Parallel implementation of a 2D wave equation finite difference solver

Jens Bauer (jtb20)

1 Code structure

The repository consists of a solver written in C++ and a post-processing code written in Python. The user can specify input such as domain size, duration and much more in a input text file which makes the code very user friendly.

The code uses the first process to read in the user input from the text file and uses collective communications to distribute the information to all processes. The domain is decomposed into rectangular strips, such that every CPU core only works on a sub-section of the global domain. The top and bottom row of every strip needs to be send to the processor above and below respectively. This is done using Non-Blocking communication, while to communications are happening, calculations in the center of the domain are being executed which leads to a significant speed-up. Additional performance gains have been achieved by replacing 2-D vectors by 1-D arrays which uses row-major ordering to represent the 2-D array. After a specified time-step, every process writes its domain to a text file. The files are named according to their process id and time-step and are used for post-processing.

The post processing code reads in all the output files generated and creates a gif of the simulation. It only requires the number of MPI processes used as a command line argument.

Figure 1 shows how the code decomposes the domain into rectangular stripes and how the information is shared between the processes for a single time-step.

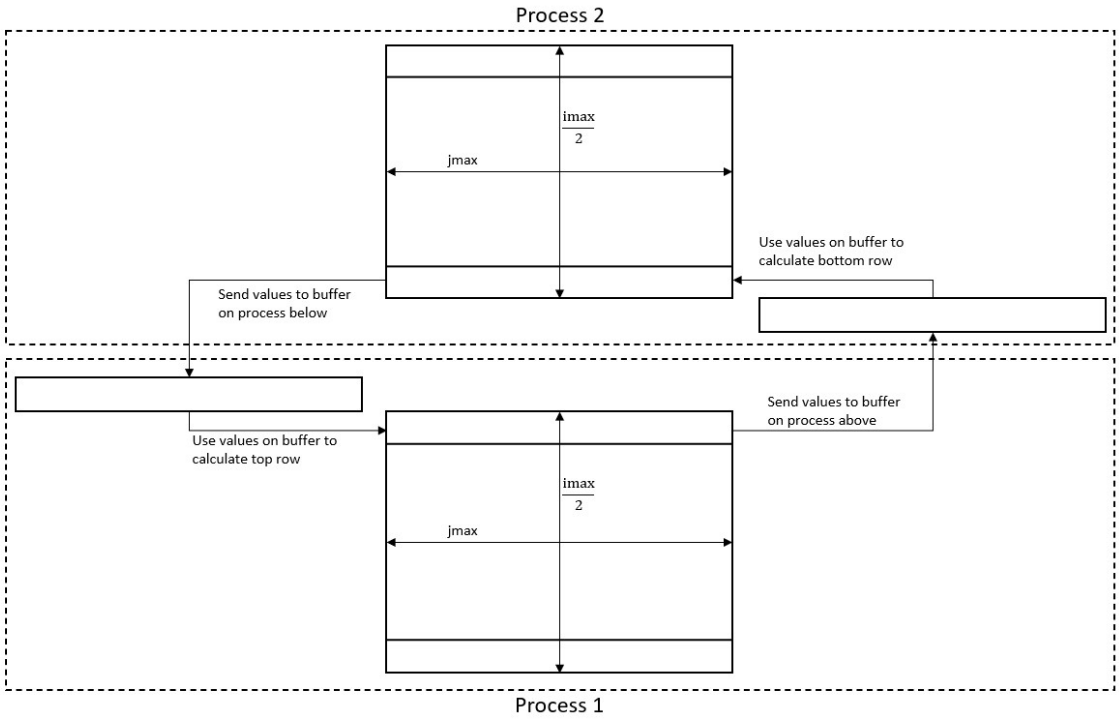

Figure 1: Visualization of the domain decomposition using two MPI processes

2 Results

The project was able to reproduce the same outputs as the serial version but it was faster and could run on multiple cores with the use of MPI. Two more boundary conditions were added as well as a variable Neumann barrier. The added post-processing code can produce gif videos of the simulation output. Figure 2 below shows a sample image of a high resolution simulation with a Neumann barrier.

