

In order to mix up the plots created, I plotted the parallel speed up for weak scaling for the best and worst method. Parallel speed up is defined as

$$S(J \times P, P) = \frac{T(J,1)}{T(J \times P, P)}$$

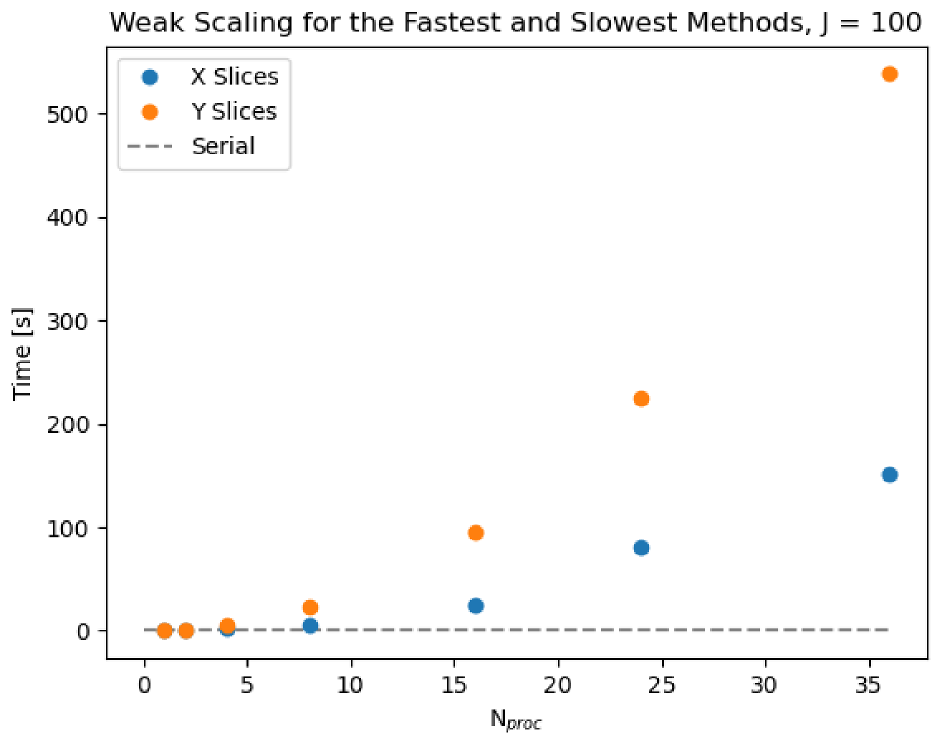

Figure 8: Weak Scaling for the best and worst method. A JxM = 100x2306 grid was used on each process, no matter how many processes were used.

in the parallel case. This can be seen in Figure 9.

We can use Gustafson's law to estimate α again:

$$S(J \times P, P) = P - \alpha(P-1) = P(1-\alpha) + \alpha$$

The result returned from Gustafson's law was not as clean as that returned by Amdahl's law though. I think this is because I could not increase the parallel part of the code without also increasing the serial part.

