High Performance Computing (CS402) Coursework 2

Vishwak Senan Ganesan

University ID: 2034389

Vishwak-Senan.Ganesan@warwick.ac.uk

University of Warwick

1 Introduction

High-Performance Computing coursework 2 contains a mini application which solves Navier-Stokes equation in a 2-Dimensional mesh. The goal of this coursework is to parallelize the application using Open-MPI. The project uses a four stencil cell style, so each cell is dependent upon the top, left, bottom and right cell's value. I have used advanced MPI approach to solve the parallization problem.

2 Decomposition Techniques

Usually in an MPI program, there would be decomposition problem where we have to decompose the source grid into multiple grids so that each processors can access it.

I have used 1-Dimensional decomposition technique to allocate the grid into sub grids and then apply the process only on the sub grids. The big grid (the first initialized grid) will be separated along the rows. let us say the size of the mesh is 10 x 10, then the sub grid size size for 2 processor setting would be 5 x 10 grid and another 5 x 10 grid.

Reason for 1-Dimensional Decomposition is that we can segregate the array into grid regardless to the number of processors. Let us say we do a 2D decomposition on the grid, then we have separate the grid along rows and along columns. For a prime number of processors (ranks) - let us say we have 7 ranks - we cannot get 2 dimensions separation of processors as 7 is a prime number and the separation would be 1 x 7 which is anyway a 1D decomposition.

Example case of decomposition is that we can split the grid with the number of processors like this.

Let N - be the number of ranks.

Let X - be the size along x-axis.

Let Y - be the size along y-axis.

First we calculate the values of divisor and the remainder by dividing the X upon N.

Divisor = N/X and Remainder = N % X.

Now when the ranks are accessing the parallel part of the code, the size for the final rank would be (Divisor + Remainder).

For example N = 4, X = 10 and Y = 10. The divisor is 2 and remainder = 2.

1. Rank 0 grid size = (2 x 10).
2. Rank 1 grid size = (2 x 10).
3. Rank 2 grid size = (2 x 10).
4. Rank 3 grid size = (4 x 10). The reason is the final rank has the size of divisor + remainder.

3 Grid Communication

Now that we know how to segregate the grid into sub grid, after each and every process done on the sub grid, the grids needs to communicate with each and every processors as the data stencil is dependant on the four parts of the grid. To solve this problem, we use the concept of ghost cells. Ghost cells / halos are extra data in the sub array which holds information about the next processor's grid boundary details.

In the previous example, for a 10 x 10 grid for 4 processors, we learnt the decomposition sub grid sizes. But after adding ghost grid to each sub processors, the size would increase from (2 x 10) to (4 x 10) as the top row will contain the bottom row of the previous rank and bottom row contains the information of the next rank.

First, we create and update these halos and then start the process for each of the sub grids. Once the grid is updated, we have to constantly update the neighbouring rank about the updated value. For this, we used blocking MPI_Sendrecv API function as it sends and receives data at the same time. This would help us prevent deadlocks that can happend between multiple processors.

4 Reduction Technique

The process on these sub arrays is done and then at the end of each iteration, it is being updated about it's boundary data to the nearest neighbours. Once this is done, we have to gather all the data to one rank, so to combine and produce the final output. I used MPI_Reduce API function from Open-MPI where we send the data on the following process and operation.

1. Create a grid with the original size and fill it with zeroes.
2. Since we know the position of x_min, x_max, y_min and y_max from the original array, we update the values only in the position and then the rest stays with the value zero.
3. Vectorize the 2D array into 1D array so that we can reduce using MPI_MAX operation. (This process is done as Open-MPI does not support the reduction on 2D).
4. Reduce all the data to the rank 0.
5. Receive the reduced data in rank 0.
6. De-vectorize the 1D array into original mesh.
7. Write the final output in the bin file.

5 Code changes

To keep the code clean and simple, I have created a file mpi_process.h file which contains all the needed functions to do the parallel processing. Little explanation of the function declaration is given below.

mpi_process.h contains these functions.

1. zero_grid() - assings the value to the full grid.
2. update_subarray() - assigns the values from the source array to the subarray according the min, max coordinates.
3. update_halos_for_subarray() - Assigns the halos cells values from the top and bot rank respectively.
4. vectorize() - Convert the 2D array and returns the 1D form of that array.
5. devectorize() - Convert the 1D array and return the 2D form of that array.