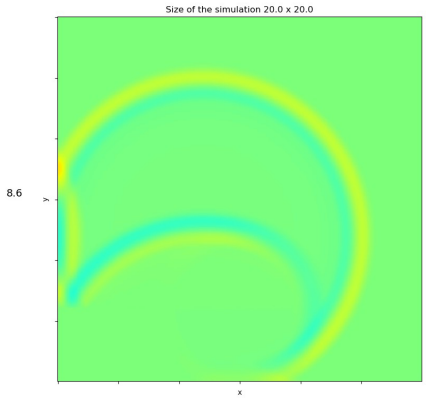

Figure 2: Picture of post-processing code from a 400 x 400 run on 8 cores

3 Code Analysis

To analyse the code quality of the MPI implantation, the code was tested on a HPC system and the run time was recorded for multiple different domain sizes and number of MPI processes used. It is to be expected that the more cores used, the faster the code will be executed which can be measured using the speed-up ratio. The speed-up ratio is defined as:

$$S_N = \frac{T_1}{T_N} \tag{1}$$

The parallel efficiency shows how fast the code is relative to an idea speed-up [2] and it is defined as:

$$E_N = \frac{T_1}{N * T_N} \tag{2}$$

The results for different size square grids are presented in Figure 3. The graph on the left-hand side shows the Speed-up ratio and the graph on the right-hand side shows the parallel efficiency. The timings were done on 1,2,4,8,16,32,62 cores.

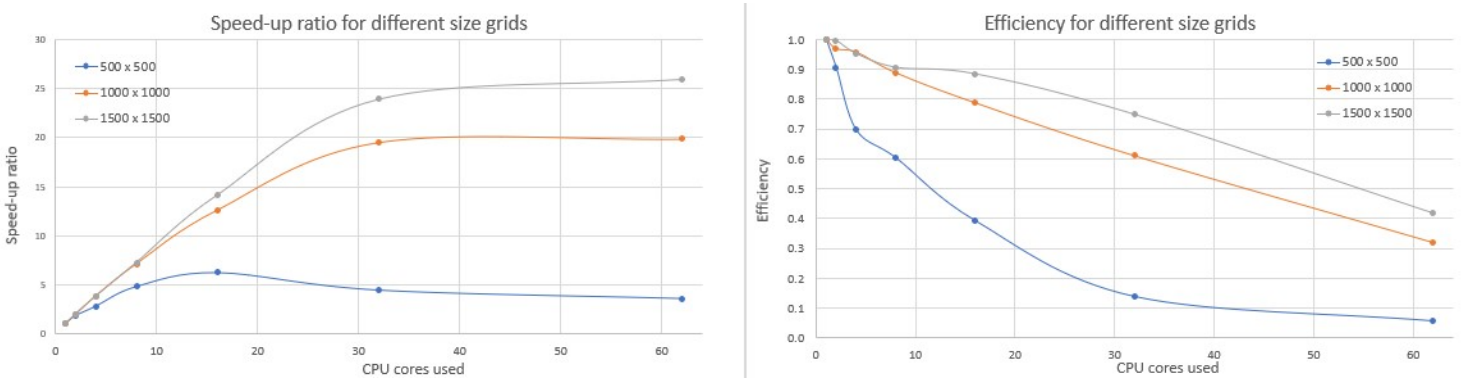

Figure 3: Speed-up and Efficiency results

As seen in the figure above, the speed-up ratio increases almost linearly for the first few additional cores but it flattens or even decreases depending on the size of the domain. This parallel slowdown is caused by the communications bottlenecking the code since the overhead from the communications will surpass the additional CPU power added [1]. The number of cores at which this occurs will depend on the size of the domain.

As the speed-up ratio increases, the efficiency of the parallel code decrease. For a small domain the parallel efficiency drop very quickly below 50% but the largest domain

it remains significantly longer a higher efficiencies. This shows again that as the grid size increases, the cost of the communications compared to the finite-difference calculation is relatively low but if the domain is small, the communications take up relatively more time which decreases the efficiency.

