MPI Programming Assignment– Solving the Wave Equation

Seyedeh Raha Moosavi

Introduction

The Finite-Difference has been widely used for solving problems. The main advantage of the FD method is that it is straightforward solution.

The reduction of computational time for long-term simulation of physical processes is a challenge and an important issue in the field of modern scientific computing.

MPI is a library of functions (subroutines in Fortran) that can be called from a C, C++ or Fortran program to enable communication between processors. It is actually deed in a language-independent way, so that in theory it may be implemented for any high-level language.

- 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 discretizing this problem is an explicit finite difference discretization. For two spatial:

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

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

The domain is a grid defined by:

$$t = t_0 + n\,\Delta t \qquad x = x_0 + i\,\Delta x \qquad y = y_0 + n\,\Delta y$$

Where n, i and j are integers.

The discretized equation is explicit and can be rearranged as follows:

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

Note that you don't need to store all of the time steps in memory (and you are likely to run out of memory quite rapidly if you try from all but the smallest problems). You will need to store 3 grids, one for the previous time step ($n-1$), one for the current time step (n) and the new time step ($n+1$). When moving to the next time step try and avoiding copying the data and rather just swap pointers (MUCH quicker).

As this is an explicit solution there are strict time step constraints:

$$\Delta t \ll \frac{min(\Delta x, \Delta y)}{c}$$

Design and structure of codes

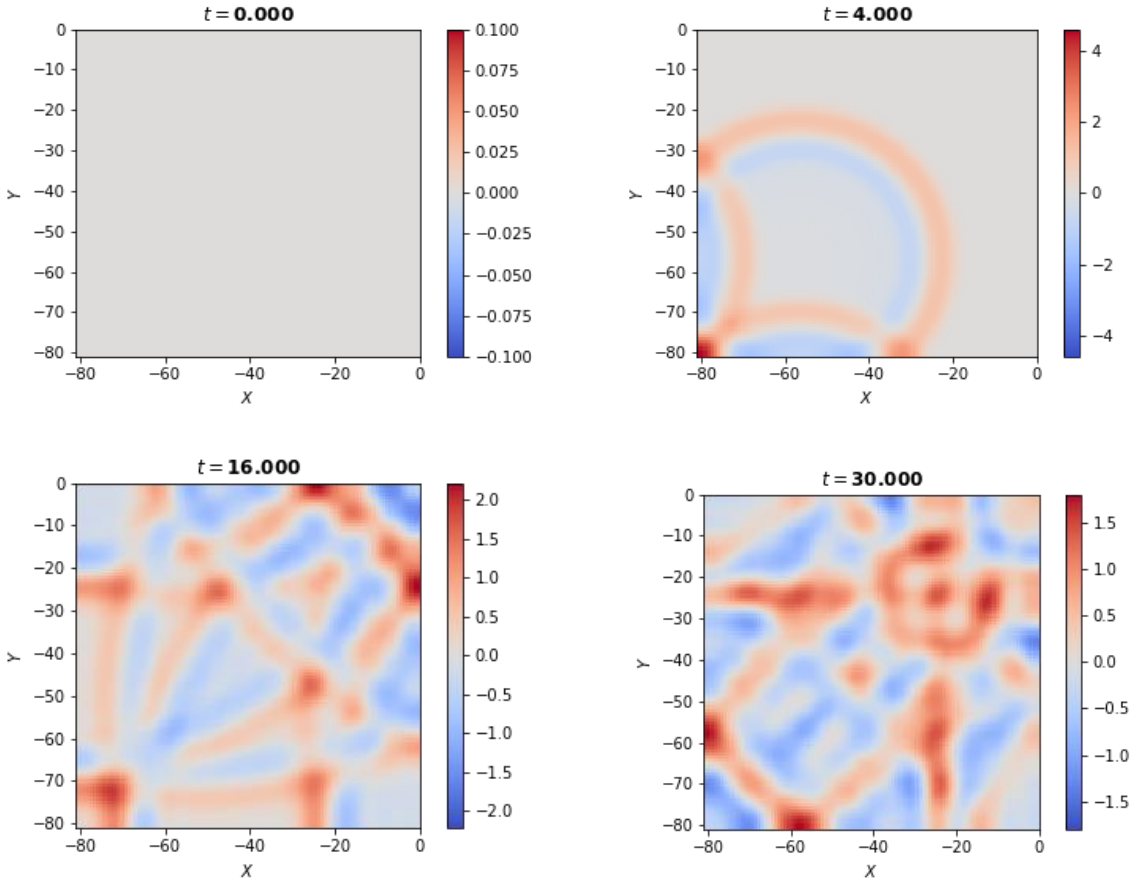
For a parallel program using MPI, firstly, some modification has been applied to the serial code version. I changed the 2-D vectors to 2-D array mapping in 1-D arrays because 1-D arrays can increase the speed of code.

