MPI Coursework Report - Parallelising the wave equation

Prashasti Tiwari

February 25, 2022

1 Introduction

In this report, the wave equation is considered, starting with a given serial code and is then parallelised. Parallel code is when the domain of the calculation can be split evenly by the different process, thus allowing a faster computational time. It can also be run on HPC with a larger number cores to further decrease computational time. The aim of this report is to parallelise the serial code, then the data gathered from the parallelising is compared with the serial code using an animation. The use of different number of cores with over different machines and time taken for each is considered and is analysed. There are factors such as the speed up ratio and the parallel effiency that is considered. These are considered both locally and on the HPC.

2 Method

The serial code for the numerical modelling of the wave equation is the starting point of the project, which calculates the solution with a forward central time stepping scheme. The time stepping scheme is kept the same in the parallel code and is done with the use of the library MPI.Parallel code involves breaking down the domain into chunks and then calculating the value for the particular chunk, which can then be exchanged with neighbours, after which calculation can be repeated, and the communication.

The first step involves finding the domain and is given by the function *find_domain*. This decomposes the number of process and assigns number of rows and columns based on the number of total points needed to calculate. Following this, the *id_to_index* function decomposes the id into its rows and columns and *find_neighbours* finds the ids of the neighbouring processes. This is useful so that the information can be communicated later. Then the number of points assigned to each grid is calculated by dividing the of imax and jmax data points, which in this case are chosen to be 500, by the number of rows and columns respectively. This is then used to create a 2D grid of points for each processor which stores doubles and includes the ghost nodes. This is done through the *create_grid* function.Then the grid is initialised using the *initialise_grid* function.

For the communications with the neighbours to take place, the MPI library requires a particular type of data to be passed. Here, different datatypes are created as the information exchanged with different neighbours are of different sizes, based on the top, bottom, left and right neighbours. The data types are created in the *createdatatypes* function. An initial condition is set to the grid using the *initial_condition* function. After which dt and output number can be incremented. A while loop for sending and receiving the data is set up before every iteration, and then data can be printed to grid. Non blocking communication is used here so that all the data can be sent to the neighbours and then received before the next iteration. This is done using 4 *MPI_Isend* and 4 *MPI_IRecv* for each neighbour and *MPI_Waitall*. After this the created datatypes are freed and memory for the arrays is also deleted before *MPI_Finalize*.

The program is run for different number of cores and the time is recorded. For each number of cores, the average value is found with over 3 runs and then also compared with spreading over different number of machines. An animation of the wave is also given in the folder.

The outputs can be used to assess the performance of the code. The Speed up ratio, the efficiency and the fraction of processes parallel are calculated. The Speed up ratio is calculated using the following formula:

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

The Parallel Efficiency is given using the formula below and is expected to drop with the increase in the number of cores.

$$P = \frac{S}{N} \tag{2}$$

This is then used to calculate fraction of code executing in parallel f. The S can be expressed in terms of f given below and will be assessed in next part for comparison of the performance, this Amdahl's law. The f is expected to remain constant.

$$S = \frac{1}{1 - f + \frac{f}{N}} \tag{3}$$

The above equations can be rearranged to give f as the following:

$$f = \frac{\frac{N}{S} - N}{(1 - N)} \tag{4}$$

These properties are presented in the section below.

3 Results

The following shows the results obtained for running the process in parallel on the HPC, and different times obtained for different number of cores, speed up ratio, parallel efficiency and fraction of the process in parallel.

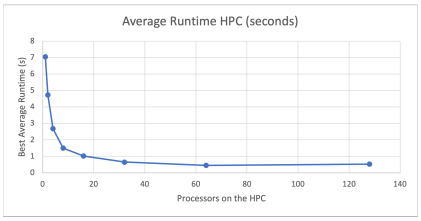
Table 1: A table show the different times for different process allocation and properties calculated based on the time

Number of cores	***Nodes***	***NCPUs***	***Micro-processes***	***Average Runtime HPC (seconds)***	***Speed up Ratio HPC***	***Parallel Efficiency HPC***	***f (fraction executed in parallel)***
1	1	32	1	7.05	1.00	1.00	0
2	1	32	2	4.73	1.49	0.75	0.66
4	1	32	4	2.68	2.63	0.66	0.83
4	2	32	2	1.79	3.93	0.98	0.99
4	4	32	1	2.38	2.97	0.74	0.88
8	1	32	8	1.50	4.70	0.59	0.90
8	2	32	4	1.76	4.00	0.50	0.86
16	1	32	16	1.02	6.92	0.43	0.91
16	2	32	8	1.13	6.27	0.39	0.90
16	4	32	4	0.87	8.14	0.51	0.94
32	1	32	32	0.65	10.86	0.34	0.94
32	2	32	16	1.28	5.52	0.17	0.85
32	4	32	8	0.95	7.39	0.23	0.89
64	2	32	32	0.41	17.35	0.27	0.96
64	4	32	16	0.54	13.02	0.20	0.94
128	4	32	32	0.53	13.36	0.10	0.93

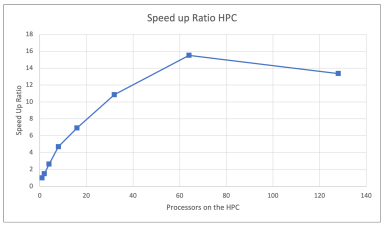
Here the number of cores were tested for various different machine numbers and processes to assess performance. It is as expected that the parallel efficiency reduces with the number of cores is increased.This will presented in a graphs below. It can be seen that the processes spread out over least number of machines perform better. For example, 8 cores over two machines has a slower time than 8 cores over 1 machine. Fraction executed in parallel for the first two process is the smallest and then it increases. The figures below collect the best run time of the processes and plot those as well as Speed up ratio, Parallel efficiency and fraction in parallel execution.

