Distributed memory parallelization of Lax-Wendroff Flux Reconstruction

TIFR-CAM,
Bangalore, India.

Report by
Devansh Tripathi
IVR No. IMS22090
IISER Thiruvananthapuram,
Kerala, India

Period: June-July, 2024

Acknowledgement

I would like to thank Prof. Praveen Chandrashekar for providing me the oppportunity to work under his supervision for the summer of 2024. This project has provided me with a lot of experiences and learnings that will be very useful in shaping my career. It has helped me in curating my interests and will greatly influence further decision making of my career. I also had a great pleasure of working with Dr. Arpit Babbar for this project. I appreciate his supporting nature and his knowledge on the topic which is also a part of my learnings this summer.

I would also like to acknowledge the interdisciplinary academic environment of TIFR-CAM and thank all the people at TIFR-CAM for making my internship experience valuable.

Distributed memory parallelization of Lax-Wendroff Flux Reconstruction

Author
Devansh Tripathi
IISER Thiruvananthapuram,
Kerala, India

Supervisor
Prof. Praveen Chandrashekar
TIFR-CAM,
Bangalore, India

Aim of the project

The real world problems requires massive amount of data generation and needs compute power of more than a few computing units such as CPU/GPUs. The aim of this project is to tackle the issue of usage of distributed memory on multiple nodes and to use the computing power of CPUs present on more than a node in an multi-node environment for an adaptive high-order numerical simulation framework for hyperbolic conservation laws — `TrixiLW.jl`

In order to extend `TrixiLW.jl` to support distributed memory parallelization — Message Passing Interface (MPI) implementation is used. MPI is an open source implementation for parallelization developed and maintained by MPI Forum. Optimization of MPI code have also been performed in order to get good speed-up and higher efficiency as demonstrated by the results of scaling tests.

Contents

Chapter 1

Introduction

Lax-Wendroff Flux Reconstruction (LWFR) is a high order, explicit spectral element method for solving systems of partial differential equations in conservative form.

$$\boldsymbol{u}_t + \nabla_x \cdot \boldsymbol{f}^a(\boldsymbol{u}) - \nabla_x \cdot \boldsymbol{f}^v(\boldsymbol{u}, \nabla \boldsymbol{u}) = \boldsymbol{S}(\boldsymbol{u}) \tag{1.1}$$

An example of above system is the compressible Navier-Stokes equations. Traditional spectral element methods for solving these equations are the Runge-Kutta Flux Reconstruction (RKFR) schemes which perform a spatial discretization (i.e., in x) and then use a **multi-stage** Runge-Kutta ODE solver to perform time discretization (i.e., in t). The Runge-Kutta method requires multiple inter-element communication for each of its stages which can become the bottleneck, especially in a parallel code.

Lax-Wendroff Flux Reconstruction (LWFR) schemes are an alternative to this as they perform the evolution in a single stage, thus decreasing the inter-element communication. The main goal while writing parallel algorithm should be to minimize the data to be communicated without creating redundant data in each process's memory.

Using parallel algorithms has its own advantages and disadvantages but the gain that we acquire is far more than the loss for data and compute intensive problems.

1.1 Advantages of parallelization

- Execution of code on many nodes each with its own computing units enables to use extra compute.
- Execution of memory intensive programs on a multi-node environment enables it to use memeory available on each node.
- Using multiple cores on many nodes can speed up the process and can save time.

1.2 Challenges of parallelization

- Execution of code in an multi-node environment requires a lot of resources such as computing units, interconnects, networking hardwares etc.

- Data communication between processes puts a additional overhead to the execution time which can be significant if code is not written efficiently.
- It increases the complexity of the code hence it is difficult to develop, modify and maintain the parallel code.

In chapter 2, we will discuss the theoritical framework for LWFR method. I have also included finite volume method (FVM) as background for the LWFR as knowing about FVM can be very helpful to comprehend flux reconstruction method.

In chapter 4, I have explained the methodology of how parallelization of `TrixiLW.jl` has been done. I also have mentioned about the data structures that has been used for parallelization. Along with the methodology that we have followed, I have also provided a alternate way of parallelization using the newly released MPI Remote memory access (RMA) in chapter 5

In chapter 6, I have performed analysis of results in terms of speed and accuracy, and provided a possible interpretation of results.

In later chapters 7 and 8, the conclusion and the results that we got from the code along with results of scaling tests are shown.

Chapter 2

Theoretical framework

2.1 Finite Difference Method

In this section the finite difference method for solving partial differential equations (PDEs) will be discussed. A finite difference method consists of the following steps:

1. Discretization of the domain on which the equation is defined.
2. For each grid point, replace the derivatives with an approximation, using the values in neighbouring grid points.
3. Solve the resulting system of equations.

There are set of approximations that are used to approximate the derivative. These includes-
For first derivative f':

$$f'(x) \approx \begin{cases} \frac{f(x+h)-f(x)}{h}, & \text{Forward difference} \\ \frac{f(x)-f(x-h)}{h}, & \text{Backward difference} \\ \frac{f(x+h)-f(x-h)}{2h}, & \text{Central difference.} \end{cases} \tag{2.1}$$

The commonly used approximation to second derivative $f''(x)$ is-

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{2.2}$$

Now, we will see the approach to numerically solve a time-dependent PDE using finite difference methods.

2.1.1 Semi-discretization

This technique is used to discretize the time dependent PDEs. The word *semi* is to describe the notion that the PDE will first be discretize in space using finite difference methods and then some time marching schemes such as Runge-Kutta methods, Lax-Wendroff method etc., will be used to move forward in time. We will show this for a linear advection equation in 1D mesh

$$u_t + f(u)_x = 0 \tag{2.3}$$

In order to discretize the second derivaitve in above equation, we will use central differences-

$$\frac{\partial u}{\partial t} = -\frac{f(x+h) - f(x-h)}{2h} \tag{2.4}$$

The eq (2.4) is called semi-discretize form of the linear advection equation. In the next section, we will be using classical Lax-Wendroff method as our time marching scheme.

2.1.2 Classical Lax-Wendroff method

Lax-Wendroff schemes are used for numerically solving conservation law or a system of hyperbolic conservation laws. These methods belongs to the class of conservative schemes and can be derived in various ways. For simplicity, we will derive the method by using model given in equation (2.3), namely the linear advection equation with $f(u) = au$, where a is a constant propagation velocity. We can use Taylor expansion of u_j^{n+1}

$$u_j^{n+1} = u_j^n + \Delta t \frac{\partial u}{\partial t}\bigg|_j^n + \frac{(\Delta t)^2}{2}\frac{\partial^2 u}{\partial t^2}\bigg|_j^n + O(\Delta t^3)$$

It is a secong order accurate method since we are retaining terms till second order. From the equation (2.3), we get by differentiation

$$\frac{\partial u}{\partial t}\bigg|_j^n = -a\frac{\partial u}{\partial x}\bigg|_j^n \qquad \text{and} \qquad \frac{\partial^2 u}{\partial t^2}\bigg|_j^n = -a^2\frac{\partial^2 u}{\partial x^2}\bigg|_j^n$$

We can use central difference from equation(2.1) and from equation(2.2) and the final result will be

$$u_j^{n+1} = u_j^n - a\Delta t\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) - a\frac{(\Delta t)^2}{2}\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right)$$

$$u_j^{n+1} = u_j^n - \frac{a}{2}\frac{\Delta t}{\Delta x}\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) - \frac{a}{2}\left(\frac{\Delta t}{\Delta x}\right)^2\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right)$$

This is the update step for Lax-Wendroff. Later, we will see in section 2.2.3, how to calculate Δt using CFL condition which is a needed but not a sufficient condition for stability of the scheme. The `Julia` implementation of this scheme is shown in the section 5.3 of chapter 5.

