MPI Programming Coursework– Solving the Wave Equation

Zonghui Liu

Problem description

The aim of this assignment is to write a parallel solver for the wave equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u$$

Where u is the displacement and c is the speed of the wave. A simple way of discretising this problem is an explicit finite difference discretisation(FD). We need to reduce the computational time for long-term simulation of physical processes using MPI library. Based on the finite difference method(FDM), the following equation can be used to update the grid and then the iteration:

$$u_{i,j}^{n+1} = \Delta t^2 c^2 \left(\frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2}\right) + 2u_{i,j}^n - u_{i,j}^{n-1}$$

Note that the superscript n refers to the time step and is not a power.

To reduce the running time, we can use MPI library to set up several processors. Each processors will be independently responsible for an area of iteration/computation, and the processors can communicate with each other.

Data Type and Structure of codes

Important Data Types

- **To record the status of the current grid and prepare for updating**: Use the std container **vector<vector<double> >** grid to record the status of the grid and the old_grid and new_grid in the same type to update the status.
- **To communicate with neighboring nodes**: Create my own MPI data types for sending data to neighboring nodes and receiving data from them at the same time. Four data types for sending and four for receiving. These four data types including left, right, top and bottom edge. Every data type include the lengths of block, the type of these data and the offsets(the absolute addresses - the address of the first element).
- **To communicate with neighboring nodes**: Use **non-blocking** communications (**Isend, Irecv**) for sending and receiving data. The reason that I use non-blocking communications instead of the blocking one is to increase the speed of code

improve the efficiency. **MPI_Waitall** is used to ensure all the communications are done successfully.

- **To load data from the current node**: To send data from the current node to other neighboring nodes, I use a 2-D array named **send_data** to buffer the data of grid without ghost layer.
- **To store receiving data from neighboring nodes**: We need to store the receiving data sent by neighboring grids(with ghost layer), so I used a 2-D array named **recv_data** to store these data and prepare for update the boundary data in the current grid.

