# ME 489
# Homework 5 - MPI Implementation of Explicit In-time Two-Dimensional Finite Difference Solver

Ata Bora Çakmakcı and Mustafa Yiğit Görgün

Middle East Technical University

7 January 2024

# Contents

# Chapter 1

# Introduction

In this study we parallelized an explicit in-time, two-dimensional, finite difference solver using MPI. The wave equation to be solved by the already provided solver is as shown.

$$\frac{\partial^2 q}{\partial t^2} = c^2 \left( \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} \right), 0 \leq x \leq 1, 0 \leq y \leq 1$$

The equation is discretized utilizing second-order finite-difference method to approximate the value of the field variable $q$ at the next time step.

$$q_{i,j}^{n+1} = 2q_{i,j}^{n} - q_{i,j}^{n-1} + \alpha_x \left( q_{i+1,j}^{n} - 2q_{i,j}^{n} + q_{i-1,j}^{n} \right) + \alpha_j \left( q_{i,j+1}^{n} - 2q_{i,j}^{n} + q_{i,j-1}^{n} \right)$$

Where the next value of $q$ is determined by using its current, past and neighbouring values. An example solution at the last time step with trivial values is shown in Figure 1.1.

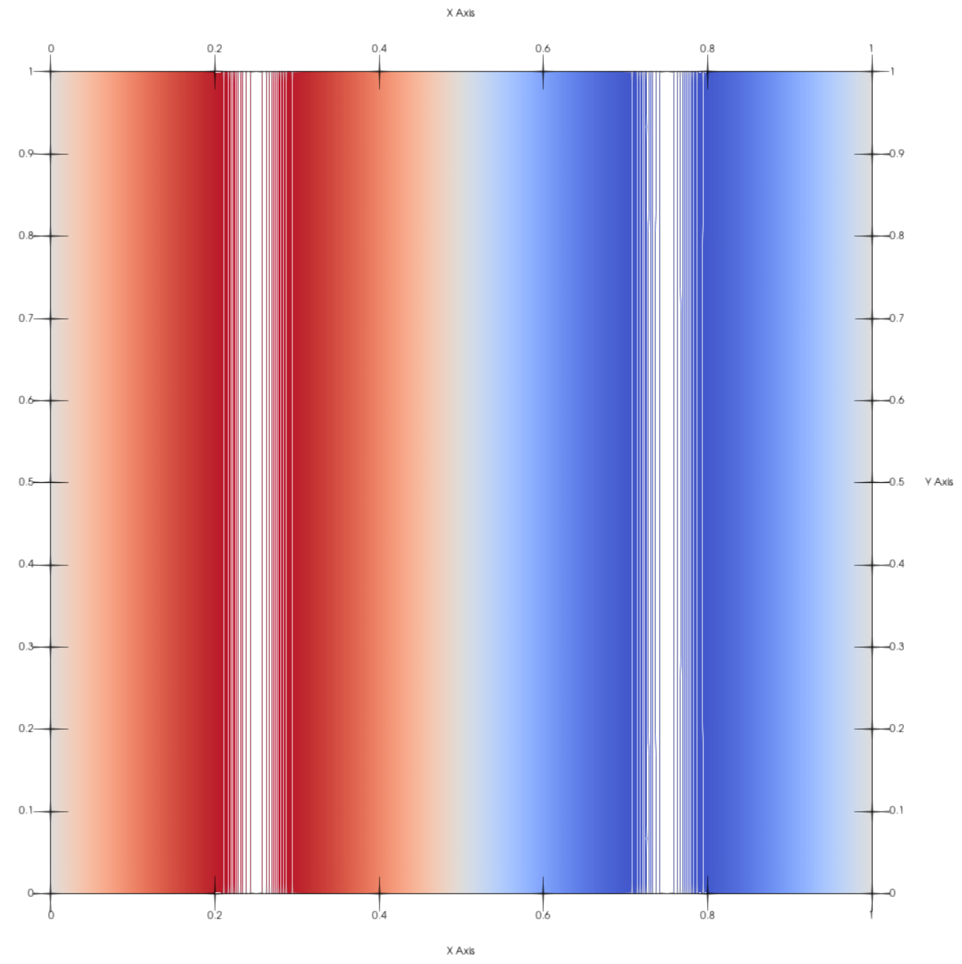


Figure 1.1: Post processed solution at the last time step for a problem size of 1020*1020

# Chapter 2

# Implementation

## 2.1 MPI Implementations

To start with we included the MPI header and initialized MPI. The rank and size of processes are obtained.