2.2 Finite Volume Methods

2.2.1 Motivation for Finite Volume Method

In finite difference methods, the derivatives are approximated by finite differences – esssentially using Taylor expansion. A large discussion on finite difference methods shows that they need the solution to be smooth and equation to be satisfied point-wise. However, the solutions to the scalar conservation law (2.3) are not necessarily smooth, so the Taylor expansion – replacing derivatives using finite differences – is no longer valid. Hence, we need a new framework for designing numerical methods for scalar conservation laws.

This section introduces the finite volume method for the numerical solution of hyperbolic PDE and system of hyperbolic PDEs. We will talk about fundamental concept of this method and also discuss a type of numerical flux.

2.2.2 Formulation for conservation laws

The basic difference between finite difference methods and finite volume methods is that finite volume methods are derived on the basis of integral form of the conservation laws, instead of the differential form as in finite differences.

For the case of one spatial dimension as in eq (2.3) the domain is divided into finite volumes (or *grid cells/intervals*) and we need to monitor the approximation to the integral of u over each of these volumes. For each time step, we need to update u using the approximation to the flux $f(u)$ that comes in and goes out from the boundary of these volumes.

If we denote i th volume by:

$$C_i = (x_{i-1/2}, x_{i+1/2}) \tag{2.5}$$

The value U_i^n will approximate the average value of u over the ith interval at time t^n

$$U_i^n \approx \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} u(x, t_n) dx \tag{2.6}$$

where $\Delta x = x_(i + 1/2) - x_(i - 1/2)$ is the length of the cell. Here we are assuming a uniform grid for easeness of explanantion.
$\sum_{i=1}^{N} U_i^n \Delta x$ approximates the integral of u over the entire $1D$ domain and if we use the method that is in conservative form, the value of $\sum_{i=1}^{N} U_i^n \Delta x$ will only change due to the fluxes at the boundary of the domain.
Now, we will see how can we develop an explicit time-marching algorithm using integral form of the conservation law. We can get integral form by integrating (2.3) on interval C_i. Steps have been shown below:

$$\int_{C_i} \frac{d}{dt} u(x, t_n) dx + \int_{C_i} \frac{d}{dx} f(u(x, t_n)) = 0$$

$$\frac{d}{dt} \int_{C_i} u(x, t_n) dx = f(u(x_{i-1/2}, t_n)) - f(u(x_{i+1/2}, t_n))$$

For given U_i^n, the cell averages at time t_n, we can approximate the cell averages at time t_{n+1}, U_i^{n+1}, after a time step of length $\Delta t = t_{n+1} - t_n$.
Integrating the above equation in time from t_n to t_{n+1}:

$$\int_{t_n}^{t_{n+1}} \frac{d}{dt} \int_{C_i} u(x, t_n) dx = \int_{t_n}^{t_{n+1}} f(u(x_{i-1/2}, t_n)) - f(u(x_{i+1/2}, t_n))$$

$$\int_{C_i} u(x, t_{n+1}) dx - \int_{C_i} u(x, t_n) dx = \int_{t_n}^{t_{n+1}} f(u(x_{i-1/2}, t_n)) dt - \int_{t_n}^{t_{n+1}} f(u(x_{i+1/2}, t_n)) dt$$

Rearranging this equation and dividing both sides by Δx yields:

$$\begin{aligned} \frac{1}{\Delta x} \int_{C_i} u(x, t_{n+1}) dx = & \frac{1}{\Delta x} \int_{C_i} u(x, t_n) dx \\ & - \frac{1}{\Delta x} \left[\int_{t_n}^{t_{n+1}} f(u(x_{i-1/2}, t_n)) dt - \int_{t_n}^{t_{n+1}} f(u(x_{i+1/2}, t_n)) dt \right] \end{aligned} \tag{2.7}$$

In (2.7), we can not evaluate the time integrals on the right hand side exactly, as $u(x_{i\pm1/2})$ varies with time along each edge of the cell. We can rewrite eq (2.7) as:

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1/2}^n - F_{i-1/2}^n\right) \tag{2.8}$$

where $F_{i-1/2}^n$ is the *average* flux along $x = x_{i-1/2}$:

$$F_{i-1/2}^n = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f(u(x_{i-1/2}, t_n))dt$$

If we can approximate this *average flux* function based on the values of U^n, then we will have a discrete method. And that approximation to *average flux* function is called *numerical flux function*.

2.2.3 The CFL Condition

The *numerical flux* function as the approximation of *average flux* is the main ingredient in a finite volume scheme. We need a approximation to:

$$\bar{F}_{j+1/2}^n \approx F_{i-1/2}^n = \frac{1}{\Delta t}\int_{t_n}^{t^{n+1}} f(u(x_{i-1/2}, t_n))dt \tag{2.9}$$

at each interface $x_{i-1/2}$. As the cell averages U_j^n are constant in each cell C_j, at each time level, we can define at each cell interface $x_{i-1/2}$ a, *Riemann problem*:

$$\begin{cases} u_t + f(u)_x = 0 \\ u(x, t_n) = \begin{cases} u_j^n & \text{if } x < x_{j-1/2} \\ u_{j+1}^n & \text{if } x > x_{j-1/2} \end{cases} \end{cases} \tag{2.10}$$

At every time level, the cell averages define a superposition of *Riemann* problems of the form (2.10), at each interface. The solution to the *Riemann* problem consists of shock waves, rarefraction and compound waves. Moreover, waves from neighboring *Riemann* problems can intersect after some time hence imposing the CFL condition is a requirement. We define *Courant number* as:

$$v = \frac{\Delta t}{\Delta x}\max_p \mid \lambda^p \mid \tag{2.11}$$

where λ's are the *eigenvalues* of a matrix A in the case of system of linear hyperbolic equations and λ can be the wave propogation speed in the case of a single hyperbolic PDE.

For a three point stencil the CFL condition leads to $v \leq 1$. It can be noted that for a wider stencils, the CFL condition can give $v \leq 2$. The CFL condition is a *necessary* but not a sufficient condition for stability of the solution.

2.2.4 Rusanov's flux

Rusanov scheme [2] is also called *local Lax-Friedrich scheme* [1] because it can be obtained from a simple modification in *Lax-Friedrich scheme* basically using different approach in choosing the parameter λ.

A Note about Lax-Friedrich scheme
Lax-Friedrich scheme has this structure:

$$f_{j+1/2} = \frac{1}{2}\left(f_j + f_{j+1}\right) - \frac{1}{2}\lambda\left(u_{j+1} - u_j\right) \qquad \lambda = \frac{\Delta x}{\Delta t} \tag{2.12}$$

The time step must satisfy the CFL condition 2.2.3, we can write time step Δt as:

$$\Delta t = \mathrm{CFL}\frac{\Delta x}{\max\limits_j \sigma(f'(u_j^n))}, \quad CFL \leq 1$$

where σ is the *spectral radius*. We see that

$$\lambda \approx \max_j \sigma(f'(u_j^n))$$

The parameter λ is related to the maximum wave speed in the whole computational domain. The only problem with this scheme is this the results will be very *diffusive* and shocks are smeared to a considerable extent.

In Rusanov's scheme, instead of global maximum we use local estimate of λ and hence the flux has the form

$$f_{j+1/2} = \frac{1}{2}(f_j + f_{j+1}) - \frac{1}{2}\lambda_{j+1/2}(u_{j+1} - u_j) \tag{2.13}$$

where $\lambda_{j+1/2}$ is an estimate of the maximum wave speed arising in the Riemann problem at the face $j + 1/2$. For Euler equations

$$\lambda_{j+1/2} = \max\{|u_j| + a_j, |u_{j+1}| + a_{j+1}\}$$

is a good choice.