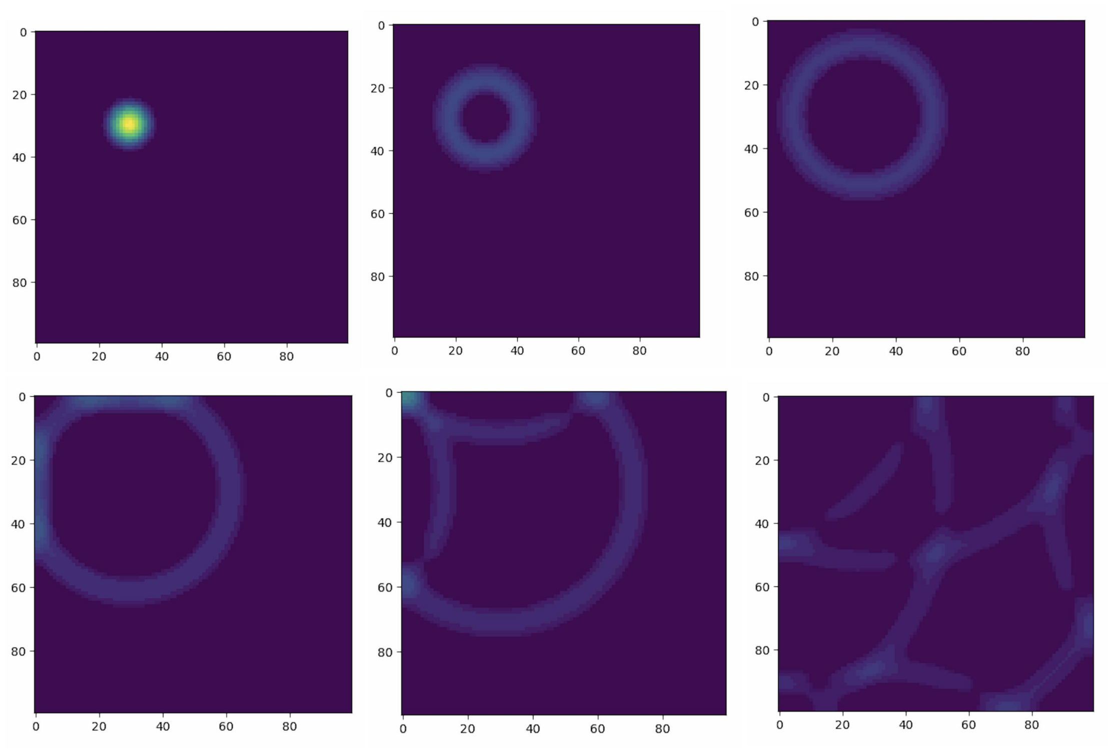
What the code implements

- Divide the whole domain into several nodes based on the number of processors, assign each node to a processor and allocate the data to each processor.
- Initialize our system and print the status to files.
- Prepare for communication between processors: create MPI data types and initialize the sending and receiving buffer.
- Do iteration: update each point in the current node and set the boundary conditions. My code can implement three kinds of boundary conditions(Dirichlet, Neumann and periodic), unfortunately, due to time limitation, the periodic one has not tested already and didn't generate the animation(so there are two kinds of animation).
- Calculate the running time.

Results

The results from different processors will be stored in different files(.dat files) to improve the code speed and then they will be combined in post-processing part.

The following pictures show the status at t = 1s, 2s, 3s, 4s, 5s, 6s respectively. The complete change process can be seen in the gif file in the post-processing folder.



Performance Analysis

Running Time Comparison

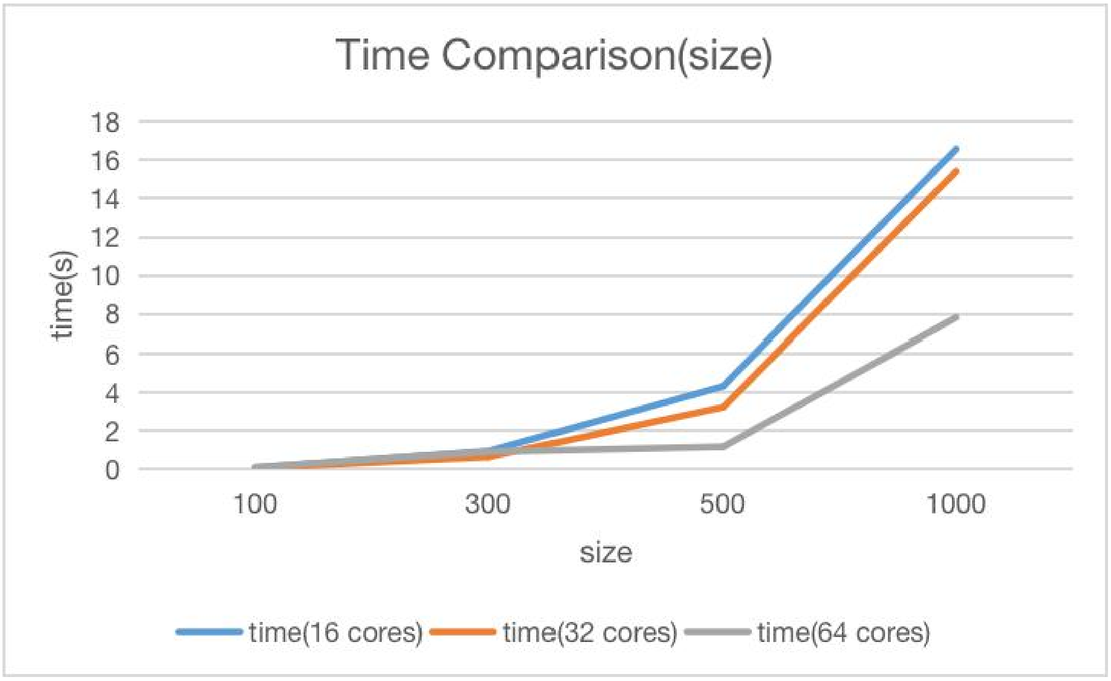


Fig 2.1 Running Time-Size

Fig 2.1 shows that the running time gradually increases as the problem size increases.