## 2.2 Domain Decomposition

Since we are instructed to assume the number of nodes in the Y direction is divisible by MPI size, we start off with dividing total number of nodes in Y direction to the MPI size and storing it as the local number of nodes in the Y direction. When computing the coordinates of the nodes we need to make sure that y-coordinates of the nodes has to be unique to the process. To ensure that we made the following adjustment to the computation of coordinates.

```
double yn = ymin + j*hy + rank*ny*hy;
```

## 2.3 Communication Between Processes

We included two halo rows in allocating memory to the $q$ values. This allows us to create a space where the process can store the received value. Inside the solver we use MPI_Send and MPI_Recv to communicate the top and bottom most real row of values to the upper and lower processes respectively. We are also allocating memory for global values for $q^{n-1}$, $q^n$, $x$ and $y$. These memories are to be filled with values coming from all processes to be printed in the rank 0. In order to collect the values from every process and fill in these global values we used MPI_Allgather. Global values are necessary for printing out the values for checking the results while halo rows necessary for calculations.

## 2.4 *applyBC* Function

Since this function is writen in only considering the series case it needed some adjustment. We changed the function such that it only applies boundary condition to the bottom-most row when rank=0 and to the top-most row when rank=size-1. The boundary condition loop for left and right is unchanged.

## 2.5 Solver

Since we implemented the halo rows, the solver isn't ignoring boundary rows correctly. The only rows that are ignored are the halo rows. Hence we are going to reapply the boundary conditions after the second-order finite-difference part to correct this mistake in the first and the "size"th process.

## 2.6 Printing Results

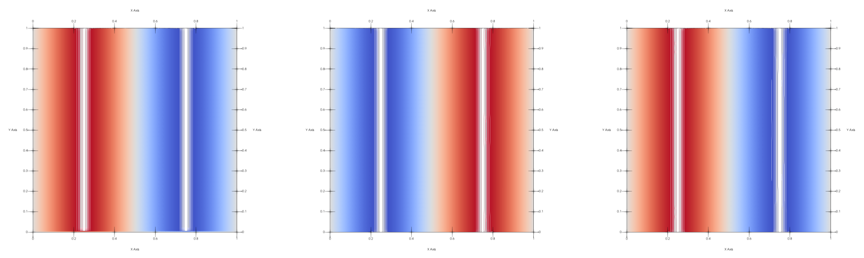
In all of the sections where some form of results is to be printed in either the screen or a file we made sure that only one of the processes print out the results by using the global variables we mentioned before. We also recorded the time elapsed using MPI_Wtime() function for each of the processes. We then use MPI_Reduce to get the one with the maximum value then print it as our wall time elapsed in seconds.

# Chapter 3

# Results and Discussion

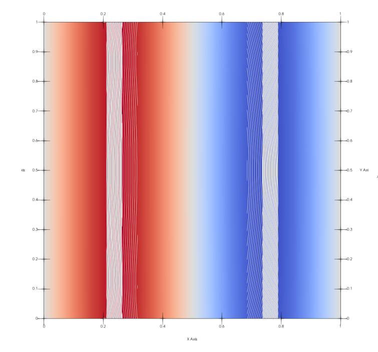
## 3.1 Results

The post processing of the evolution of the solution throughout the time steps and the solutions at the last time step at different resolutions can be seen in Figure 3.1 and Figure 3.2 respectively.

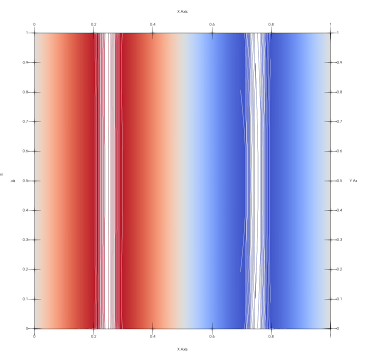


(a) (b) (c)

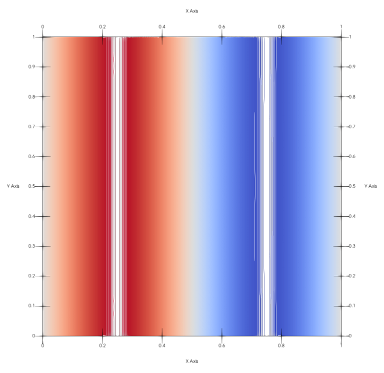
Figure 3.1: Evolution of the solution for 180*180 resolution at (a)0 (b)0.5 and (c)1 seconds



