HPC4M - ASSIGNMENT 3

KAROLINA BENKOVA

1. Problem introduction

In this assignment we are going to be working with the 2D equation

$$u_{tt} = u_{xx} + u_{yy} \quad \text{for} x, y \in (-1, 1), t \in (0, T),$$

subject to the initial and Dirichlet boundary conditions

$$\begin{aligned} u(x, y, t = 0) &= \mathrm{e}^{-40((x-0.4)^2+y^2)}, \\ u_t(x, y, t = 0) &= 0, \\ u(x = \pm 1, y, t) &= u(x, y = \pm 1, t) = 0. \end{aligned}$$

Our task is to solve the equation above, parallelising code for serial numerical solution of the equation with equal number of grid points in each dimension, using a formula emerging from central differencing scheme for space and leap-frog for time.

The parallelisation is going to be done by dividing the space (domain) into several segments assigned to several processes. Just like when solving the 1D heat equation, the neighbouring segments are going to overlap by two rows or two columns as the scheme only works for inner points of the boundary. The boundaries of the segments are therefore going to be acquired by receiving the row/column on the boundary from an inner row/column of the next segment (halo swapping) at each time step.

2. Domain decomposition

When implementing the code, the rows of an array are the x-axis and the columns are the y-axis. In total we have $M + 1 = 2306$ gridpoints in both x and y dimension.

2.1. Horizontal strips.

First we divide our domain into horizontal strips, i.e. each segment is going to include all y and an equal portion (J) of gridpoints in the x-dimension. The amount of gridpoints J is going to be calculated by the formula

$$M + 1 = \mathit{nproc} \cdot (J - 2) + 2,$$

where M is the amount of intervals in the main domain, *nproc* is the number of processes we employ (i.e. the number of horizontal strips), and J the number of rows belonging to each strip. For halo swapping we use the MPI functions for synchronous sending and receiving. This case was handled the easiest of all as we were exchanging rows of data and C++ is row-major, i.e. the data could be sent and received without an additional hassle.

2.2. **Vertical strips.** Now we divide the space into vertical strips (or columns), i.e. each segment will include a chosen number J of columns from the original domain and number of processes so that the same formula as in the horizontal case is satisfied. However, in this case we need to take extra care when doing halo swapping as we are exchanging columns of data. To overcome this, we can instead take the transpose of the strips and use the same procedure as in the horizontal case. The received transpose is then transposed again to get a column shape and used for calculating the value at the next time step. The disadvantage of this domain decomposition in combination with C++ is that we need more operations and memory (or in a different approach to the coding side of the problem, more process to process communication) to achieve the same result as in slicing the domain horizontally. In a language like Fortran which is column-major we would encounter the opposite problem with vertical slicing being the easier option. To save memory (and potentially computation time) I could have opted for sending the elements one by one in a for loop as is the case below - it would be interesting to compare the metrics with each option.

2.3. **Equally sized squares.** Combining the previous two domain decompositions we can divide the domain into squares of equal size, i.e. each process will have assigned the same number of grid points. We pick the number of processes and J using the formula

$$M + 1 = \sqrt{nproc} \cdot (J - 2) + 2,$$

as the square root of the number of processes gives us how many rows and columns will the domain be divided into. We assign the processes to the squares in the domain starting from the upper left corner being process number 0, and ending in the bottom right corner with process number $nproc - 1$. For halo swapping we need to include both up-down and left-right swapping. First, we swap up-down, exchanging the first $J - 1$ elements of the overlapping rows. After this is finished, we swap left-right, exchanging all J elements in the overlapping columns. When specifying the rank of the process we send to or receive from, we use the rank of the process in left-right swapping and the row and column of the process in up-right swapping using the formula

$$\text{row} = \text{floor}\left(\frac{\text{rank}}{\sqrt{nproc}}\right),$$
$$\text{column} = \text{rank}\%\sqrt{nproc},$$

then e.g. the rank of the process in the above row is $\sqrt{nproc}(\text{row} - 1) + \text{column}$. In halo swapping, sending rows in the up-down direction was of no problem just like in the horizontal slicing case. However, for the left-right swaps we needed to use a different approach. Unlike in the vertical slicing, we chose to send the elements one by one in a for loop, using the corresponding row in the original domain as a tag to avoid both mixing the elements up (as we were sending and receiving multiple items from the same rank) and ending up in a deadlock (this might have not been necessary since we used `MPI_Ssend` as it's dealing with data synchronously but I haven't tried it without using the tag). A simple sketch of the halo swapping technique can be found in Fig 1.