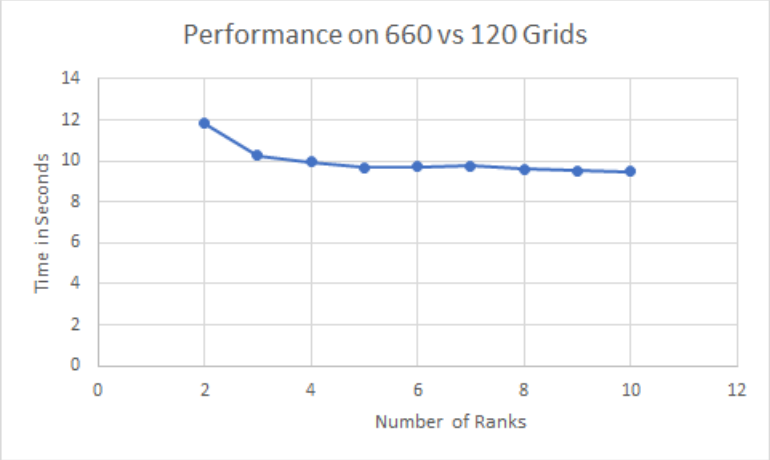

Figure 1: Performance for the grid 660 x 120 size

6. get_sub_grid_details() - Given a rank, world size, rows and colums, it will return the x size, y size, x min, x max, y min and y max as a list (in that order).
7. copy_to_global_array() - Exclusively used for the reduction operation and to update the zero grid.

Apart from the header file I have created, some changes were done inside the karman.c file

karman.c file includes these changes

1. Before main loop, all array's sub grid is created along with the halos values.
2. Inside the main loop, the send/recieve operation is implemented to update the halos.
3. Reduction operation is implemented after the main loop.

6 Evaluation of the code

The evaluation is done on these 2 methods, having the same size and increasing the ranks and having the same rank and increasing the grid size along x axis (choosing x axis here as I am decomposing along rows). The initial profiling is done as follows in table 1. From the table 1, we can understand that the poisson function is our main target to parallelize.

The parallelization performances are given in the figure 1. These metrics are obtained by running it on DCS machines and since kudu's queue was full when I was testing, I used the DCS machine's results itself. This figure 1 tells us about how our model performs when it is given more number of ranks with the default constant grid size of 660 x 120.

The parallelization test for increase in grid size is given in figure 2. This test is also done with DCS machine as kudu systems are full. The results may not be promising as the increase in number of process has less effects on the parallelization and increase in size also doesn't bring that much increase in execution time. But the parallized program has significant improvement on the serial version of the code. This is because the loops work on the smaller scaler and smaller size.

7 OpenMP Implementation

Since I have used the hybrid approach to solve the parallelization on the karman application, this section contains the detailing of the openMP code. OpenMP is used only in loops where there is no reductions or independancies in the code. This situation happens only while we assign the values in

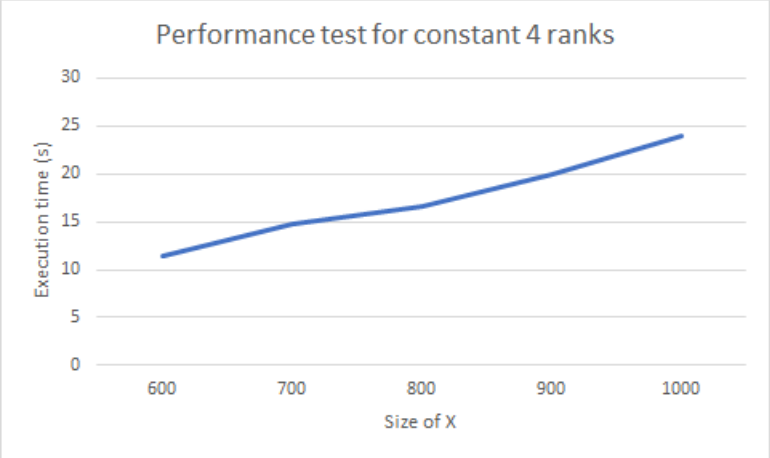

Figure 2: performance on increase in X axis size with 4 ranks.

Function	Time percentage	Time (s)
Poisson	94.70	24.12
compute_tentative_velocity	4.44	1.13
update_velocity	0.47	0.12
compute_rhs	0.28	0.07
set_timestep_interval	0.20	0.05
apply_boundary_conditions	0.04	0.01

Table 1: Serial Code Profiling

the loops and create memory space for columns. Hence pargma clauses are used only for assignment and memory utilization for loops. Since I have used OpenMP in the application, I have made changes in Makefile and added -fopenmp flag to use the pragma section.