	time(100*100)	time(300*300)	time(500*500)	time(1000*1000)
1	1.19523	23.9315	68.898	777.847
2	0.515018	8.98167	23.6484	268.433
4	0.240811	4.79321	18.141	222.187
8	0.128315	2.06609	5.76277	61.743
16	0.081096	0.938285	4.31265	16.5687
32	0.073699	0.639502	3.21995	15.4149
64	0.103943	0.907108	2.17523	7.82821
128	0.194373	1.11587	2.00648	4.39984
256	0.870483	1.20288	2.72253	4.69584

The table shows that the running time decreases and then increases as the number of processors increases. This is because when there are too many cores, the ratio of communication in the overall running time increases, which is not the case we want.

What's more, when the system with only one core, the running time will be significantly long if the size of domain is large. It shows that parallel pattern make a lot of sense in our scenario.

Speedup Ratio Comparison

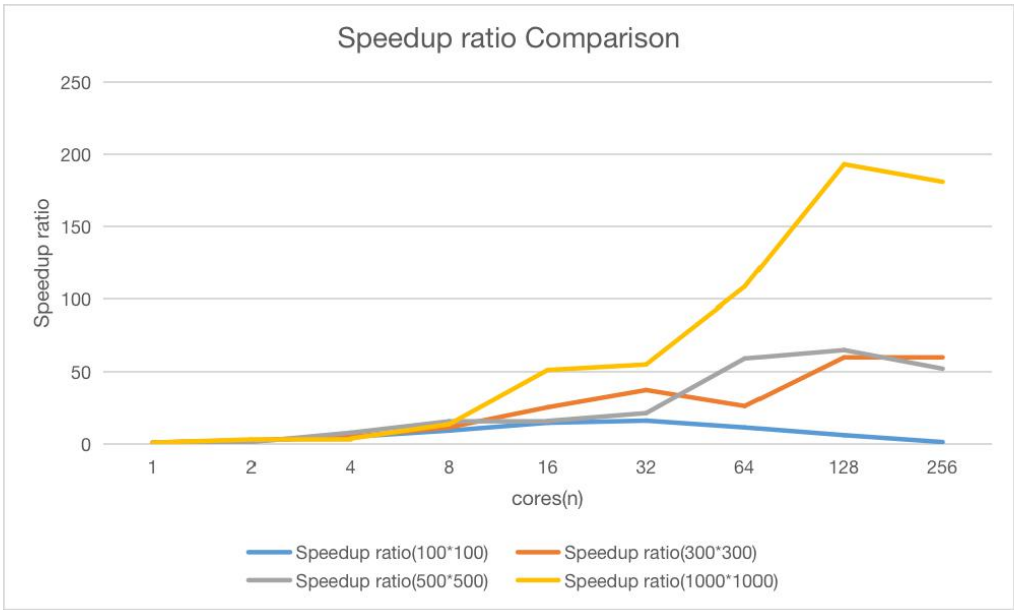


Fig 2.2 Speedup ratio-Cores

Speedup ratio means the number of times quicker the code is in parallel relative to the serial code.

Fig 2.2 shows that the speedup ratio gradually increases as the number of cores increases. And the speed up following the Amdahl's Law.

