HPC Assignment 3

Samuel Richard Bonsor

In this workshop we parallelise a serial code for the numerical solution of the 2D wave equation using a variety of techniques to decompose the domain of solution. Namely, horizontal strips, vertical strips, and equally sized squares. We then look to compare the performance achieved using each of these methods.

Firstly, there are a couple of important points to note about running the code as given. Firstly, the user is expected to provide a compatible combination of the number of processors ($nproc$), and the number of spatial grid points in each dimension (M). The relations that must be satisfied in the case of the strip decomposition (both horizontal and vertical), and the square decomposition are

$$J = 2 + \frac{M-2}{nproc}, \qquad J \in \mathbb{Z}, J > 3,$$
$$\text{and } J = 2 + \frac{M-2}{\sqrt{nproc}}, \qquad J, \sqrt{nproc} \in \mathbb{Z}, J > 3,$$

respectively. In this context J is the number of rows/columns assigned to each processor in the strip decomposition and the length of the square sides in the square decomposition. Also, the user is expected the change the hard coded value of $\sqrt{nproc}$ within the source code para_wave_square.cpp before compilation. This is to avoid any potential errors in using the $\sqrt{nproc}$ directly from the number of processors given due to the fact that the sqrt function in C++ provides a floating point number which may not round correctly when converting to an integer. In terms of a workflow to run all three parallel codes together it is required to follow the following steps:

1. Change the values of M and $root\_proc$ in the relevant .cpp files. Keeping in mind the restrictions mentioned above.

2. Change the number of processors within the .slurm scripts.

3. Ensure that the submit directory in the .slurm script is correct. This is where the output files will be returned.

4. Run the compile_now script.

5. Submit the three runs to cirrus using the three provided .slurm scripts.

6. The .out files will then contain the run times of the three programs and the actual results of the runs will be in a set of .csv files.

We also bring attention to an important aspect of the implementation using the square decomposition. If you were to perform the halo swapping procedure in the standard manner, swapping the $J-2$ middle elements on each side of the square between the adjacent squares to the left/right and top/bottom then we would be left with corner elements that have not been swapped correctly. To address this first the left/right halo swap is performed normally. The following up/down swap is then done using $J-1$ elements (which thus includes a single element that was swapped to the current square by an adjacent square). This has the effect of performing a "diagonal" swap which takes care of the errant corners.

In Fig.1 we present the results at three separate times for each decomposition. We note that all three decompositions provide identical plots. Both to each other and to the result from the serial code given by Rene Lohmann. This is a good check that all is functioning correctly.