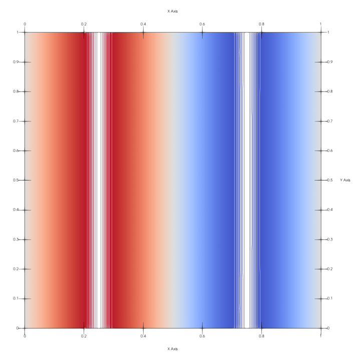
(a)



(b)



(c)



(d)

Figure 3.2: Post process of the solution at the last time step for (a) 20*20 (b)60*60 (c)180*180 and (d)1020*1020 resolutions with same parameters

Firstly, we recorded the time elapsed for different number of MPI processes with different resolutions. Then we plotted the time elapsed versus number of MPI processes for 180*180 resolution to make a strong scaling analysis. We also ran the code multiple times to get an average value for the time elapsed.

| Resolution \ Number of Processes | series | 1(with MPI) | 2 | 4 | 6 | 10 |
|---|---|---|---|---|---|---|
| 20*20 | 0.309022 | 0.419144 | 0.241538 | 0.155123 | 0.200903 | 0.155488 |
| 60*60 | 2.558143 | 2.930173 | 1.622076 | 0.914772 | 1.028260 | 0.780037 |
| 180*180 | 23.115470 | 24.243686 | 12.603723 | 7.178683 | 6.844577 | 5.659869 |

Table 3.1: Elapsed time(s) for different number of processes with different problem sizes.

| Resolution \ Number of Processes | 2 | 4 | 6 | 10 |
|---|---|---|---|---|
| 20*20 | 0.640 | 0.498 | 0.256 | 0.199 |
| 60*60 | 0.789 | 0.699 | 0.415 | 0.328 |
| 180*180 | 0.917 | 0.805 | 0.563 | 0.408 |

Table 3.2: Scaling factors of parallelization with MPI with respect to the serial version of the code

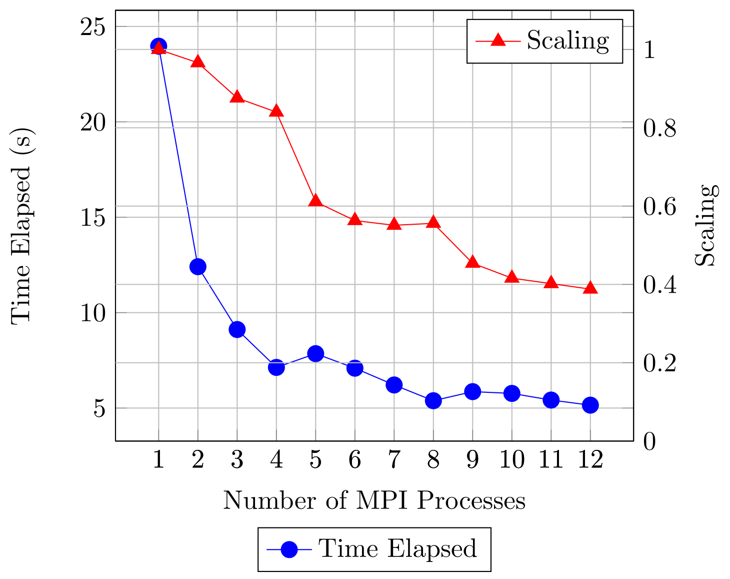


Figure 3.3: Time Elapsed and Scaling vs. Number of MPI Processes for a problem size of approximately 180*180

## 3.2 Discussion

The first thing observed about this study is solely implementing MPI into the code without actually utilizing it results in worse performance than series version which makes sense but it can be seen from the Table 3.1 the difference in series version and 1 process version continues to increase while percent-wise decrease. This can be due to the initializing of MPI and the time it takes depends only on the problem size. In Table 3.1 this statement proves to be correct since the time differences are increasing with multiply of 3 while the problem size in Y direction is also increasing with a multiply of 3.

Secondly it can be seen from the 3.1 that as the problem size gets smaller implementing MPI yields less increase in performance. This is due to the above mentioned situation where MPI takes a certain ammount of time to run depending on the problem size in the Y direction. As the problem size gets bigger and bigger the time taken by the solver increases more rapidly than the time taken by the MPI hence the effect of the time taken by the MPI in the scaling decreases revealing a more accurate scaling of the implementation.

Finally it can be seen from the Figure 3.3 that in MPI implementation time elapsed decreases as the number of processes increases up to 5 processes. At the 5 process we observe a jump in the behaviour of the time elapsed and this point also points out the dip of the scaling factor too. It can be said that as long as the scaling is concerned this solver should be run with at most 4 processors. On the contrary, if the time is an issue, the solver can be run with 12 or more processors.