2.3 Flux Reconstruction

The flux reconstruction (FR) method is a class of discontinuous Spectral Element Method for the discretization of conservation laws. FR method utilizes a nodal basis which is usually based on some solution points like Gauss point etc., to approximate the solution with piecewise polynomials inside the element. The main idea is to construct a continuous approximation of flux utilizing a numerical flux (discontinuous) at the cell interfaces and a *correction function*. The solution at the nodes is then updated by a collocation scheme in combination with a Runge-Kutta method.

The choice of the correction function affects the accuracy and stability of the method. FR method can be shown to be equivalent to some discontinuous Galerkin and spectral difference schemes, using a proper choice of solution points and correction function, as shown in [3, 4]. The quadrature-free nature of FR method together with the ability to cast the operations as matrix-vector operations makes these methods to be performed efficiently on modern vector processors [5].

At some time t, we have the piecewise polynomial solution defined; the FR scheme can be explained by the following steps:

Considering scalar form of conservation law $u_t + f(u)_x = 0$
Step 1. In each element, we contruct a flux approximation by interpolating the flux at the solution points leading to a polynomial of degree N, given by

$$f_h^\delta(\xi, t) = \sum_0^N f(u_j^e(t)) l_j(\xi)$$

where $\xi \in [0, 1]$ is a point in reference element $[0, 1]$ and l_j is Lagrange polynomial of degree N. The above flux is discontinuous flux across the interfaces of the elements.

Step 2. Continuous flux approximation is build by adding correction terms at the element boundaries

$$f_h(\xi, t) = \left[f_{e-\frac{1}{2}}(t) - f_h^\delta(0, t)\right] g_L(\xi) + f_h^\delta(\xi, t) + \left[f_{e+\frac{1}{2}}(t) - f_h^\delta(1, t)\right] g_R(\xi)$$

where g_L, g_R are correction functions at left and right boundary respectively, and

$$f_{e+1/2} = f(u_h(x^-_{e+\frac{1}{2}}, t), u_h(x^+_{e+\frac{1}{2}}, t))$$

is a numerical flux that makes the flux unique across the cells.

Step 3. Then we obtain the system of ODE by collocating the PDE at the solution points

$$\frac{du_j^e}{dt}(t) = -\frac{1}{\Delta x_e}\frac{\partial f_h}{\partial \xi}(\xi_j, t), \qquad 0 \le j \le N$$

which can be solved in time by schemes such as Runge-Kutta.

Correction functions. The correction functions g_L and g_R at the left and right boundary respectively, should satisfy few conditions in order to get a working scheme.
1. The end point conditions:

$$\begin{aligned} g_L(0) &= 1, \qquad g_R(0) = 0 \\ g_L(1) &= 0, \qquad g_R(1) = 1 \end{aligned}$$

which insure the continuity of the flux at interfaces.
2. They should **approximate zero** in some sense. This means that they should not add or subtract anything in flux inside the element. Two correction functions Radau and g2 are of major interest as they cooresponds to commonly used DG formulations.

2.4 Lax-Wendroff Flux Reconstruction

Lax-wendroff method is a numerical method for solving system of hyperbolic conservation laws. This method can be derived using Taylor's expansion. We will also use the PDE to rewrite some of the time derivatives in the expansion as spatial derivaitves. Using Taylor's expansion in time around $t = t_n$, we can write the solution at next time level as

$$u^{n+1} = u^n + \sum_{m=1}^{N+1} \frac{\Delta t^m}{m!} \partial_t^m u^n + O(\Delta t^{N+2}) \tag{2.14}$$

We are retaining terms up to $O(\Delta t^{N+1})$ so the overall accuracy, in both space and time, is of the order $N+1$. The expected spatial error is expected to be of $O(\Delta x^{N+1})$. Using the PDE, $\partial_t u = -\partial_x f$, we can re-write the time derivatives in terms of spatial derivatives

$$\partial_t^m u = -\partial_t^{m-1}\partial_x f = -(\partial_t^{m-1} f)_x, \quad m = 1, 2 \ldots$$

Now eq(2.14) becomes

$$\begin{aligned} u^{n+1} &= u^n - \sum_{m=1}^{N+1} \frac{\Delta t^m}{m!}(\partial_t^{m-1} f)_x + O(\Delta t^{N+2}) \\ &= u^n - \Delta t \left[\sum_{m=0}^{N} \frac{\Delta t^m}{(m+1)!} \partial_t^m f \right]_x + O(\Delta t^{N+2}) \\ &= u^n - \Delta t \frac{\partial F}{\partial x}(u^n) + O(\Delta t^{N+2}) \end{aligned} \tag{2.15}$$

where

$$F(u) = \sum_{m=0}^{N} \frac{\Delta t^m}{(m+1)!} \partial_t^m f(u)$$

is the approximation to the time average flux as it can be written as average of truncated Taylor's expansion of the flux f in time.

As the step described in the previous section, we will first reconstruct the time average flux F inside each element by a continous polynomial $F_h(\xi)$ using discontinous flux and correction functions. Then truncating the eq(2.15), the solution at the nodes is update by a collocation scheme as follows

$$(u_j^e)^{n+1} = (u_j^e)^n - \frac{\Delta t}{\Delta x_e} \frac{dF_h}{d\xi}(\xi_j), \qquad 0 \le j \le N \tag{2.16}$$

Above is the single step Lax-Wendroff update scheme for any order of accuracy.

Chapter 3

Profile of Organisation

Name of Organisation: Centre for Applicable Mathematics
Affiliation: Tata Institute of Fundamental Research, Mumbai
Website: https://www.math.tifrbng.res.in/

About: The Centre for Applicable Mathematics (CAM) is a premier research centre for mathematics and is part of the Tata Institute of Fundamental Research. Research at CAM is focused on mathematical analysis, theoretical and computational analysis of PDEs and their applications, and probability theory, complex analysis and related areas. CAM runs graduate student programmes that lead to a PhD in mathematics. There are several opportunities to visit and work at CAM (postdoctoral fellowships, short term visiting positions, summer programmes, etc). Members of CAM are actively engaged in outreach activities as well — ranging from working with gifted high school students to training researchers in the latest advancements at the frontiers of their disciplines.

Mail Address: Tata Institute of Fundamental Research, Centre For Applicable Mathematics, Post Bag No 6503, GKVK Post Office, Sharada Nagar, Chikkabommsandra, Bangalore 560065, Karnataka, India

Email: math [at] math [dot] tifrbng [dot] res [dot] in

Chapter 4

Methodology

4.1 Simulation of Conservation laws

4.1.1 Trixi.jl

`Trixi.jl` [6] is a numerical simulation framework for conservation laws written in `Julia`. It is an open source package hosted on Github

4.1.2 Features of `Trixi.jl`

`Trixi.jl` package can be used for simulation of conservation laws in 1D, 2D and 3D and have the following features:

- Support cartesian and curvilinear meshes.
- Structured and Unstructured meshes.
- AMR and Shock capturing.
- High-order accuracy in space and time.
- Discontinuous Galerkin methods.
- CFL-based and error-based time step control.
- Periodic and weakly-enforced boundary conditions.

`Trixi.jl` has multiple governing equations such as Compressible Euler, Compressible Navier-Stokes, Shallow water equations etc. The code written in this package also have support for shared memory parallelization via multi-threading and multi-node parallelization via `Message Passing Interface (MPI)`. It also provides visualization and post-processing tools for the results.

It uses Runge-Kutta methods for discretization in time with other high-order methods for space discretization. RK methods are multi-stage methods for solving system of ODEs that we get after collocating the PDE at solution points, at the last step as described in the section of previous chapter 2.3.

4.2 TrixiLW.jl

The one shortcoming of RK methods is their multi-stage nature. As we know, we need as many stages as the order of the method and sometimes number of stages has to be more than order of method to get the required accuracy in the case of high order methods. This can create a bottleneck in a parallel code as we need to communicate data at every stage and can significantly affect the performance of the code.