Parallel Efficiency Comparison

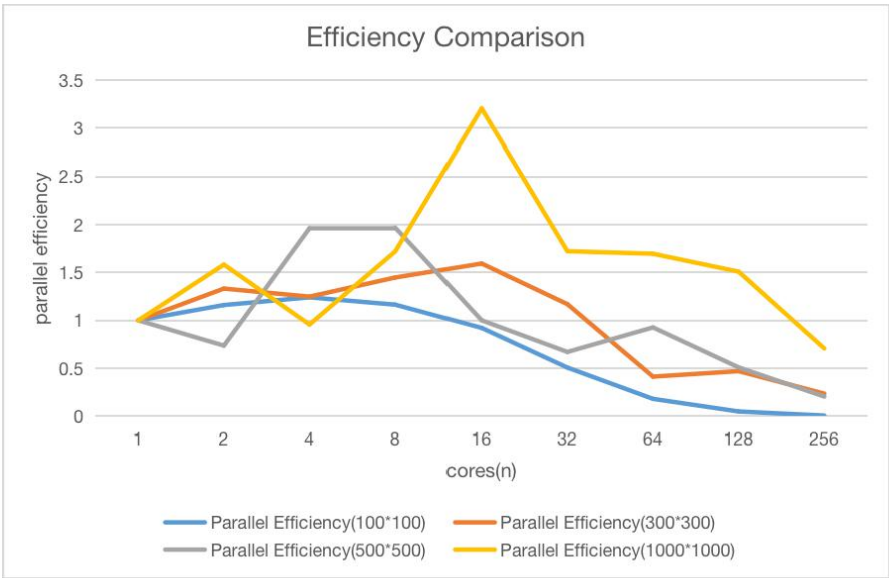


Fig 2.3 Parallel Efficiency-Cores

Fig 2.3 shows that the parallel efficiency will usually drop as the number of cores used increases. When the number of processors increases, the number of communications will also increase and then the communication ratio will increase at the same time, finally the efficiency decreases.

However, I do not know the reason of the special point(size=1000 and cores=16).

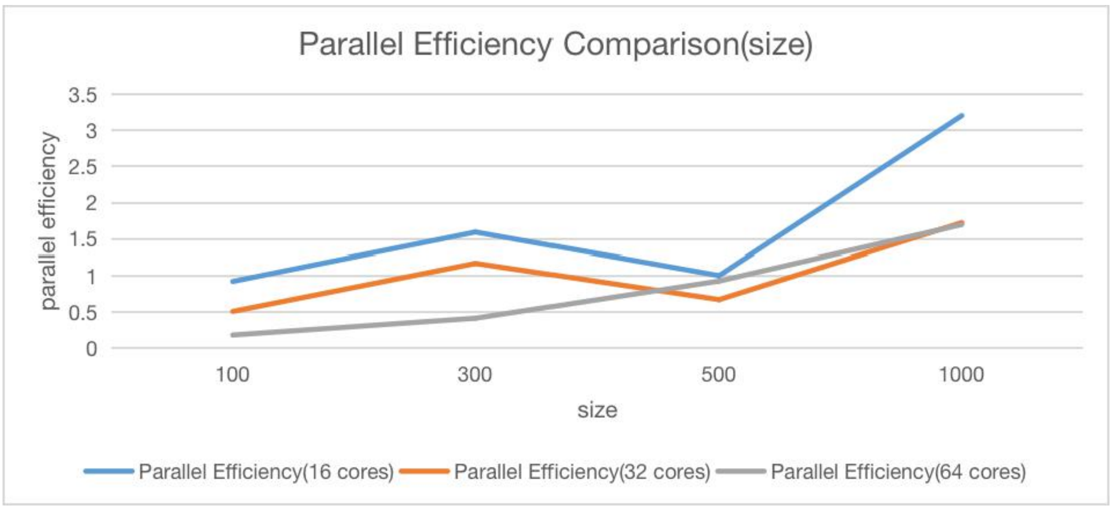


Fig 2.4 Parallel Efficiency-size

Fig 2.4 shows that the parallel efficiency increases as the size of domain increases.

Improvement

- Due to time constraints, the periodic boundary conditions are not be implemented by now. But the key of that part is how to get the neighboring nodes periodically which is similar to the Conway's Game of Life.
- The data types to store grids is a vector and it can be changed into a 1-D array. 1-D arrays will have higher efficiency.
- The data types to buffer the sending data and receiving data is 2-D arrays. They can be 1-D arrays which can be indexed faster.