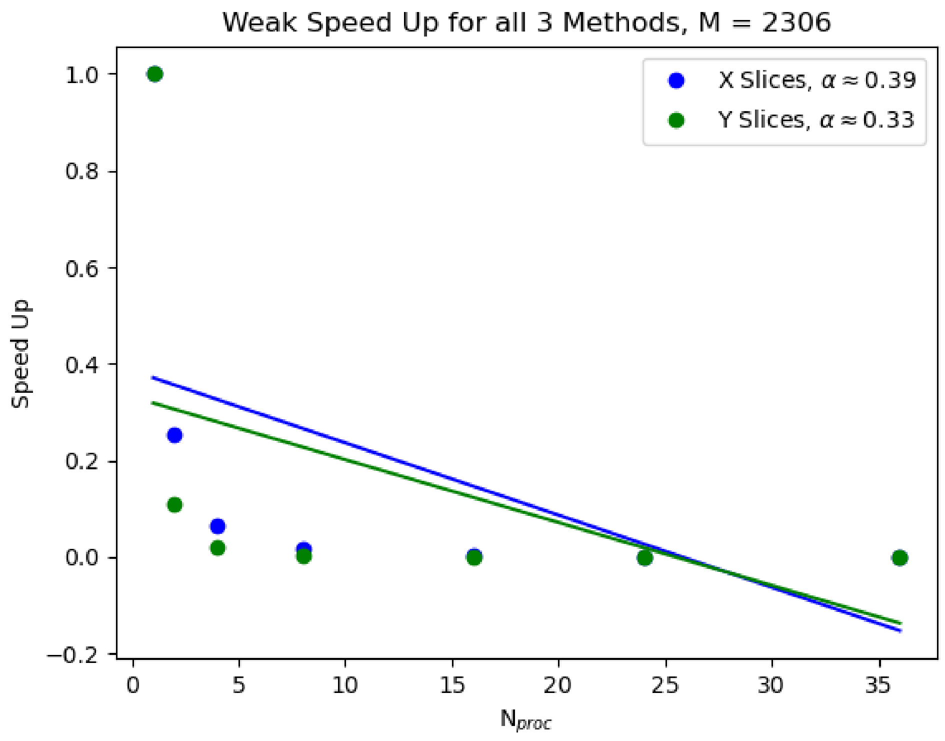

Figure 9: Speed up for weak scaling for all three methods. A JxM = 100x2306 grid was used on each process, no matter how many processes were used.

1.5 Writing Times

Finally, a plot of the writing time as a percentage of the total running time for a 2306x2306 grid is plotted as a function of the number of processes used. The method for writing data used was to gather all the data into one large grid on the root process and then to write this to a file from the root process. This is not the most efficient way to do this, and I would have liked to learn about parallel HDF5, but I might try to learn how to use that for the main project.

We can see that the writing time is quite a considerable portion of the code, and it would be very beneficial to be able to reduce this with parallel HDF5 or something similar. We also see that this percentage increases considerably with the number of processes, which is to be expected as the root process has to gather from more processes.

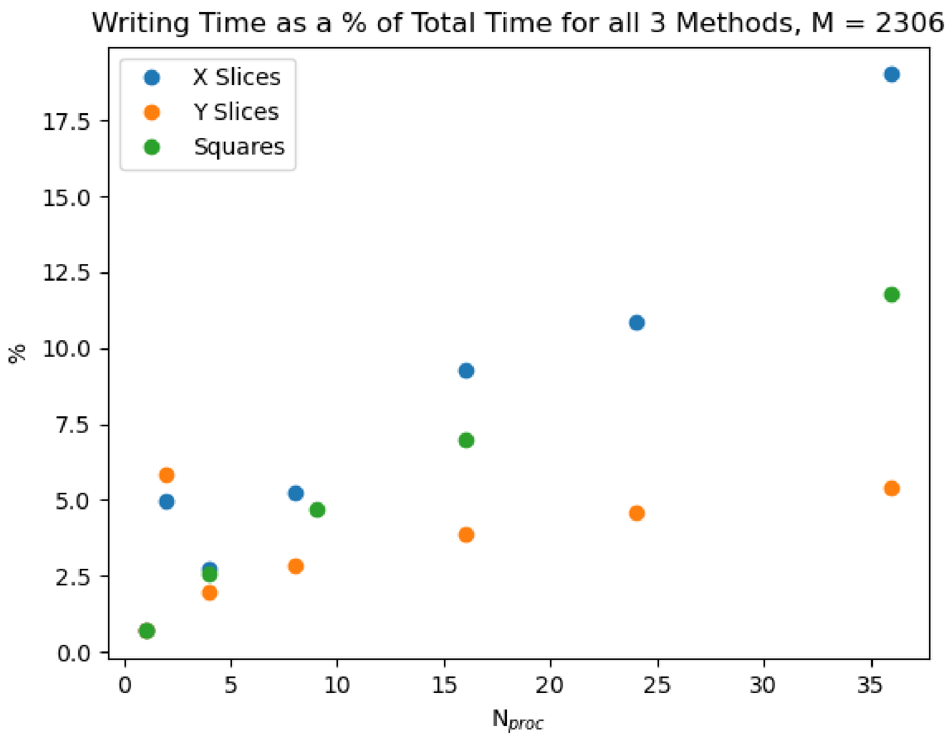

Figure 10: Analysing how writing time increases for all three methods with the number of processes.