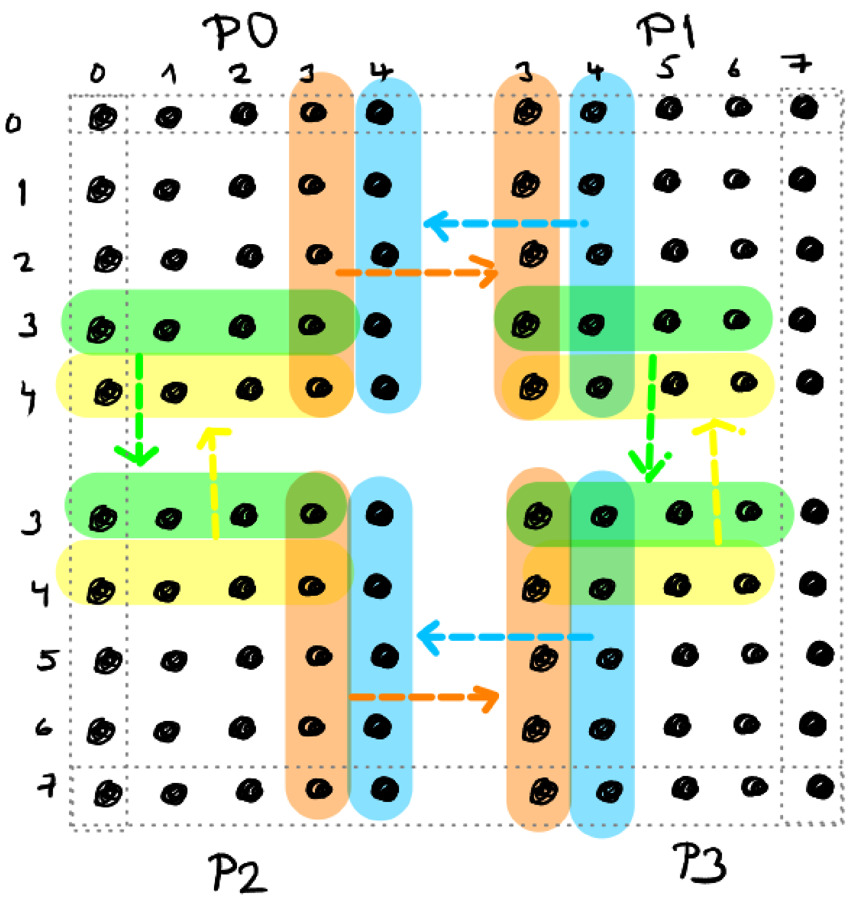


FIGURE 1. Halo swapping with domain decomposed into squares for $M = 7$, $J = 5$ and 4 processes. First up-down swaps happen, followed by left-right swaps.

3. Validation of results

Besides validating the results visually in a 3D plot for chosen time stamps, we can also use Vim to compare the .csv files to get a more precise answer to whether our code works properly or if there's a bug. This can be done by running the command

```
cmp -s file1.csv file2.csv || echo "files are different"
```

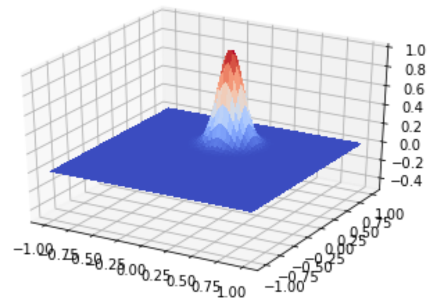
in the terminal which returns the phrase "files are different" if that's the case. For debugging we can also use

```
vimdiff file1.csv file2.csv
```

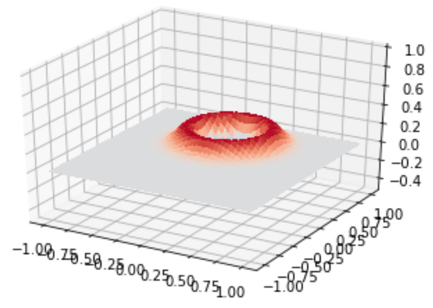
and Vim shows us where the differences in the two files are located. The results of all three scripts were compared with the results of the serial code (provided by Rene) using both techniques. The plots at selected times can be seen in Fig 2.

