Performance analysis

The performance of the code is studied by measuring its timings on Imperial's High-Performance Computing Cluster (CX1). The program is executed for a domain of 1000x1000, and the rules of the game are applied over 1000 iterations. These dimensions have shown to work well in practice, matching the expected results. Using larger domains has been avoided as it results in unnecessary consumption of power.

The code has been run on a single machine, using 1, 2, 4, 8, 16 and 32 cores in parallel. This range of parallel processes is enough for showing the expected trend in speed-up and parallel efficiency. Furthermore, this ensures that the parallel subdomains used are evenly split, which removes the impact that the grid distribution might play on performance. Further work involves using uneven subdomains and a number of processes that is not a power of 2.

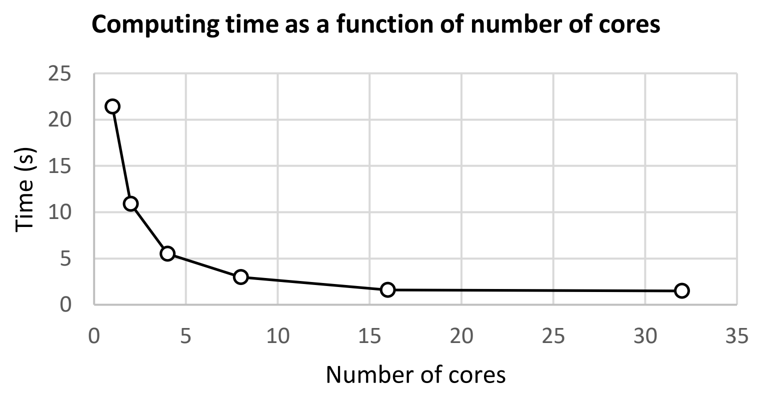
In terms of load balancing, oddly split arrays could negatively impact the code's performance. To study so, reduction operations are used to find the maximum and minimum times taken by the processes. In the context of HPC, however, the time taken by each processor is close to equal as a result of having the domain evenly split.

While using the HPC results in impressive speed-ups, there are some limitations to be considered. Firstly, the performance of two different machines is not assured to be the same – some machines are quicker than others, which might lead to discrepancies in their efficiency. Also, the increase in communication time between cores must be considered for a large number of processes.

To further understand the code's performance, three different measurements are taken as a function of the number of cores using the HPC:

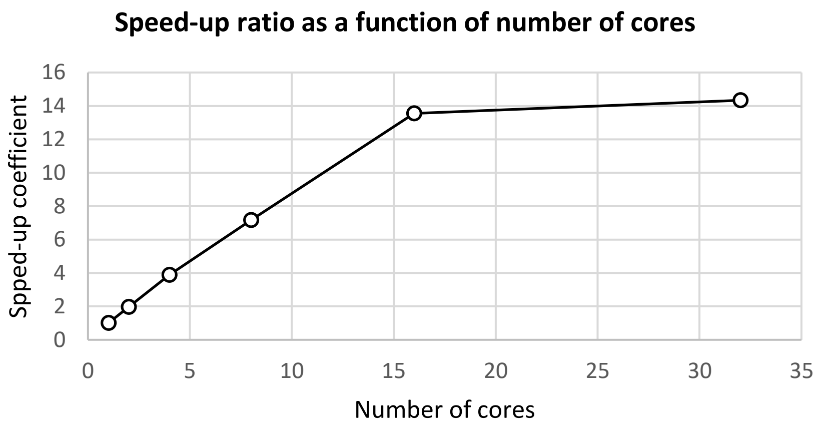
- Time
- Speed-up ratio
- Parallel efficiency

Furthermore, the changes in time as a function of the domain size have been studied for a different number of iterations.

