ACSE-6-MPI-Report

Son Hoang Thanh

1 Introduction

This report contains a short description of the MPI implementation of the wave equation PDE solver on the 2D computational grid and attempts to analyse its performance as the problem is scaled and solved with larger amount of resources. It gives a brief explanation of basic design choices made for this project and its implementation and finally showcases the test results of its performance on HPC system.

2 Design Approach

The aim of this project was first and foremost the development of efficient and fast program for domain decomposition of the initially provided serial implementation of wave equation program, which could potentially be used for solving larger scale problem of this class using a distributed memory system approach. Therefore, extensive care was taken in ensuring localisation and vectorization of data wherever possible for maximum possible optimization. Additionally, pre-computation of the most used values, and their use within simple and optimized ‘for‘ loops and functions was desired.

3 Implementation and Optimization

The communications between processes were carried out using the non-blocking send & receives. To allow for flexible sending and receiving between the neighbouring cells, each process needed to construct 8 different MPI Datatypes for corresponding 4 edge-boundaries and 4 ghost-boundaries for each of the process grids. This approach was taken such so that each process was able to be incorporated into the for loop, with the iterator being an array of datatype to send and receive. Additionally the send and receive tags were manually set to ensure correct receive order, like in case of receiving with periodic Boundaries or when the requested number of processes is small. The data buffers used were the grids themselves which allows for not only direct sending and receiving to grids without use of intermediary buffers (see Figure 1), but also the correct placement of data from boundaries to ghost cells.

The workload of each process was balanced such that the resources are properly distributed and process communication can be achieved in similar time frame, with minimal synchronisation needed. Each process calculates all the variables it requires locally, such as an array of neighbours which encodes the rank and the position of the neighbours, the most optimal domain process topology and the internal size of the sub-domain. The local grids are dynamically allocated as 1D arrays to ensure that they can be properly vectorized and by passing all

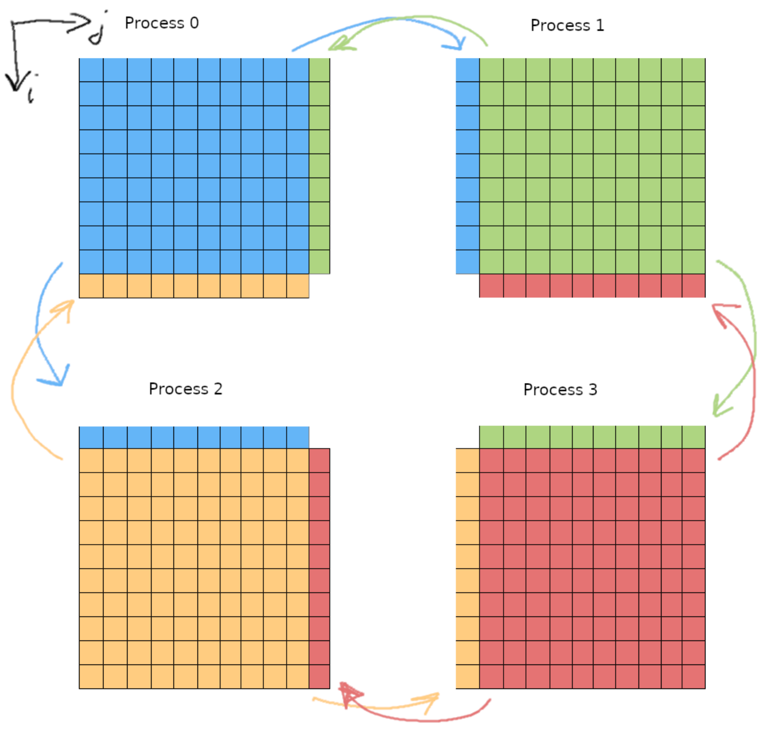


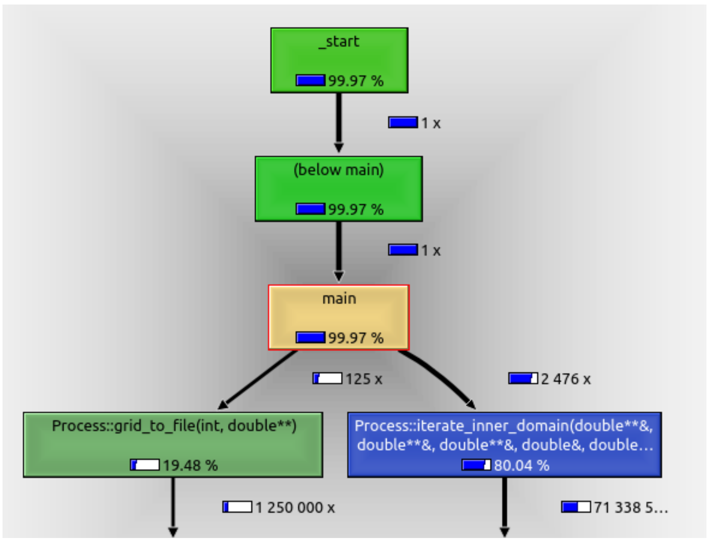
Figure 1: Communication schematic on he 2x2, 4 process domain with fixed boundaries (Own work)

the variables as references whenever possible, we try to reduce possible cache misses and de-localization.