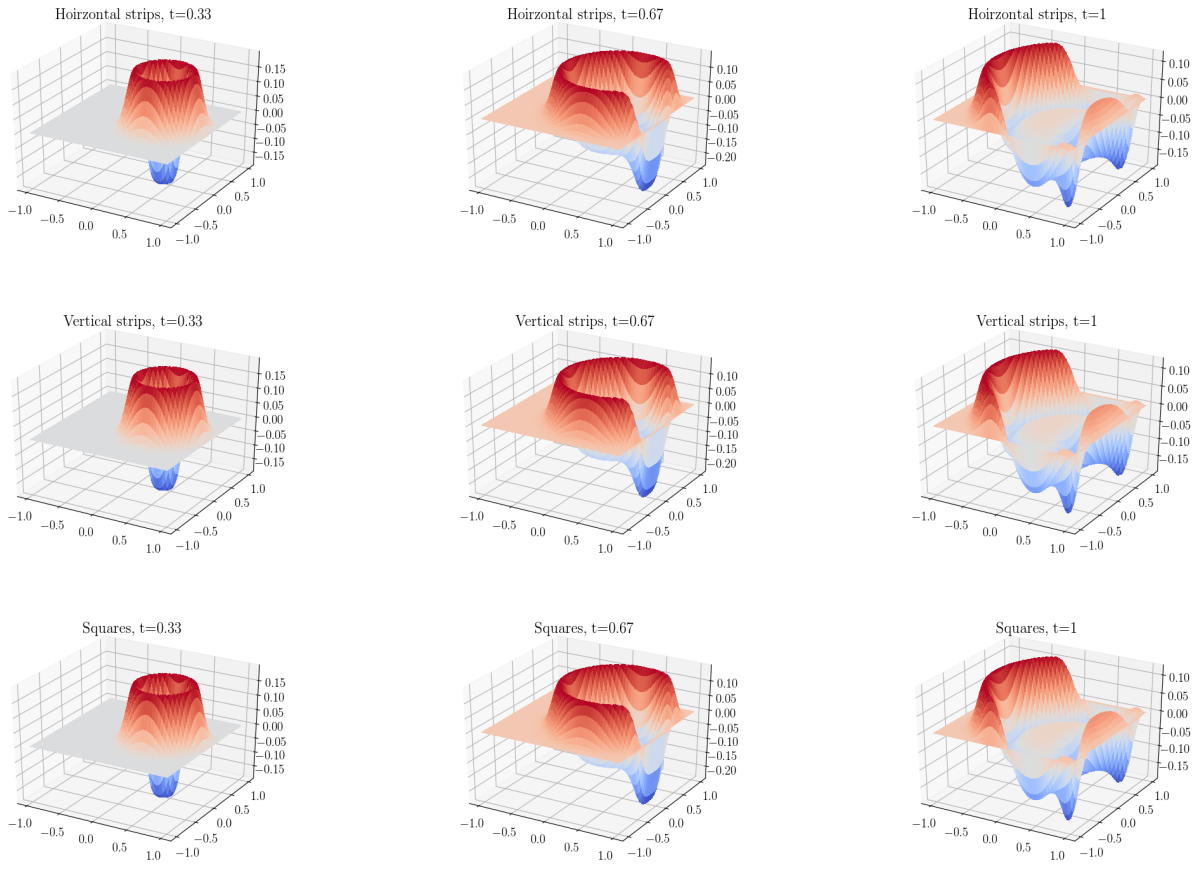

Figure 1: The results of running the three parallel codes. I used $nproc = 4$ and $M = 2306$ for these runs.

The different decompositions do however have varying run times, as seen in Fig.2. It is a little tricky to see from the low processor number runs in this figure but in all cases the horizontal strip decomposition is the fastest, followed by the square decomposition, and finally the vertical strip decomposition. This is to be expected ad C++ works in row major order so the various loops over the long rows in the horizontal strip decomposition are more efficient. The same could be said of the square decomposition, however in this case there is much more communication between process as each processor needs to swap data with as many as 4 other processes so the communication overhead is higher.