4. Performance metrics

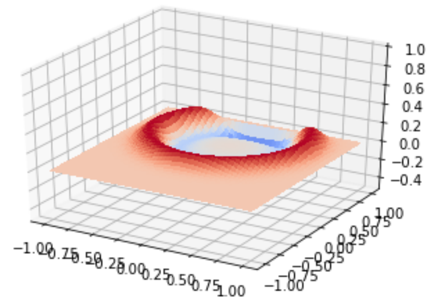
In order to evaluate the performance of the parallel code, we can take a look at its performance metrics, namely parallel speed-up and parallel efficiency, which can be plotted to investigate the



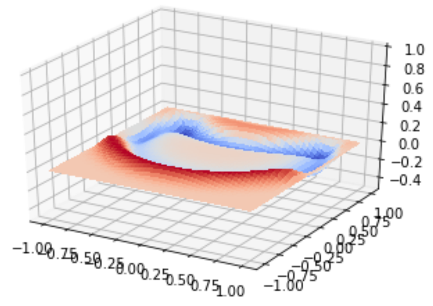
(A) t=0



(B) t=0.33



(C) t=0.66



(D) t=

Figure 2. Plots of the solution of the 2D wave equation at selected times.

strong scaling (i.e. how the performance of the code changes when we increase the number of processes).

We take $M = 2305$, vary the number of processes P and measure the computation time $T(M, P)$ (without writing into the output files), running the code on Cirrus. A simple plot of the number of processes and the computation times can be found in Fig 3 we can see that the computation times are decreasing up to a certain point where it reaches a 'minimum' and an increased number of processes doesn't have an effect on the time.

Next we can compare the parallel speed-up $S(N, P)$,

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

where N is the size of the problem (M in our case), P is the number of processes and T is code runtime with N and P, $T(N, 1)$ can be the runtime of serial code. Ideally we would like the performance to increase linearly with the number of processes ($S = P$), however, due to serial part of the code this is often not possible (so we get $S < P$). Strong scaling gives us an information about the performance with N fixed and P changing, comparing parallel speed-up with P. In Fig 4 we see that the speed-up slows down for the horizontal and square decompositions. The vertical decomposition, however, shows a larger slope and has the potential to overcome the speed-up of the horizontal decomposition. We found that the time to run the serial code is $T(N, 1) = 1264.17$s.

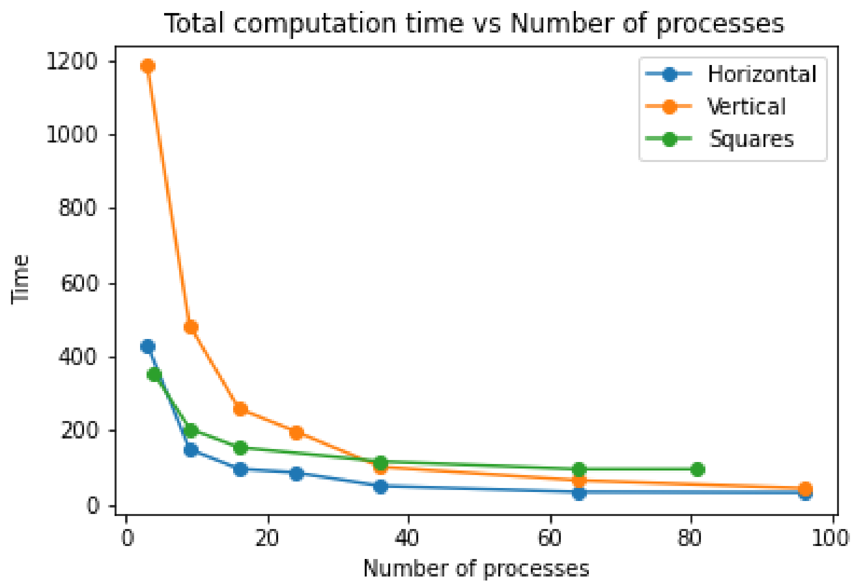


FIGURE 3. Total computation time vs number of processes for all three kinds of domain decomposition.