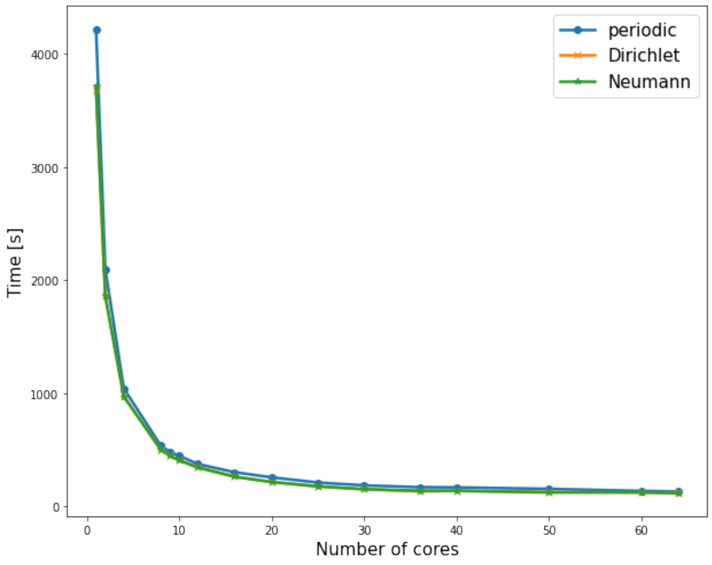
Finally, after the sends/receives are staged, the PDE iteration is performed on the inner domain, and after the receives and sends are complete, the boundaries are calculated. Also, if required, fixed boundary conditions are enforced. In the end, each process writes to its own output file, which can then be recombined using the post-processing Python scripts.

4 Performance Analysis

The performance of the program was profiled with the use of valgrid/callgrid for the base configuration in config.txt, with example of two most computationally functions shown in Figure 2.


_start
99.97 %
1 x
(below main)
99.97 %
1 x
main
99.97 %
125 x
2 476 x
Process::grid_to_file(int, double**)
19.48 %
Process::iterate_inner_domain(double**&, double**&, double**&, double&, double...
80.04 %
1 250 000 x
71 338 5...

Further testing in multicore/multi-process performance was carried out with aid of DUG HPC in Houston. The MPI program was passed in the form of loop job into the HPC scheduler, with each sub-job running different configuration of requested number of cores/processes or different sizes of the computational domain. The results of those job runs can be seen in figures below.



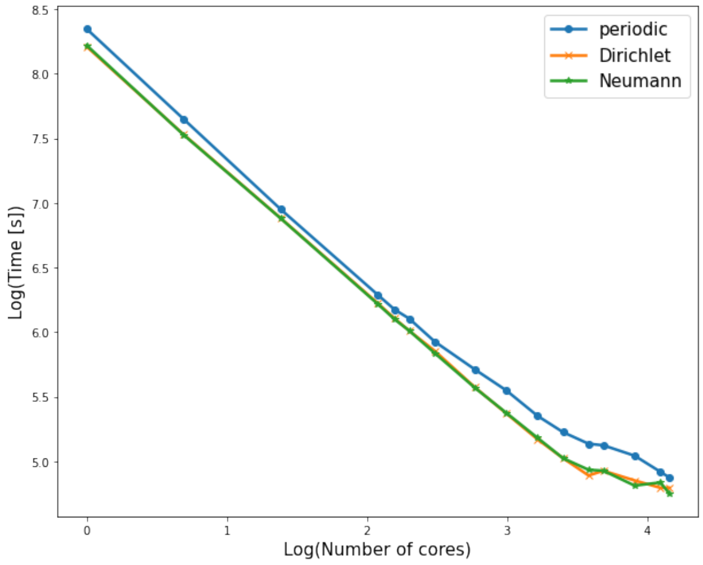
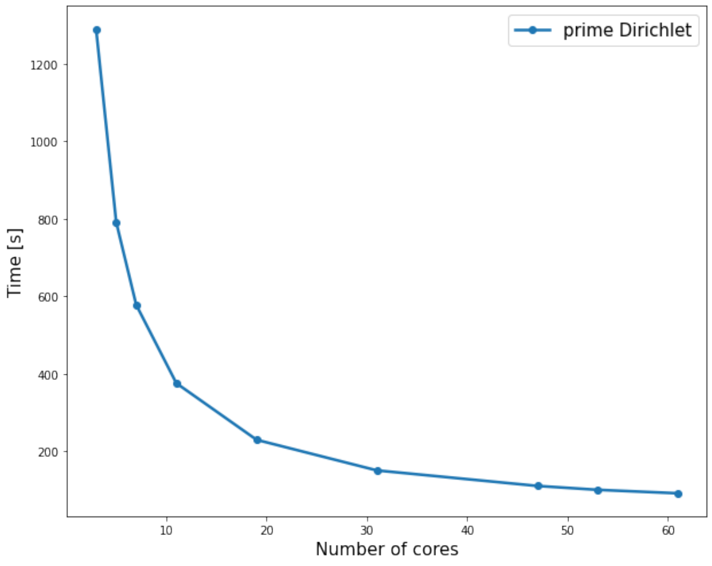