Then starting to parallelized the code using several MPI functions including:

- Creating my own datatypes for sending and receiving data. Four datatypes for sending and four for receiving. These four datatypes including right, left, top and bottom edge which their addresses have gotten and then offsets list calculated using addresses subtract by address of start of grid.
- Using non-blocking communications (Isend, Irecv) for sending and receiving data. It can increase the speed of code, because all data send and receive and then using MPI_Waitall using requests, all wait for data to communicate.

The result from different processors were stored in different files to improve the code speed and then they have been collected together in postprocessing.

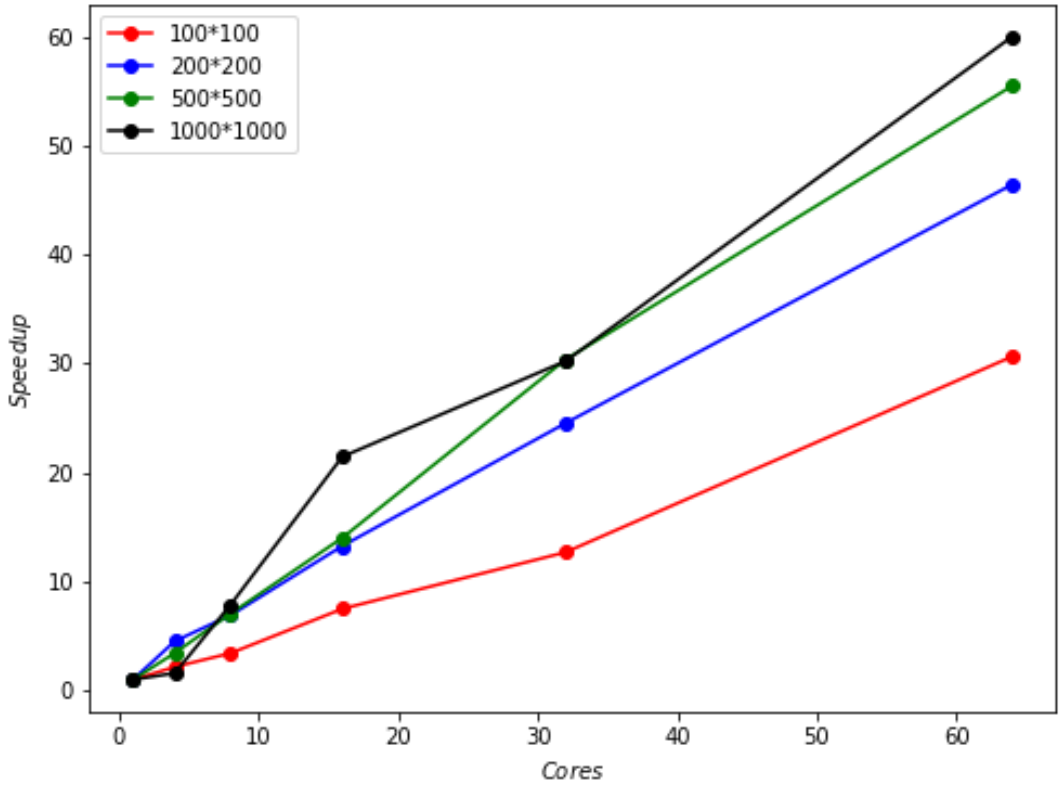
The results for solved equation using finite difference at different time steps plotted in the following figure.



