ME 489
Homework 4 - MPI Implementation of Explicit In-time One-Dimensional Finite Difference Solver

Ata Bora Çakmakcı and Mustafa Yiğit Görgün

Middle East Technical University

17 December 2023

Contents

Chapter 1

Introduction

In this study we implement explicit in-time, one-dimensional, finite difference solver using MPI. You can see the equation solved by the Finite Difference Solver below.

$$\frac{\partial q}{\partial t} = \nabla \cdot (k \nabla q) + f(x,t)$$

We approximated the right side of the equation as using central differencing method at node i as

$$\nabla \cdot (k \nabla q) + f(x,t) \approx \text{rhs(q, t)} = \frac{k}{dx^2}(q[i-1] - 2q[i] + q[i+1]) + f(x[i],t)\Big)$$

We modeled the equation as a one dimensional data with boundary conditions on the extreme left and right data points. We first generated the x-coordinates with uniform spacing. Then solved for all the nodes in the current process. Finally we implemented point-to-point communication between each process of consecutive ranks. After the complete MPI implementation we additionally implemented the OpenMP on top of the MPI implemented solver. During these tests we used the Intel i7-10750h processor

Chapter 2

Implementation

2.1 Solver and MPI Implementations

2.1.1 Finding x-coordinates for uniform spacing:

In order to find the uniform spacing both in and between nodes we utilized the rank and the node number as shown below:

```
x[i]=rank*n*dx/size+dx*(i-1);
```

This allows us to generate uniformly spaced x-coordinates for both single and multi-process. It can be seen that `x[n+1]` and `x[0]` do not make sense in the first look. This is due to these nodes being ghost nodes and the values of these nodes do not directly contribute to the computations. These nodes are simply for communicating between nodes and they only hold the values send by the consecutive nodes to be used in the computations. This explanation is also true for the computation of q values coming up next.

2.1.2 rhs(q,t) computations for the next time step

In order to find the values of rhs(q,t) for the next time step we implemented the following computation:

```
double rhsqt;

rhsqt=k/pow(dx,2)*(q[i-1]-2*q[i]+q[i+1]+source(x[i],time));

qn[i]=q[i]+dt*rhsqt;
```

The new q values are stored in the qn in order to be the q values in the next time step. The q values computed in the first and nth indices are using the values sent by the MPI from the consecutive ranked processes stored in the zeroth and (n+1)th indices respectively.

2.1.3 Point-to-point communication using MPI:

In our model the processes are ordered with respective to their ranks as it goes from the "left-most" process, which is the process with the rank of 0, to the "right-most" process, which is the process with the rank of (size-1). Firstly the right-most *real* nodes are communicated to the left-most *ghost* nodes of the process in the right of the current one. It can be seen that there is not a process that the (size)th process can communicate hence this process jumps directly to the left communication. Secondly the left-most *real* nodes are communicated to the right-most *ghost* nodes of the process in the left of the current one. It can be also seen that the first process does not have any nodes to communicate in the left. After the communication is done the q calculations are done for the next time step explained above this section. Lastly the right most ghost node of the last process and the left most ghost node of the first process is recalculated according to the boundary conditions.

2.1.4 Extra: Multi-process output for confirming results:

In order to obtain the output for confirming the communication we implemented global variables for both x and q then stored the q and x values of each process inside of these values using the `MPI_Allgather` function of MPI. This function gathers the data given by each of the processes and orders the block of data with respect to the rank of each process. This allows us to view the data in a single output file. We also changed the single process file output to output files for each time step instead of one output file with row for each time step. This allowed us to view the result as an animation in ParaView.

2.2 OpenMP Implementations

We implemented OpenMP on both the calculation of the q in the next time step loop and updating q loop. A simple implementation of the following code was enough for both for loops:

```
#pragma omp parallel for
```

We also set the number of threads via a globally defined variable named `THREAD_NUM` and set the number of threads with `omp_set_num_threads(THREAD_NUM)`.

Chapter 3

Results and Discussion

3.1 Results

We first 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 each of the resolution tested to make a strong scaling analysis. To run the code with the intended resolution we divided the resolution(number of nodes) to number of MPI Processes. We also run the code multiple times to get an average value for the time elapsed.

Secondly we ran the OpenMP implemented code with different number of MPI processes with different number of threads for a fixed resolution. Then we plotted the time elapsed versus number of OpenMP threads for a fixed number of MPI processes for each of the different resolutions.

Number of Processes / Resolution	1	2	4	6
60	0.000130	0.000115	0.000114	0.000131
600	0.103402	0.055802	0.033349	0.044876
6000	105.723598	52.990292	30.032296	40.169219

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

Number of Processes / Resolution	2	4	6
60	0.5652	0.285	0.16539
600	0.9265	0.7751	0.576
6000	0.9975	0.88	0.4386

Table 3.2: Scaling factors for MPI only experiments in Table 3.1

Number of Processes / Number of Threads	1	2	4	6
4	0.000389	0.001895	1.439126	4.686524
8	0.000662	1.643784	8.635415	14.061206

Table 3.3: Elapsed time(s) for different number of OpenMP threads with different number of MPI processes for a fixed problem size of 60

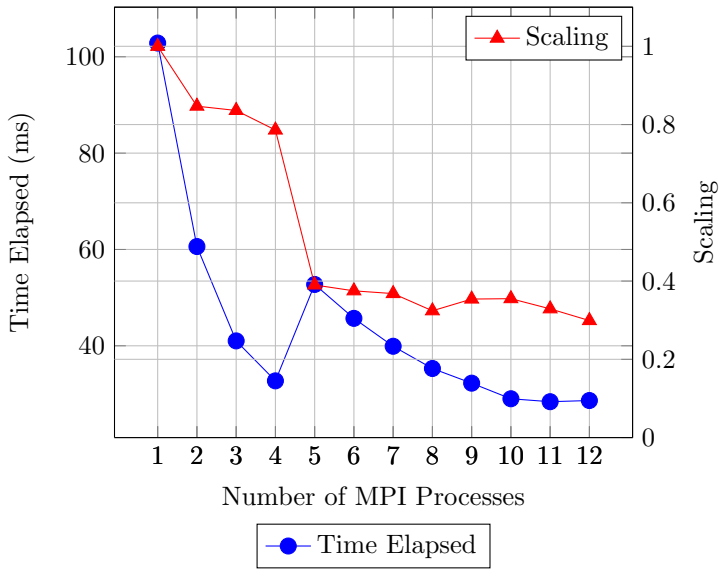

Figure 3.1: Time Elapsed and Scaling vs. Number of MPI Processes for a problem size of approximately 600

3.2 Discussion

It can be seen that in MPI implementation only 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 it is wise to use 4 MPI processes at most to run this code. Another thing observed is that the co-use of OpenMP and MPI does not result in an increase in efficiency, moreover implementing in this way significantly reduces the efficiency of the MPI code. The final thing observed from this experiment is that if the problem size is small enough the use of MPI does not seem to affect the time elapsed in an observable manner. The use of MPI is not always a method that decreases time elapsed and its benefit is also determined by the problem size.