Figure 3: Top: Time of execution vs (non-prime) number of processors used for different types of boundaries. Bottom: log-log representation of the top graph. Test performed on 4000×4000 grid sized domain.

As we can see, the program does substantially improving the execution time of the single-core program which can be seen on Figure 3 and Figure 4 for prime and non-prime number of processes. We can observe that the time of execution follows a form of power law which depends on the number of processes. By investigating using a log-log graph, we can see that for small amount of cores up to around 30, the program has an execution gradient of around -1, which starts to flatten for higher number of processes. The Dirichlet and Neumann implementations, as well as those using prime number of processes, seem to be performing slightly better than periodic version, with lower log-log gradient and more stable higher end behaviour.



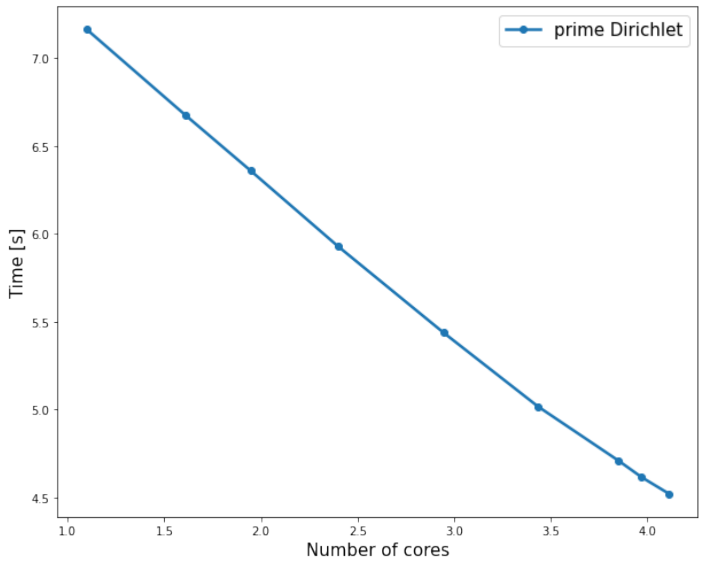


Figure 4: Top: Time of execution vs (prime) number of processors used for fixed, Dirichlet boundary. Bottom: log-log representation of the top graph. Test performed on 4000×4000 grid sized domain.

In Figure 6 which shows the behaviour of the program as we vary the grid sizes for fixed number of cores, we can observe from the log-log graph that for lower number of cores (4 and 12), as we increase the grid size, the gradient increases above the value of 2 for larger grid sizes for all three processor configurations. However, it should be noted that in case of 25 cores the initial test with size 100×100 takes longer time than for 200×200 grid, in contrast to other configurations.

Based on these profiling and performance results, we can attempt to explain and assess the reasons for this discrepancy in behaviour for different sets of processors used. If we once again look at the log-log results from Figure 3 and 4 we can see that the program behaves as desired within range of 2 to 30 processes, as it showcases the linear decrease in execution time with increasing number of cores, which perfectly matches the Amhdal's Law. We can indeed see this relative speedup for that range

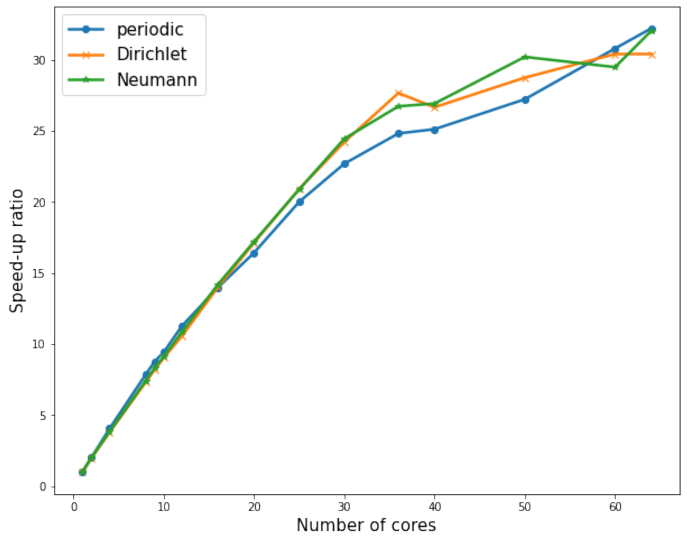


Figure 5: Graph of time speedup in relation to single core execution for the non-prime process numbers. (Based on results from Figure 3)