Performance tests

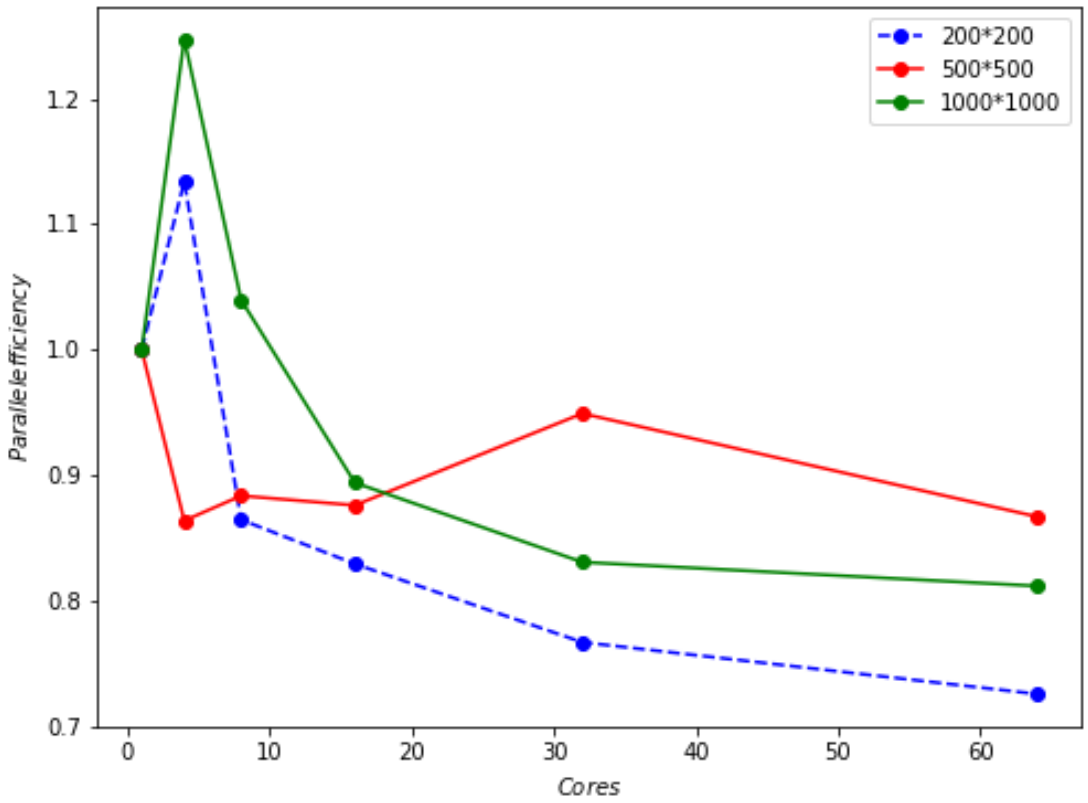
In the graphs below, I have shown how our code scales with using more threads. I have used dug system for running parallel codes on 4, 8, 16, 32, 64 cores. I used C++ and the Visual Studio compiler on Windows using MPI. Tests were run on a Serial implementation and Parallel implementation using MPI for 4, 8, 16, 32, 64 cores for the following grid sizes:100x100, 200x200, 500x500 and 1000x1000. For each grid size, I executed the code 3 times and used the averages of the 3 runs. The following diagrams show the run time of each implementation and speedups in relation with the serial implementation respectively:

The runtime increases with increase in grid size, regardless of the number of cores used. For better understanding of the performance improvement with increasing number of cores and I plotted each cores run time divided by one threads run time:



Results demonstrated that increasing the number of cores improved running time for all grid sizes. The speed up following the Amdahl's Law.

The following figures shows the efficiency of parallel code versus number of cores for different grid Size. The efficiency calculated using divided the speed up calculated above by number of cores.



Memory management and Improvements:

Efficient allocation and deallocation of memory during the different part of program and different function to ensure zero memory leaks.
Datatypes created and freed after using it for nonblocking communications using Isend, Irecv. It can improve efficiency since it is possible to send or receive an array of data.
Using non-blocking communication can also improve the speed of running program by sending some requests and then wait for request to receive and done after.
Changed the vector to arrays and then Mapping 2-D arrays on 1-D arrays to improve the runtime speed. For 100*100 grid size, the changed from vector to 1d array will reduce the run time by 23%.

References:

Ruzhansky, M., Altybay, A., & Tokmagambetov, N. (2020). A parallel hybrid implementation of the 2D acoustic wave equation. *International Journal of Nonlinear Sciences and Numerical Simulation*.