In `TrixiLW`, we have extended the `Trixi` package to use Lax-Wendroff method for time discretization instead of RK methods. LW method is a single step update method. The features of `TrixiLW.jl` are as follows:

4.2.1 Features of TrixiLW

`TrixiLW` can be used for conservation laws in 2D with the following features:

- Support for cartesian and curvilinear meshes.
- Structured and Unstructured meshes.
- High-order accuracy in space and time.
- Discontinuous Galerkin methods.
- CFL-based and error-based time step control.
- Periodic and non-periodic boundary conditions.
- AMR and Shock capturing

This package also supports shared memory parallelization via multi-threading. As a part of this summer project, I have added support for multi-node parallelization via `Message Passing Interface (MPI)`. For visualization and post-processing, we use the tools of `Trixi` itself.

4.2.2 Multi-node parallelization of TrixiLW

Multi-node parallelization implies that we can use the computing units of various nodes connected to each other via fast interconnects. This can help us encounter the issue of insufficient memory on a single node and harness the computing power of many nodes. We can use memory of other nodes which are connected to each other via channel-based fabric such as `InfiniBand` that facilitates high-speed communication between interconnected nodes.

Although, using multi-node parallelization can also creates a communication overhead for the running program yet it is used since there is a nice trade-off between the overhead created due to communication and other aspects such as memory, computing power, data storage etc.

Message Passing Interface (MPI) is an open source implementation that laid down the rules of communication between processes. It is developed and maintained by MPI Forum. There are many open source libraries that are based on MPI and prvoide language specific routines. These includes `OpenMPI, MPICH` etc.

Strategy of parallelization

`TrixiLW.jl` is based on Lax-Wendroff Flux reconstruction (LWFR) method. In this method, we need to communicate u (solution values), U (time average solution) and F (time average flux) values of the mpi_interface elements to other required ranks.

Figure 4.1: interfaces and mpi_interfaces

For simplicity we are taking example of 1D mesh as in figure(4.1), the red lines are `mpi_interfaces` which are not owned by the current rank and grey lines are `interfaces` which are owned by the current rank. The values need to be communicated only at the `mpi_interfaces` and not at the normal `interfaces` as both the elements around normal `interfaces` are owned by the current rank. We have used non-blocking communication routines for communication between ranks. These routines include: `MPI_Isend` and `MPI_Irecv`.

4.2.3 Data Structures

At each interface, as we have seen, three variables needs to be stored u, U and F, following is the `MPIInterfaceContainer` data structure that will be used to store the data.

```
# Container data structure (structure-of-arrays style)
# for DG MPI interfaces
mutable struct MPIInterfaceContainerLW2D{uEltype <: Real}
                                        <:AbstractContainer
    u::Array{uEltype, 4}     # [leftright, variables, i, interfaces]
    U::Array{uEltype, 4}
    F::Array{uEltype, 4}
    local_neighbor_ids::Vector{Int} # [interfaces]
    orientations::Vector{Int}       # [interfaces]
    remote_sides::Vector{Int}       # [interfaces]
    # internal `resize!`able storage
    _u::Vector{uEltype}
    _U::Vector{uEltype}
    _F::Vector{uEltype}
end
```

Listing 1: Struct for mpi_interfaces

The `struct` in listing (1) has some extra fields for such as $\_u$, $\_U$ and $\_F$, these field are needed for internal storage since in `Julia` only 1D arrays can be resized. Therefore, memory will be allocated for these 1D arrays and data will be stored in them. Later, that allocated memory will be re-used using `unsafe_wrap` method of

Julia without making another copy of data hence avoiding unnecessary memory allocation.

Other field such as local_neighbor_ids will store ranks of neighboring interfaces, orientations will store the orientation of the interface (either x or y direction) and remote_sides will contain the information whether mpi_interfaces are in positive direction or negative direction of the axis.

There will be another function called prolong2mpiinterfaces! which will be responsible for prolonging the element data to mpi_interfaces and filling up the above mentioned container. After storing the element data in this container, we will be creating the buffers that will be used for MPI communication between ranks. The MPICache container will look something like this:

```
mutable struct MPICache{uEltype <: Real}
    mpi_neighbor_ranks::Vector{Int}
    mpi_neighbor_interfaces::Vector{Vector{Int}}
    mpi_neighbor_mortars::Vector{Vector{Int}}
    # buffers for data storage
    mpi_send_buffers::Vector{Vector{uEltype}}
    mpi_recv_buffers::Vector{Vector{uEltype}}
    # requests object for non-blocking communication
    mpi_send_requests::Vector{MPI.Request}
    mpi_recv_requests::Vector{MPI.Request}
    n_elements_by_rank::OffsetArray{Int, 1, Array{Int, 1}}
    n_elements_global::Int
    first_element_global_id::Int
end
```

Listing 2: Struct MPICache

mpi_send_buffers and mpi_recv_buffers are a collection of buffers that store data for each rank present in communicator MPI_COMM_WORLD.

4.2.4 Mesh Types: TreeMesh and P4estMesh

In this project, the two types mesh structures used for discretization of domain for conservation laws are TreeMesh and P4estMesh. Each of these meshes have distinct properties and challenges involved in parallelization of code.

TreeMesh is a cartesian, h-non-conforming mesh type used in many parts of of TrixiLW.jl. Often, the support for this mesh type is developed best for serial and parallel code since it was the first mesh type in TrixiLW.jl, and it is available in two spatial dimensions for both serial and parallel code. It is limited to hypercube domain i.e. squares in 2D.

P4estMesh is a unstructured, curvilinear, non-conforming mesh used in TrixiLW.jl. It is available for two spatial dimensions with quadrilateral 2D cells. Parallel code for this mesh have also been developed.

The figure(4.2) shows the two-dimensional unstructured curved mesh with 3 elements and the data that needs to be stored in order to navigate through the mesh. We have extended Trixi.jl to support Lax-Wendroff flux reconstruction

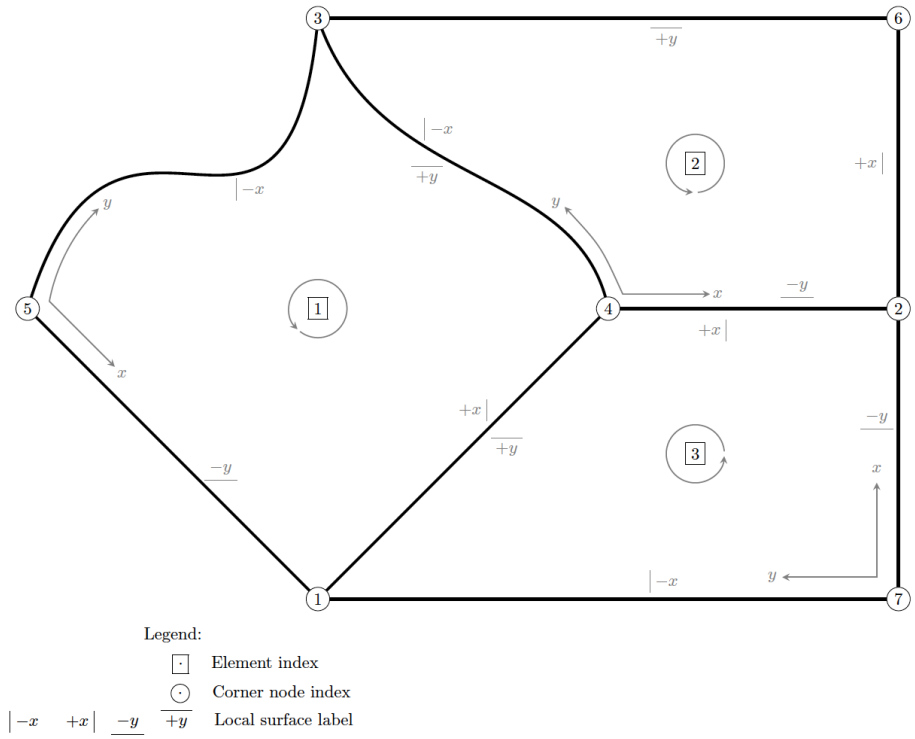


Figure 4.2: P4est Mesh

(LWFR) method hence our main goal was to write code that can re-use most of the functionality provided by `Trixi.jl` and only the code required by LWFR part should be written again.