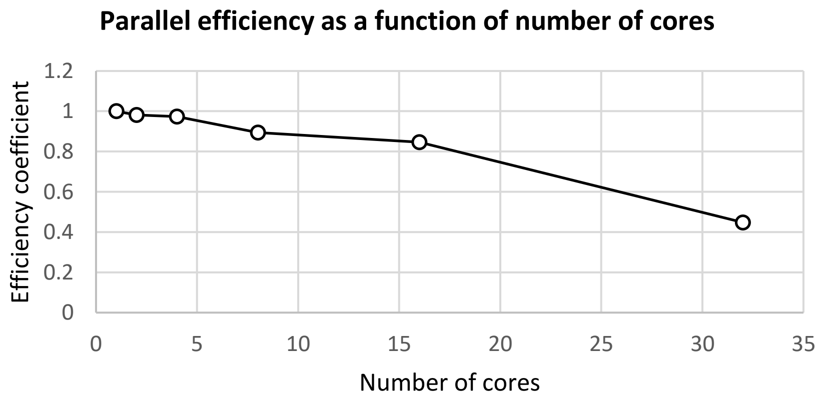


The changes in computing time as a function of the number of processes show an exponential decay. This might be because peer to peer communications scale better on large machines.

The difference in computing time between 16 and 32 cores is minimal. This suggests that for the current set-up, using no more than 16 cores would pose a sensible compromise between core usage and speed of convergence.



The speed-up ratio describes how many times quicker the code is in parallel relative to the serial code. The results show a linear increase in speed-up for up until 16 cores. After that, the increases in speed-up become nearly flat. This is consistent with the insight from the previous plot.

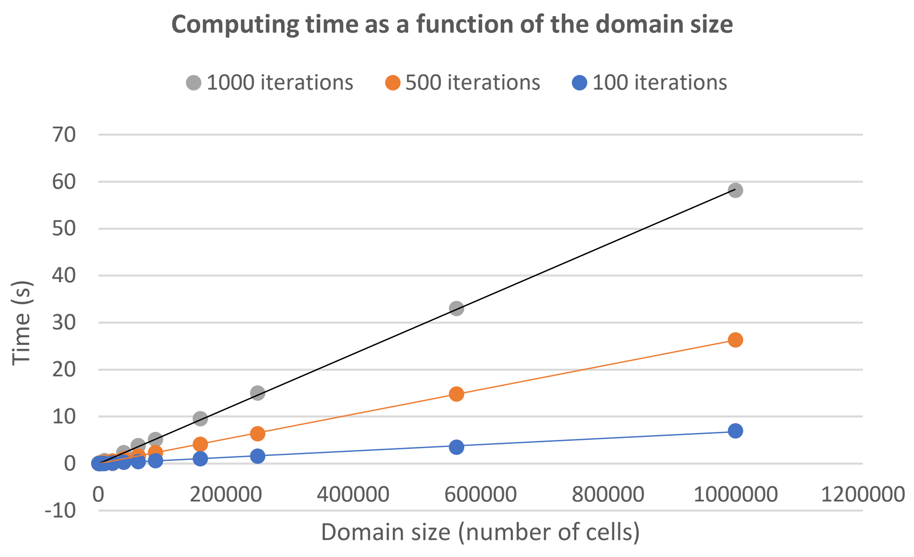


The parallel efficiency metric describes how fast is the code relative to an ideal speed-up. As the number of cores used increases its parallel efficiency drops.

As in the two graphs above, there is a sharp drop after 16 processors. Ideally, changes in the number of cores would result in minimal decreases in parallel efficiency. This is true for cores 1 to 16, where the efficiency coefficient does not fall below 85%.

One might be tempted to use Amdahl's Law to estimate an upper bound for the code's parallel efficiency. However, the proportion that the serial code represents with respect to the parallel code is negligible – especially for large domains such as the ones being tested.

Lastly, the computing time is studied as a function of the domain size for the serial version of the program. The plot below shows that time increases linearly as a function of the domain size, and that time increases exponentially as a function of changes in the number of iterations. This consolidates the thought that the serial version performs poorly over large grids and a high number of iterations.



Furthermore, as mentioned above the HPC plots suggest that the parallel version of the code performs much faster as the number of cores is increased. This indicates that parallelising the code results in significant performance gains, especially for large grids. Increasing the number of cores (processes) used to run the program above a certain threshold (in this case, 16) results in decreasing marginal improvements in computing time. Using a number of cores about such threshold will result in unnecessary overhead and power consumption in exchange for a small increase in execution speed.