This is analogous to Amdahl's Law, the proportion of the execution time which is parallelisable decreases the smaller the array is. Timing the parallel and non-parallel parts of the code has shown that for an array of size $100x100$ the theoretical maximum speed up is only 3.7 due to the serial part taking up about 26% of the overall run-time. This value decreases exponentially as the array size increases which leads to significantly higher maximum speed-up values.

Figure 4 shows the speed-up ratio for grids with the same number of grid points but for domains with different aspect ratios. The aspect ratio is the number of rows (m) divided by the number of columns (n) of the domain:

$$A = \frac{m}{n} \tag{3}$$

$A > 1$ means that the domain is tall and thin and $A < 1$ means that the domain has fewer rows than columns (fat and wide).

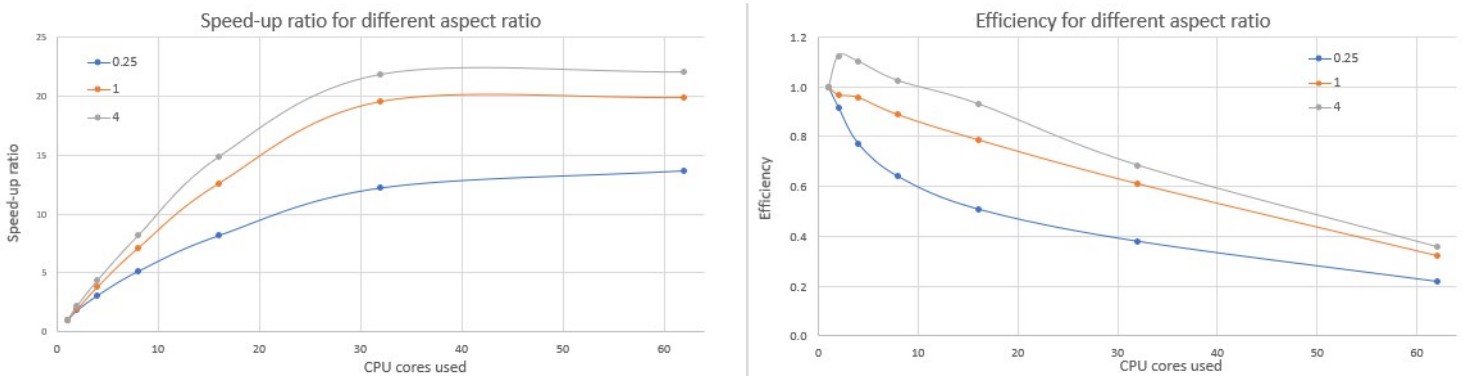

Figure 4: Speed-up and Efficiency results

This analyses has shown that the speed of the code depends on the aspect ratio of the domain. As domain gets taller and thinner ($A > 1$), the amount of data that needs to be

send between the processes decreases which makes it more efficient. A fat and wide domain means that more data, relative to the size of the domain, needs to be send which creates bottlenecks.

Another thing to note in the figure above is the super-linear speedup for $A = 4$ between 2 and 8 processes. For persistent algorithms such as this, the reason for super-linear behaviour is that more cache is available which can reduce the number of clock cycles that are being used to fetch data from RAM. [3]

4 Improvements

The code works well and it relatively efficient in the way the communications and calculations are set up. However, instead of dividing the domain into rectangular strips a more efficient solution would be to divide the domain into squares. The squares then communicate with their neighbours to the left/right as well as up/down. This would make the speed-up ratio less susceptible to the aspect ratio of the grid.

5 Conclusion

This project has successfully implement a parallel version of a finite difference solver using MPI. Through the use of a input text file the user has many input options and a choice of four boundary conditions. The code scales well on large HPC systems but it struggles with domains with a small aspect ratio. The post-processing can generate gifs which is great for visualizing the result.

References

[1] B. Berg, R. Vesilo, and M. Harchol-Balter. hesrpt: Parallel scheduling to minimize mean slowdown. *Performance Evaluation*, 144:102147, 2020.

[2] S. Neethling. *ACSE-6 Patterns for parallel programming.* 2021.

[3] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in hpc systems: Why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2016.