4 Analysis

From the above data, it can be seen that when the processes are split over the nodes, the same number of processes over fewer nodes is takes a shorter time to execute.

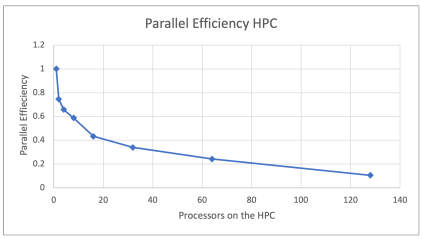


(a) Best Average Run times for HPC

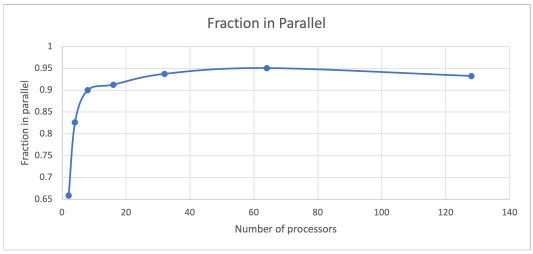


(b) Speed up Ratio HPC

Figure 1: Comparing how the number of processes affects the runtime and speed up ratio



(a) Parallel Efficiency



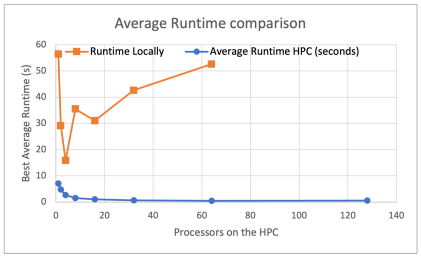
(b) Fraction in Parallel excution HPC

Figure 2: Comparing how the number of processes affects the Parallel Efficiency and fraction of code execution in parallel

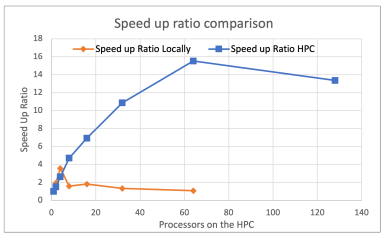
In figures 1 and 2, it can be seen that the as expected the run time drops exponentially with the number of processors. This is a decay curve and with increase in processors, the time decrease in time decreases. The speed up ratio increases however for 128 processor decreases slightly, which could be an anomaly. The speed ratio can be linear however, in this case smaller tasks have better cache storage, making it more efficient. Amadahl's law also estimates that the speedup cannot be bigger than $\frac{1}{(1-f)}$. For case studied here, the average f excluding the first 2 number of cores is 0.91, which gives $\frac{1}{(1-0.91)}$, and therefore the theoretical limit for speed up is 11.1. However the largest value here is 17.35, which could be due to the inefficiencies of the parallel code. This could be due to the domain decomposition with processes

The parallel efficiency as expected and decreases with the number of cores as is a decay curve. This is as expected due to Amadhal's law with E tending to 0. In Figure 2b, the fraction in parallel execution is presented. Comparing the fraction executed in parallel values in figure 2b it can be seen that are largely constant beyond 4 processes. It increases for 1 to 2 and 2 to 4 cores however remains constant thereafter. This is as expected, as the value of the fraction of code execution in parallel should be the same regardless of the number of cores.

In the figures below, the average runtime is compared with local machine runtime with cores over 8 being oversubscribed. The speed ratio, Parallel efficiency and fraction in parallel are also compared.



(a) Best Average Run times for HPC vs local



(b) Speed up Ratio HPC

Figure 3: Comparing how the number of processes affects the runtime and speed up ratio for HPC vs local machine

The runtime for the local processes is greater than runtimes on the HPC as expected, and the change in the speed up is negligible as there is not much change in the run times. The run time is larger as processes

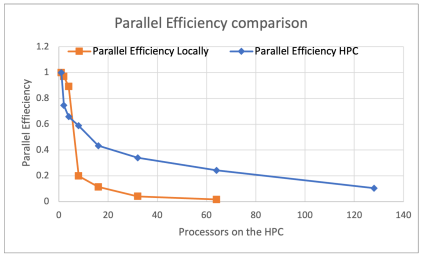


Figure 4: Parallel Efficiency Comparison of local vs HPC

greater than 8 are oversubscribed threads. The parallel efficiency path is similar to the parallel efficiency of the HPC, therefore it is as expected.

5 Conclusion

Here, different factors were assessed and compared to the local values, this is was as expected with the over-subscription of the processes. Comparing the parallel efficiencies a decay curve is seen. The speed up ratio is linear for some parts, however when executed with smaller number of processors there is less cache. The speed up ratio locally is negligible. Largely, comparing the parallel code was as expected, however with the speed up ratio being an anomaly for 64 processors.

In general, the time taken for the wave equation to be computed is lower as expected, however after increasing the processors to a certain number, time decrease can be modelled as a decay curve. Thus allocating more processors has a diminishing return.