All the functions that are exclusive to LWFR are dispatched with a extra argument `time_discretization` which is of type `AbstractLWTimeDiscretization` in order to differentiate these functions. In the next section, we will discuss an alternative approach to parallelization than using non-blocking communication routines such as `MPI_Isend` and `MPI_Irecv`.

Chapter 5

Alternative Methodology

5.1 Remote Memory Access: MPI 3.0

In the reign of parallel computing, there exist two major approaches for communicating data between processes: message passing and direct memory access of the remote process. Shared memory approach is also there but that is limited to the procesess on the same node.

MPI-1 provides a way for message passing between processes while MPI-2 gives approach for direct memory access of remote processes and MPI-3 extends it to a great extent. MPI RMA [8] uses operations like *put*, *get* and *accumulate* to *write*, *read* and *reduce* data respectively to a remote process without involving the remote process. One limitation of RMA approach is this it require special hardware support but with increasingly high development in networking caters this need to much extent.

5.1.1 Shared memory vs RMA

MPI remote memory access must be distinguished from shared memory programming model in the following ways:

- MPI RMA approach can be used for communication between cores on the same node as well as cores on different nodes. While shared memory approach is limited to processes on the same core using *threads.*
- In MPI RMA data can be accesses using `MPI_Put` and `MPI_Get` operations while in shared memory we can access data using usual load and store operations as shown in the pictures.
- In shared memory, we have a single address space, shared by multiple threads of execution while in RMA, programmer decides how much memory will be exposed to the remote process for communication.

The major disadvantage of shared memory model is *simultaneous access by different threads to a same memory location.* In order to avoid this race condition, many sophisticated techniques in compilers and special routines are needed such as *locks and mutexes.*

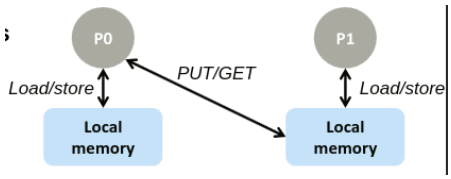


Figure 5.1: RMA[1]

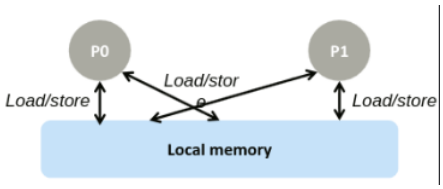


Figure 5.2: Shared Memory[1]

5.2 Introduction to RMA

In traditional message passing, one process *sends* data to other process using `MPI_Send` and other process *receives* data using `MPI_Recv`. Every *send* is compulsory to have a *receive* at the receiver process. This cooperative nature of MPI can impose order or need *tag* matching for delivery of data and that will have performance costs due to overheads.

MPI RMA provides a way of data communication from *origin process* (process that sends data) to the *target process* (process to which data has been sent), without the involvement of *target process*. The calling process specifies both send and receive buffer. Since a single process is involved, these routines are also called *one-sided routines*:

There are three main steps in using RMA:

- **Defining a memory window:**
  Memory *window* is collection of memory locations defined by a group of processes and only those locations can be modified by the remote process using RMA operations. This involves creation of a new MPI object `MPI_Win`. There are 4 window creation routines specified by MPI Forum.

- **Moving the data from *origin* to *target* process:**
  There are several routines to move the data between processes without the involve of the process instead directly writing the data into remote processes memory. These routines include: `MPI_Put, MPI_Get, MPI_Accumulate` etc.

- **Specifying how do we know that data is available to remote process:**
  This is to say that how do we know that receive has been completed? There are several routines such as `MPI_Win_fence, MPI_Win_lock/unlock` etc. that makes sure that data is available to the target process for its local load/store operations.

5.2.1 Memory Window

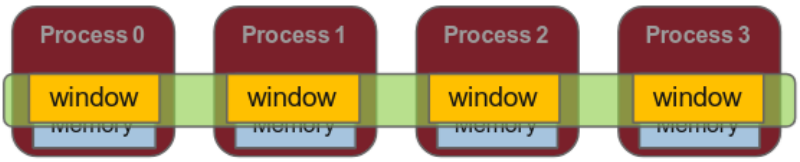


Figure 5.3: Memory window in RMA[1]

It is the memory region of a process that can be accessed by another process using RMA operations. All or a group of processes can create a window or many windows by contributing some part of their memory to the window. It is a contiguous section of memory. There are these 4 window creation routines:

- `MPI_Win_allocate:` In this routine memory allocation and window creation both are handled by MPI implemention.
- `MPI_Win_create:` This routine creates the window from already allocate memory. User have to provide some allocated memory to be create as a window.
- `MPI_Win_allocate_shared:` This routine helps in creating a shared memory window from already allocate memory for the process on the same node hence these process can access each other data with simple load/store and avoid communication completely.
- `MPI_Win_create_dynamic:` It creates a window but does not attach memory directly to it. Memory can be attached later using `MPI_Win_attach` and can be deattached using `MPI_Win_deattach`.

By default, the memory that has been allocated by MPI implemention (if user specifies so) is ttfamily contiguous unless the the `non-contiguous` option is `true.` This can have performance implications as well since `non-contiguous` memory can be allocated aligning to the cache sizes which will decrease the number of cache misses.

After all the communication has been done and window is no longer required, it can be free using `MPI_Win_free` by all the processes that have created the window.

```
1 int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
2                     MPI_nfo info, MPI_Comm comm, MPI_Win *win)
3 int MPI_Win_free(MPI_Win *win)
```

Listing 3: Syntax for C

There are few interesting parameters that are required by window calling routines:

- **disp_unit:** This argument unit should be the multiple of `sizeof()` of simple type such as 'int' etc. that creates up the window object. For example if an array of `Floats` is making up the window then `disp_unit` should be multiple of `sizeof(Float)`. This is the local unit size for displacements, in bytes.
- **info:** This arguemnt of only used for optimization purposes. A value `MPI_INFO_NULL` is always valid.

5.2.2 Moving Data

Since we have now seen how to make memory as a window for RMA opertaions, we need to specify how to move data between process. MPI provides several routines to specify what data to move and to which location. Three of the simplest routines have been described below:

MPI_Put

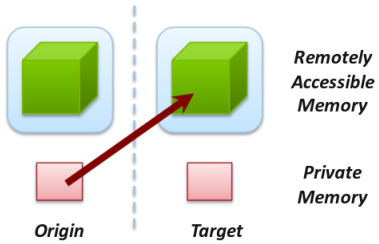


Figure 5.4: MPI_Put[1]

`MPI_Put` is like a "store/write to remote memory" operation. It is a non-blocking communicating routine. This routines writes data from `origin` process's memory called `origin address` to `remote` process's memory at the location as described by `displacement` argument. The data that need to be moved can be anywhere in the `origin` process's memory, it need not to be inside a window.

Programmer need to pay attention while providing the `displacement` argument. **The destination of data is always relative to the memory window exclusive to `remote` process not with the whole window object**.

MPI RMA operations define a separation between moving of data and completion of the operations. Window synchronization routines such as `MPI_Win_fence` should be used after RMA opertions for completion of these operations. Between two `MPI_Win_fence` calls any number of `MPI_Put` operations can be issued. But if a `MPI_Put` and `MPI_Accumulate` operation overlaps (issued to same memory location) between two `MPI_Win_fence` calls then result will be `undefined behaviour.` Along with that, `MPI_Put` and `MPI_Get` both can not be used within two `MPI_Win_fence` calls.