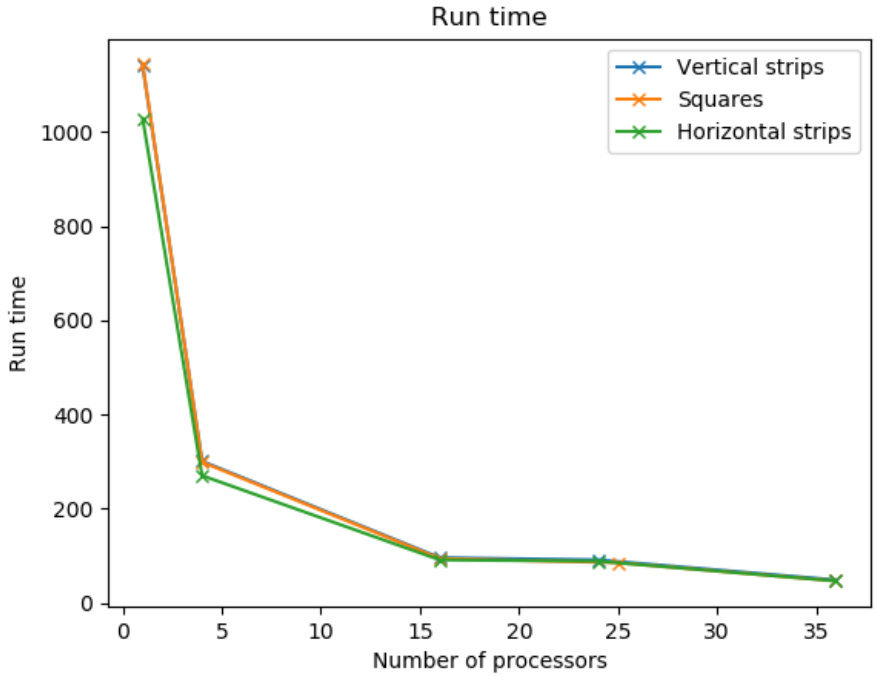

Figure 2: A plot of the program run time against the number of processors, including data I/O.

In Fig.3 and Fig.4 we present plots of the two metrics introduced in the lectures. The parallel speed-up and the parallel efficiency. It is worth noting beforehand that I'm not sure why the results for $nproc = 24, 25$ are so much lower than the surrounding points would suggest. Regardless the general trend is clear. For the range of processor numbers examined here we always achieve a decreased run time, with the speed up increasing as more processors are added. However, we can alos clearly see that the parallel efficiency is steadily decreasing. This means that, per processor, we are getting less of a speed up each time we add more. This is to be expected here as the problem size is being kept the same (weak scaling). Thus, as more processes are added, more and more of the computational burden lies in communication. Clearly this is not something that adding more processors helps with, thus the decreasing efficiency. We would expect the efficiency to further decrease as more processors are added and eventually for the speed-up to plateau as the task is then completely communication limited.

Finally, we repeated each of our runs without printing out any of the data. In Fig.5 we show the fraction of time taken for the I/O process and see that as more processors are added and increasing fraction of time must be spent on outputting the data. This is because each subprocess has to communicate back to the root process the build up the final matrix to output. Interestingly, although the horizontal strip decomposition is the fastest to run overall we see that it is also the one that has the largest fraction of time taken up with outputting the data.

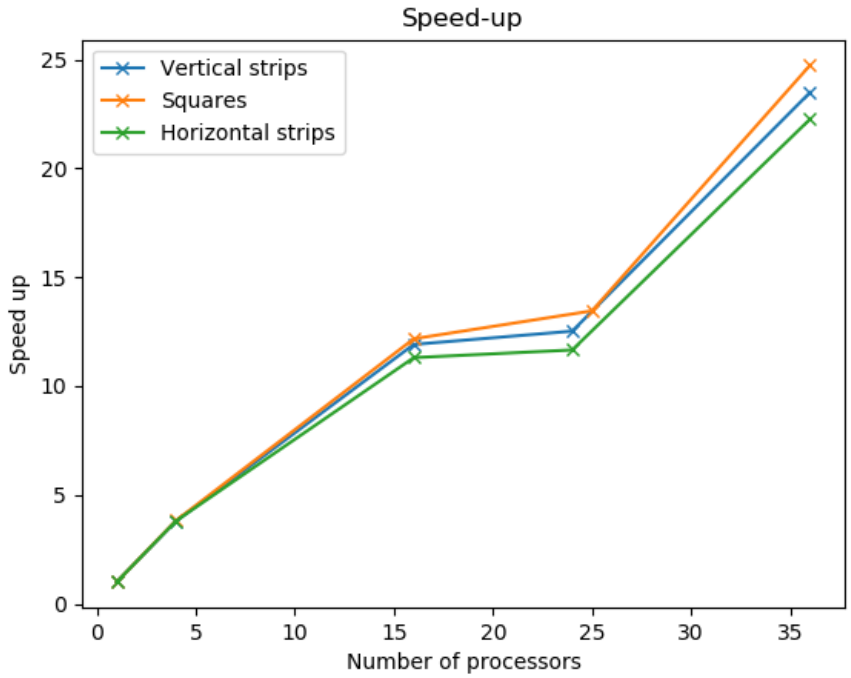

Figure 3: A plot of the parallel speed-up achieved against the number of processors.