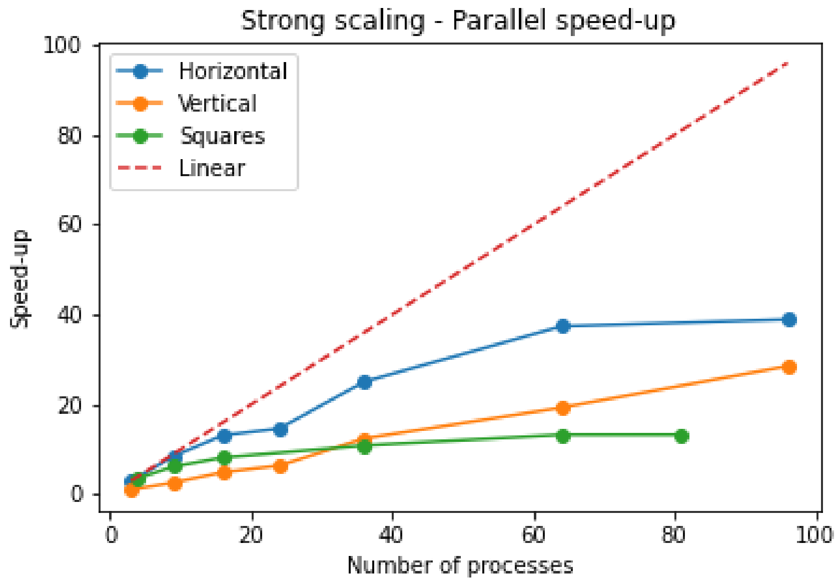


FIGURE 4. Speed-up vs number of processes for all three kinds of domain decomposition.

Another metric we can plot is the parallel efficiency

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{PT(N,P))}.$$

An efficient setup for running parallel code should be close to $E = 1$ (i.e. when the serial process is sped up P times using P processes), whereas a bad one would be approaching zero. In the parallel efficiency plot in Fig 5 we see that the horizontal and square decompositions become less efficient with increasing number of processes, the vertical slicing case stays on the same level.

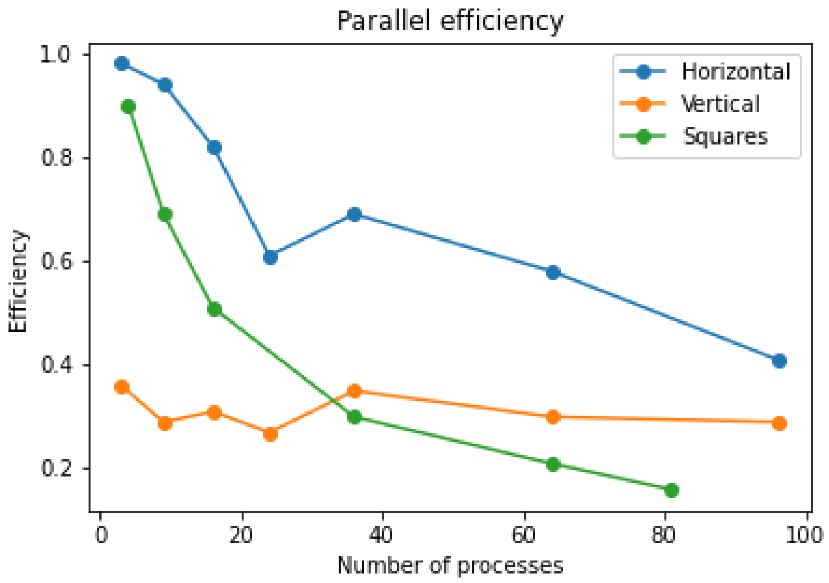


Figure 5. Efficiency vs number of processes for all three kinds of domain decomposition.

The best results with respect to parallel speed up and efficiency were achieved in the case of horizontal domain decomposition. In terms of running time, vertical decomposition was the worst for up to 40 processes, however, horizontal decomposition turned out to be the fastest for any amount of processes we experimented with.