`MPI_Put` is **not** an atomic operation. An **atomic operation** is an operation that can be completed within one CPU cycle and hence these operation can not be interrupted in between by any other process. They execute at lowest level and cannot be broken further.

```
1 int MPI_Put(const void *origin_addr,
2             int origin_count, MPI_Datatype origin_datatype, int
3             target_rank, MPI_Aint target_disp, int target_count,
4             MPI_Datatype target_datatype, MPI_Win win)
```

Listing 4: Syntax for C

MPI_Get

`MPI_Get` routine is used to get data from remote process to the calling process. It can get data from remote process's window to anywhere in the origin (calling) process memory means `origin_addr` need not to be inside window for calling process. This operation is also not an **atomic** operation.
While calling this routine, programmer needs to specify the `displacement` argument relative to the window exclusive to the remote process.

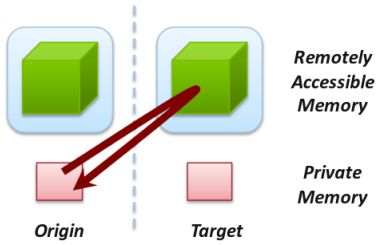


Figure 5.5: MPI_Get[1]

This displacement arguement will specify the location of the data at remote process that needs to be written at the origin process's memory. After using this routine, there should be a call to a window synchronization routine to ensure that the data has been received at the origin process.

```
1 int MPI_Get(void *origin_addr, int origin_count,
2             MPI_Datatype origin_datatype, int target_rank,
3             MPI_Aint target_disp, int target_count,
4             MPI_Datatype target_datatype, MPI_Win win)
```

Listing 5: Syntax for C

MPI_Accumulate

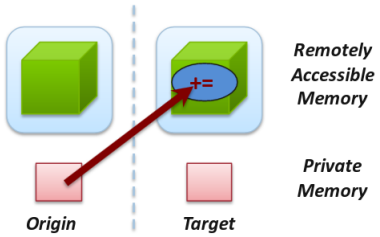


Figure 5.6: MPI_Accumulate[1]

`MPI_Accumulate` routine provides a way to move and combine data at the target process using any of the reduction operations such as `MPI_SUM`, `MPI_MAX` etc. It can be seen similar to `MPI_Reduce` operation but without the involvement of target process. There are few restrictions for using `MPI_Accumulate` as listed below:

- It allows **only predefined operations** such as `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_LAND`, `MPI_LOR` etc.
- It allows only **predefined** MPI datatypes and MPI derived datatypes where all components are of same predefined datatype.

MPI_Accumulate as MPI_Put

Since `MPI_Accumulate` is an atomic operation while `MPI_Put` is not an atomic operation and if we want to use atomic `MPI_Put` we can achieve this using `MPI_Replace`

as an operation of MPI_Accumulate. This will put the value in origin_addr buffer to the target location as provided while calling this routine.

```
int MPI_Accumulate(const void *origin_addr, int origin_count,
                MPI_Datatype origin_datatype, int target_rank,
                MPI_Aint target_disp, int target_count,
                MPI_Datatype target_datatype, MPI_Op op,
                MPI_Win win)
```

Listing 6: Syntax for C

5.2.3 Synchronization Routines

Synchronization routines are a way to tell the process that data is available for local load/store operations from RMA operations as well as data is available for RMA operations from local load/store operations. They need to be called before as well as after RMA operations. The reason for calling them before RMA operations is to make sure that any local load/store operation that had been performed on the window object has been successfully completed and that data can be used in RMA operations. This can be omitted if there are no local load/store operations on the window before RMA operations. Later calling them will make sure the completion of RMA opertions itself.

There are two types of target synchronization in MPI RMA:

1. Active Target synchronization.
2. Passive Target synchronization.

Active Target synchronization

Active target synchronization involves both the origin and the target processes in synchronization using MPI_Win_fence. Any MPI RMA calls need to be wrapped by MPI_Win_fence call.

MPI_Win_fence

MPI_Win_fence is a collective call to all processes in the group associated with the window object that has been passed to MPI_Win_fence. It makes sure that any local stores to the memory window will be visible to RMA operations before any RMA operation takes place. In any pair of successive MPI_Win_fence there may either be any local stores to the local memory window or MPI_Put/Accumulate calls but not both.If there no MPI_Put operation in between two MPI_Win_fence call then there may be both load and MPI_Get operations on the memory window.

assert argument can be used to indicate exactly what kind of operation MPI_Win_fence is synchronizing. Typically, it is used to separate MPI_Put/Accumulate operations from local load and store operation.

```
int MPI_Win_fence(int assert, MPI_Win win)
```

Passive Target synchronization

In this synchronization, the target process does not need to call any synchronization routines such as `MPI_Win_fence`. Before starting any RMA call, there should be a `MPI_Win_lock` and after those calls there should be a `MPI_Win_unlock` but only on the origin proce. This means that all the RMA calls will get completed after `MPI_Win_unlock` returns.

`MPI_Win_unlock` ensures that the RMA operations have been completed successfully and all the data transfers are visible to the target memory esssentially all the synchronization is done by `MPI_Win_unlock`. There is another version of lock that locks the window on all the processes called `MPI_Win_lock_all`. `MPI_Win_unlock_all` is used to unlock the window then. But there is some restrictions on type of lock used for `MPI_Win_lock_all`, it can only provide shared access using `MPI_LOCK_SHARED` and the exclusive.

`MPI_Win_lock` call is in itself a *non-blocking* call hence it will return the control to the calling process just after the call and it will not wait for the target process to *acquire* the lock except in the case when the target process is same as the calling process, there MPI standard specifies it to be a *blocking* call.

Since `MPI_Win_unlock` performs the synchronization and that will be called at the end. But what if we want to use the communicated data before calling `MPI_Win_unlock`? In that case, there are few routines available for synchronization before calling `MPI_Win_unlock`.
For Separate Memory Model: [2]

- `MPI_Win_flush`: This routine can do force completion of all RMA routines that has been called till that point, both at the 'origin' as well as on target process.
- `MPI_Win_flush_local`: This routine can complete all RMA operations locally for the process which is calling the RMA operations.

Hence, the buffers involved in pending RMA operations at the origin process can be re-used safely after calling above routine.
For Unified Memory Model: [2]

- `MPI_Win_sync`: It synchronizes private memory with public memory window in passive RMA.

Unified model is indeed coherent but the timings of when the updates of public memory become visible in private memory is not explicitly defined hence it is advisable to use this function.

Note:
Writing RMA program by having a memory model in mind may limits portability hence prefer `MPI_Win_flush` operation as they are available on both models.

5.2.4 Memory models

Separate memory model
The entire RMA implementation is based on the concept of public and private memory. Local load and store operations use private memory while the RMA operations

[2] For memory models, see section 5.2.4

uses public (exposed) memory. In separate memory model, the private and public memory remains logically separated and public memory window holds copy of the variable from private memory. These type of model is generally used on **non-coherent** systems.

`MPI_Win_fence` essentially makes the sure that whatever changes were made to public memory has been copied to private memory as vice-versa. Different values of assert parameter can be provided to avoid unnecessary copying.The above reasoning also says that only RMA routines must be used to update memory location in window object.

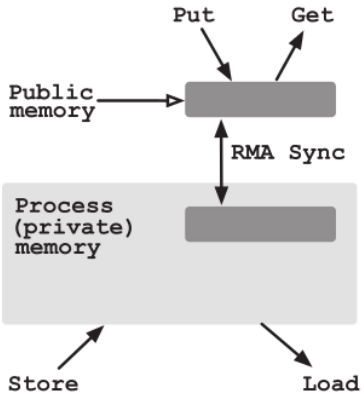


Figure 5.7: Separate Memory Model[1]

Unified memory model

In coherent systems, MPI implementations does not differentiate between the public and private memory i.e. changes made to private memory location will eventually become visible to public memory but the keyword here is *eventually*.