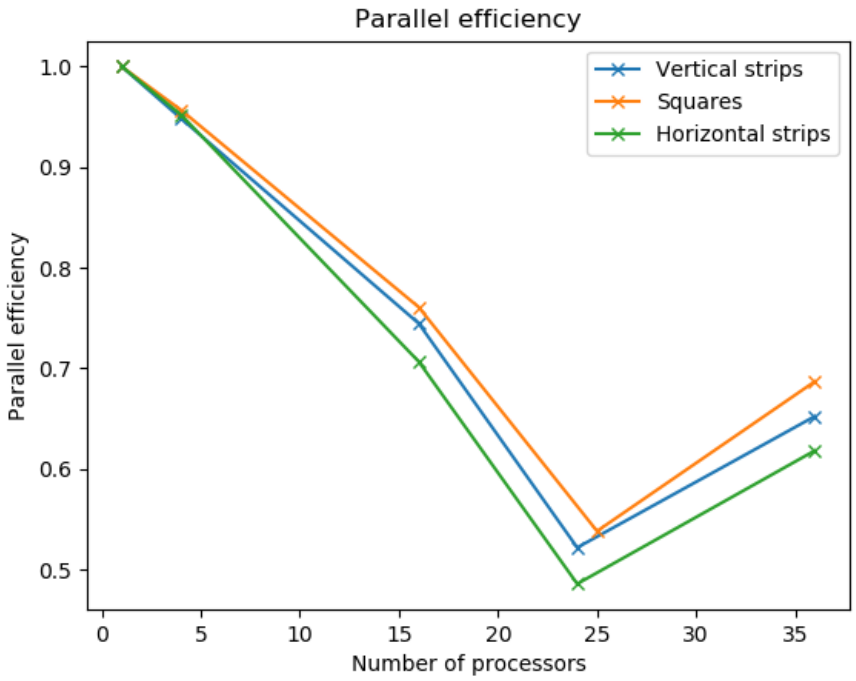

Figure 4: A plot of the parallel efficiency achieved against the number of processors.

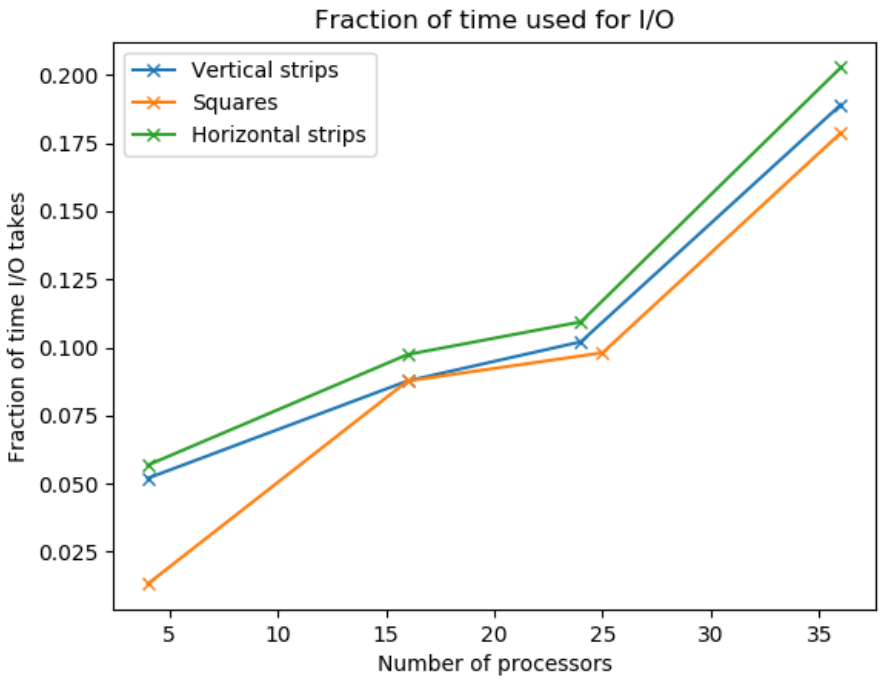

Figure 5: A plot of the time taken for data input and output against the number of processors.