in Figure 5, which indicates that for those processes, the substantial portion of the code has been parallelized and vectorized. This, combined with our profiling outcomes, seem to indicate that our program is not CPU-bound but network/communication and IO bound. This hypothesis would explain the behaviour observed in Figure 6, as when we increase the dimensional size of the domain, larger communication buffers need to be involved and more idle time is spent waiting for communication to happen. Moreover, the anomaly in timing of 25 core program on small 100×100 domain can be understood, as the overhead cost involved in starting, managing and finalising communications on that many processes would be far greater than grid computation itself.

Finally, a factor not mentioned before but which could make significant difference for these large scale problems would be the use of higher level of abstraction in code, such as the use of object-oriented programming and addition of features, which although add to readability and sustainability of the code, but overall might incur extra costs in these situations when performance really matters.

5 Evaluation

Although not without struggles, this project can be deemed to be successful in its goal in not only increasing the performance of the serial implementation of wave equation, but was also as great learning opportunity which made me realise the hardships of writing readable and efficient parallel code. Nevertheless, this program could be further improved upon, with greater development still possible in many parts of the code, such as implementing better communication and decomposition algorithms and parallelizing the grid to file writing.

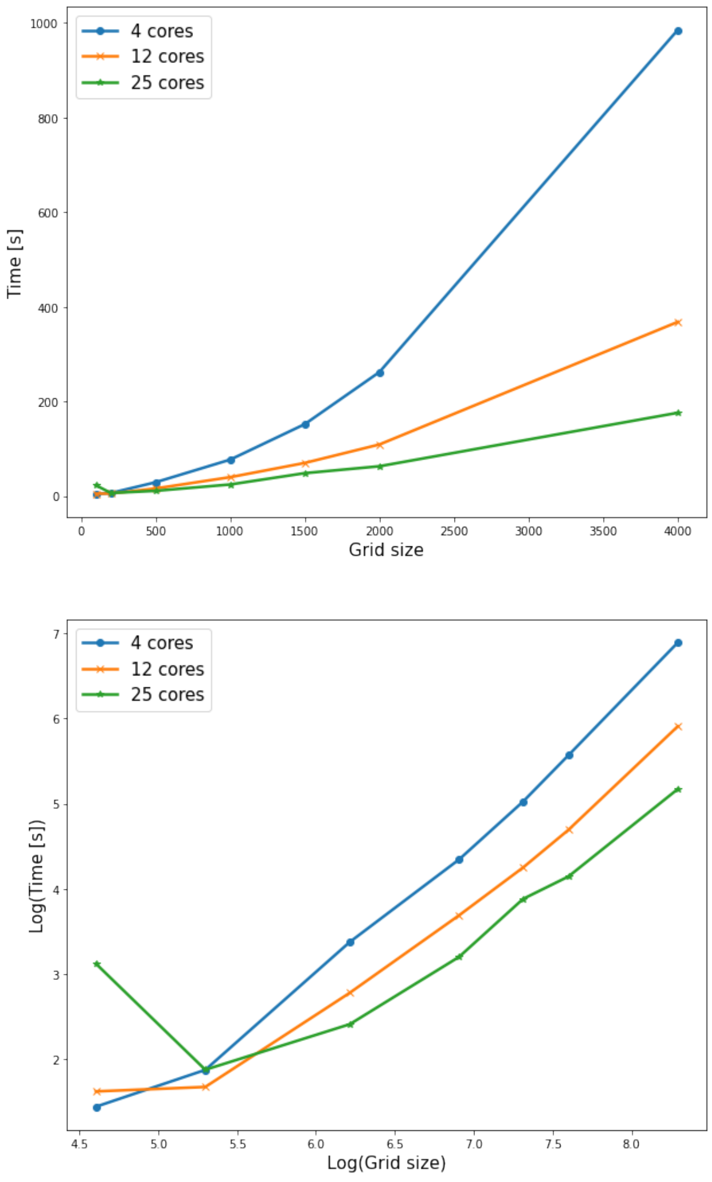


Figure 6: Top: Time of execution vs domain grid size (with periodic boundaries). Bottom: log-log representation of the top graph. Test performed with 4, 12 and 25 cores