Hence programmers are advised to make sure the visibility of changes by using synchronization routines such as `MPI_Win_fence`, `MPI_Win_unlock` or `MPI_Win_Sync`

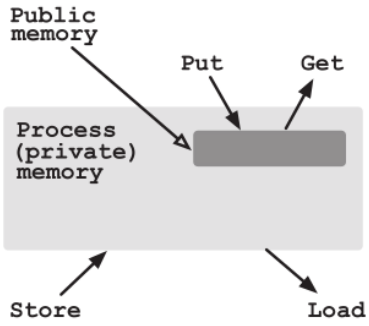


Figure 5.8: Unified Memory Model[1]

Note:
To determine whether the MPI environment provides the unified or separate memory model, one can check the the MPI_WIN_MODEL attribute on the window. The value of this attribute is a pointer to an integer that can have one of two values:

[1]All images credits: [7]

MPI_WIN_UNIFIED or MPI_WIN_SEPARATE for the unified and separate memory models, respectively.

5.3 Implementation

In this section, we will discuss about the implementation of classical Lax-Wendroff method for linear advection equation using finite difference approach in 1D case. The theory related to finite difference approach can be found in section 2.1 of chapter 2. This is the parallel code implemented using MPI RMA.

```
# getting boundary values and halo exchanges
function get_ghost_values!(param, u, win)
    if size == 1
        u[1] = u[param.N_local]
        u[param.N_local + 2] = u[3]
        return
    end
    next = (rank + 1) % size
    if rank !== size - 1
        buf1 = fill(0.0, 1)
        MPI.Win_lock(win; rank=next, type=MPI.LOCK_SHARED, nocheck=true)
        MPI.Put!(u[param.N_local+1], win; rank=next, disp=0)
        MPI.Get!(buf1, win; rank=next, disp=1)
        MPI.Win_unlock(win, rank=next)
        u[N_local+2] = buf1[1]
    else
        buf2 = fill(0.0, 1)
        MPI.Win_lock(win; rank=next, type=MPI.LOCK_SHARED, nocheck=true)
        MPI.Put!(u[param.N_local], win; rank=next, disp=0)
        MPI.Get!(buf2, win; rank=next, disp=2)
        MPI.Win_unlock(win; rank=next)
        u[N_local+2] = buf2[1]
    end
    MPI.Win_fence(win)
end
```

The above code is writtten for periodic boundary conditions. `get_ghost_values!` function is the only function that required MPI communication and rest all code is serial. In above function, the first `if` block is to handle the case if total number of processes is 1 essentially the serial case. Now, talking about the main part related to MPI RMA, we are using `lock/unlock` mechanism hence this is passive target synchronization.

`MPI_Put` has been used to write the values from `u[param.N_local+1]` to the `next` rank at the displacement 0 means at the starting of the window of `next` rank. Then `MPI_Get` has been used to get the values from `next` rank's displacement 1 to the buffer `buf1`. The displacement 1 simply means that we need to skip the first memory location in the window of the `next` rank.

At the end, `MPI_Win_fence` is used in order to complete the store operations that we are performing at the location `u[N_local+2]`. We can also omit the use

of `MPI_Get` completely and can just use `MPI_Put`. By doing this, we also not need to perform the store operation at the last hence no need to use `MPI_Win_fence`. `nocheck=true` is an optimization option that will skip the check for access to the window by other processes as we know that the code is written in such a way that there won't be any illegal access attempts to the window. The optimization in MPI RMA is the responsibility of the programmer by providing such options. This is not the most optimized code and it can further be optimized.

The reader should not get confused with `if` conditions as they are required due to periodic boundary conditions and subject to change as depends on boundary condition used.

In this project, I have also implemented the serial and parallel (MPI RMA) implementation of linear advection equation in 1D and 2D cases and is available on Github on this link.

Chapter 6

Analysis and Interpretation

6.1 Analysis of results

In this project, multi-node parallelization of the existing serial code for simulation of conservation laws have been performed. Multi-node parallelization enables the program to use the capabilities of a server consists of several nodes and each node having various computing units connected with fast interconnects such as `InfiniBand`. Each node also have its own memory hence in order to get access to other process's memory we need to exclusively communicate the information that we want the other process to have access and there comes `MPI` into the picture.

The parallel code has been tested by running two problems: **Kelvin-Helmholtz instability** and **isentropic Euler vortex**. The scaling tests for both of these problems have been performed and results are shown.

6.1.1 Speedup calculation

The speed up results have been shown for the parallel code. These speedup results are in comparison to serial code execution time which is essentially running the parallel code with total number of process as 1. Formula used for calculation speedup is

$$\text{Speedup} = \frac{\text{Execution time with 1 process}}{\text{Execution time with N processes}}$$

For **Kelvin-Helmholtz instability**,
Execution time with 1 process: 68230.47 seconds[3]
Execution time with 16 process: 5157.72 seconds

$$\text{Speedup} = \frac{68230.47}{5157.72} = 13.23 \text{ times}$$

For **isentropic Euler vortex**,
Execution time with 1 processes: 176760 seconds[3]
Execution time with 16 processes: 14904 seconds

$$\text{Speedup} = \frac{176760}{14904} = 11.86 \text{ times}$$

[3]All the data has been collected on Dual Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

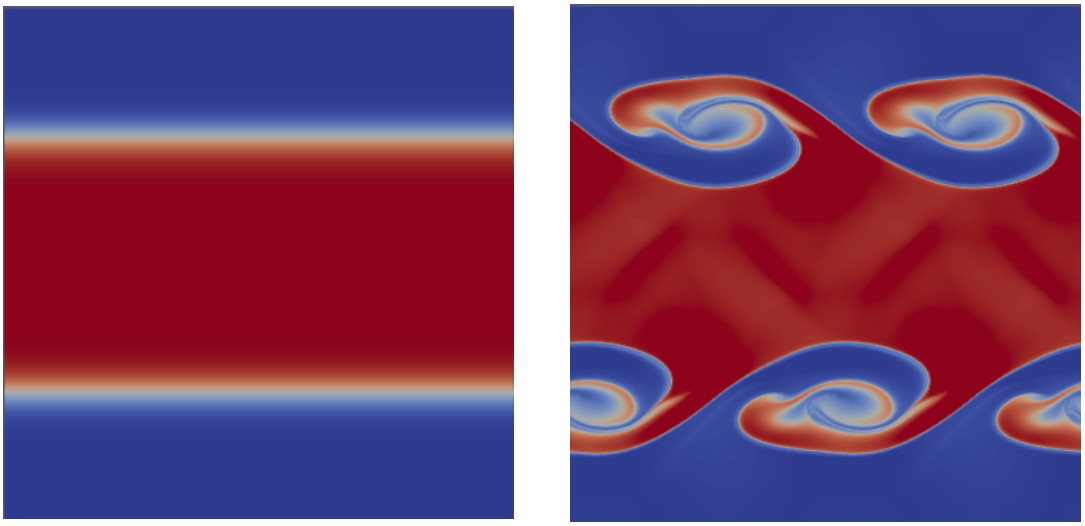
6.1.2 Efficiency calculation

The efficiency of a parallel code is a measure of how effectively a parallel program is able to utilize the available computational resources compared to its sequential counterpart. It is calculated by:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processors}} \times 100\%$$

For **Kelvin-Helmholtz instability**,
Speedup = 13.23
Total number of processes = 16

$$\text{Efficiency} = \frac{13.23}{16} \times 100\% = 82.7\%$$



(a) (b)

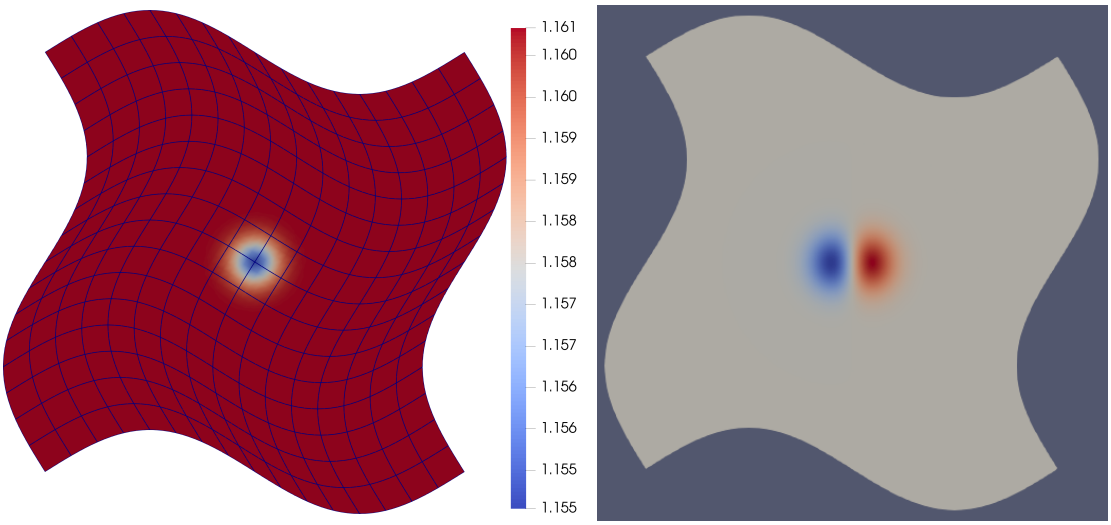
Figure 6.1: Kelvin-Helmholtz instability at $t = 3$ using polynomial degree $N = 4$, (a) Initial condition, (b) density plot

For **isentropic Euler vortex**,
Speedup = 11.86
Total number of processes = 16

$$\text{Efficiency} = \frac{11.86}{16} \times 100\% = 74.13\%$$

6.2 Scaling test

The numbers in speedup and efficiency that we got in results shows a promising hope for the problems of real world involving complex CFD simulations, big data etc., to be simulated using multi-node environment. These results can be improved further with better optimization techniques and utilising the features of modern hardware



(a) (b)

Figure 6.2: Isentropic vortex using polynomial degree $N = 4$, (a) density plot, (b) velocity in x direction

such as vector processors etc.
The results for scaling test are shown below for both the problems.

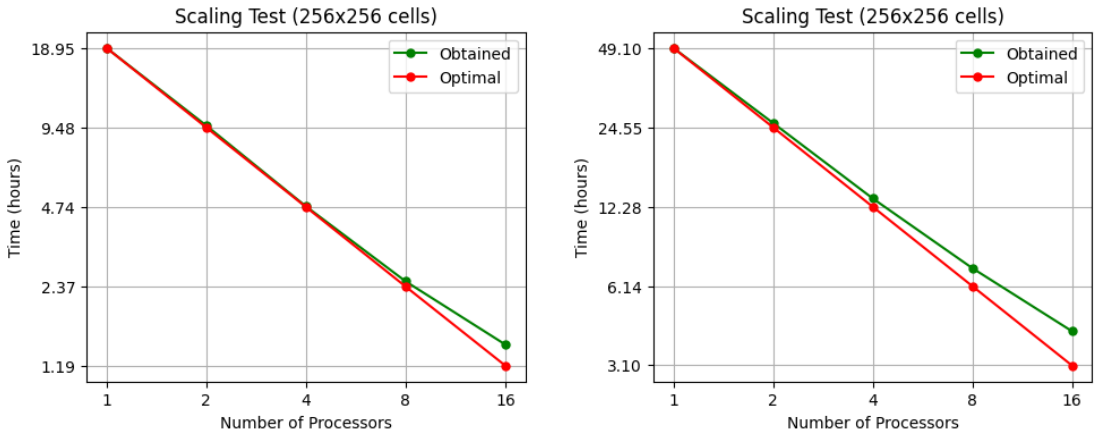


Figure 6.3: Scaling results

Chapter 7

Conclusion

The results presented in chapter 6, shows us that the real world problems that are data intensive and memory consuming, can be simulated with the help of parallel processing. They can be simulated with a good speedup and can utilize the hardware capabilities quiet effectively as shown by the results of efficiency. With harnessing the computing power and with larger pool of memory of many nodes connected with fast interconnects, we can simulate large problem in the fields of computational fluid dynamics, medical simulations, big data and many more.

We also have proposed a alternative methodology for parallelization in chapter 5. The implementation has not been done yet for alternative methodology and that remains the work for future projects.

The code following the methodology presented in chapter 4 is available on Github but currently it is private. This code can be accessed on this link after the paper [9] gets accepted for publication.

Chapter 8

Results

In this project, I have added support for multi-node parallelization in `TrixiLW.jl`. The code that has been written for this is available on Github and the link is provided in chapter 7. The code has been tested for two problem as mentioned in chapter 6 and respective initial conditions and results that have been generated by the code are shown there. These presented results have been verified with the results of serial version of the code. The serial version of the code and its respective results can be found in this paper [9]. The comprehensive results for parallel code are shown in analysis section 6

Scaling tests for the parallel code have also been performed for both the problem and the results have been shown in the section 6.2. The results of scaling test shows that the code has performed well with minimizing the overhead due to the MPI communication which is the major challenge of using parallel algorithms.

The efficiency results shows that the code is able to use the underline hardware effectively for communication which is a good sign for a parallel algorithm as these days we are seeing rapid development in networking and computing hardware.

Animation of the solution for isentropic vortex problem and kelvin-helmholtz instability.

References

[1] Chandrashekar, P. (2019). *Numerical methods for hyperbolic system of conservation laws.* https://math.tifrbng.res.in/ praveen/pub/ncm2019.pdf

[2] V.V Rusanov (1962). *The calculation of the interaction of non-stationary shock waves and obstacles.* USSR Computational Mathematics and Mathematical Physics, 1(2), 304-320.

[3] W. TROJAK AND F. D. WITHERDEN, *A new family of weighted one-parameter flux reconstruction schemes, Computers & Fluids*, 222 (2021), p. 104918.

[4] H. T. HUYNH, *A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods*, Miami, FL, June 2007, AIAA.

[5] P. VINCENT, F. WITHERDEN, B. VERMEIRE, J. S. PARK, AND A. IYER, *Towards Green Aviation with Python at Petascale, in SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, Nov. 2016, IEEE, pp. 1-11.

[6] Schlottke-Lakemper, M., Gassner, G. J., Ranocha, H., Winters, A. R., & Chan, J. (2024). *Trixi.jl (v0.8.3).* Zenodo. https://doi.org/10.5281/zenodo.12683615

[7] Hoefler, T., Balaji, P., Gropp, W., & Thakur, R. (2016, November). *Advanced MPI Programming* [Tutorial] https://web.cels.anl.gov/ thakur/sc16-mpi-tutorial/slides.pdf

[8] Hoefler, T., Gropp, W., Thakur, R., & Lusk, E. (2014). *Using advanced MPI: Modern features of the message-Passing-Interface.* MIT Press.

[9] Babbar, A., & Chandrashekar, P. (2024). *Lax-Wendroff Flux Reconstruction on adaptive curvilinear meshes with error based time stepping for hyperbolic conservation laws.* arXiv e-prints, arXiv:2402.11926.

Declaration

I, Devansh Tripathi (Full name) hereby declare that the details/facts mentioned above are true to the best of my knowledge and I solely be held responsible in case of any discrepancies found in the details mentioned above.

(Signature of Scholar)

Date: 26/07/2024

Place: TIFR-CAM, Bangalore, India