Stage de fin d'etudes

Adaptive Mesh Refinement with p4est

Student: Alexandru Fikl
Institut Sup' Galilée
alexfikl@gmail.com

Supervisor: Pierre Kestener
Maison de la Simulation
pierre.kestener@cea.fr

Supervisor: Samuel Kokh
Maison de la Simulation
samuel.kokh@cea.fr

Contents

1 Introduction **7**

2 Adaptive Mesh Refinement **11**
2.1 Block-based AMR 12
2.2 Cell-based AMR 14

3 The `p4est` Library **21**
3.1 Encoding the Forest 22
3.1.1 Morton Index 23
3.1.2 Inter-tree Connectivity 25
3.2 Traversing the Tree 28
3.3 Parallel Algorithms 30
3.4 Implementation and Data Structures 37
3.4.1 Creating a Forest 37
3.4.2 The Ghost Layer 41
3.4.3 Iterating Over the Octants 42
3.4.4 Adapting and Load Balancing 43
3.5 Numerical Results 45
3.5.1 Forest Creation, Adaptation and Partitioning 46

4 Discretization of the Transport Equation using `p4est` **51**
4.1 Dimensional Splitting 53
4.2 Flux Choice: The Upwind Scheme 54
4.3 Time Subcycling 56
4.4 Numerical Results 60
4.4.1 Advection on the Diagonal 60
4.4.2 Rotation of Zalesak's Disk 63
4.4.3 Deformation in 2D 65

5 Simulation of a Two-Phase Model using `p4est` **67**
5.1 Model Description 67
5.2 Numerical Scheme 69
5.3 Numerical Results 73
5.3.1 SOD: Shock and Rarefaction Wave 75
5.3.2 Two Rarefaction Waves 77
5.3.3 Dam Break 78

6 Conclusions **81**

Acknowledgments

I would like to express my gratitude to my two supervisors at *Maison de la Simulation*, M. Pierre Kestener and M. Samuel Kokh, for all their guidance and invaluable insight during this project. The long afternoons we have spend brainstorming and debating different features or design choices from `p4est` and AMR in general, have been both fun and incredibly productive.

I would also like to thank all the staff at *Maison de la Simulation* for making me feel welcome at all times. Their different fields of study and willingness to talk at length about them have provided a very enriching environment, full of ideas that one is unlikely to get acquainted with in a more academic setting.

Last, but not least, I would like to thank all the other people that were kind enough to proofread this work and help me sort out many little kinks.

1 Introduction

Solving partial differential equations has always been one of the most important facets of scientific computing, be it in theoretical mathematics, astrophysics, chemistry, etc. Many characteristics and properties as well as a good deal of intuition was developed around partial differential equations and the ways to solve them. Even though on a theoretical level knowledge has steadily advanced, the problems we face have also become larger and larger in scope.

A new challenge in tackling these large problems is utilizing the growing computational power available to scientists. There have been many attempts at improving well known numerical methods and transforming them in new ways to better fit the world of *High Performance Computing*, as well as designed novel techniques.

In this paper we will study a small segment of these efforts, namely the interest in adapting the mesh used for some numerical simulations. We will be looking at the `p4est` library [14] that tries to offer a complete framework for dealing with the mesh in highly parallel environments using scalable algorithms for creating, adapting and load-balancing the resulting mesh.

`p4est` provides an implementation of *Adaptive Mesh Refinement* (AMR), often referred to as **cell-based AMR** (in contrast to block-based AMR), that makes use of groups of quadtrees (in 2D) and octrees (in 3D) (conveniently called *forests*) to allow describing complex geometries and capturing phenomena developing at different scales (e.g. in the case of turbulence).

Internship

The research into `p4est` was done in the form of an internship, that doubles as a Masters project, in the third (and final) year at the engineering school *Institut Sup' Galilée* that is part of *Université Paris XIII*. The institution that sponsored the internship was *Maison de la Simulation*.

The main goals of this work, and internship, is to *document* and *evaluate* the `p4est` library. Even though the algorithms used in the `p4est` library have been explained in several articles ([14], [16] and [17]), they are highly technical and not straightforward to understand. We have complemented these articles with extensive examples that will, hopefully, allow for a broader understanding of the methods used by `p4est`.

To evaluate the `p4est` library we have developed a new code [31] that makes use of it, as a mesh-handling backend, to solve hyperbolic equations using different schemes involving

the *Finite Volume Method*, such as directional splitting and higher order MUSCL schemes. The code was created with a special focus on flexibility and readability to serve as further documentation for `p4est` and its capabilities.

Maison de la Simulation

Maison de la Simulation is a joint research facility between **CEA** (the *Centre d'énergie atomique*), **CNRS** (the *Centre national de la recherche scientifique*), **INRIA** (*Institut national de recherche en informatique et en automatique*) with the involvement of two universities: **Université Paris-Sud** and **Université de Versailles**. The laboratory was created in 2012 in the DigiteoLabs building in Saclay.

The main focus of the laboratory is promoting **High Performance Computing** in France and worldwide as part of a bigger effort to fully understand and make use of emerging technologies and advancing computing power.

In addition to being a research center, Maison de la Simulation is also an *expertise center* open to the scientific community. As part of this goal, there is an ongoing effort to educate both existing engineers and students to efficiently use the computing infrastructure available to them. Given this direction, Maison de la Simulation has a small number of permanent staff with most of the personnel consisting of temporary staff from various research institutions and PhD students.

As a hub for HPC in France, Maison de la Simulation has worked on various projects that involved either optimizing existing code (e.g. for *Gysela5D*), fully developing new applications (e.g. *Ramses-GPU*) or developing novel techniques (e.g. *new Godunov-type schemes for magnetohydrodynamics*).

Structure

The paper is structured in 6 chapters, including the current one, that will try to offer a broad view on the world of AMR and the `p4est` library.

Chapter 2 deals specifically with the historical context surrounding the developments of several Adaptive Mesh Refinement Methods. We will be looking at cell-based AMR and block-based AMR in detail and will try to integrate them into a wider research field for multiscale numerical simulations.

Chapter 3 tries to offer some insights into the `p4est` library by explaining the different methods it uses to store the mesh and the lengths to which it goes to provide novel parallel algorithms for AMR. We will also be looking at some preliminary tests for the functionality offered by `p4est`.

Chapter 4 will look at the first set of equations and numerical schemes that we have tested using the `p4est` library. We will be looking at the *transport equation* and the challenges posed by implementing a simple Finite Volume scheme in the context of AMR.

Chapter 5 presents a more complex two-phase model. This model will provide a more comprehensive test for `p4est` that will hopefully prove that it is well equipped to handle the different problems that arise from complex phenomena.

Finally, in Chapter 6 we will present the conclusions of our tests and opportunities for future work.

2 Adaptive Mesh Refinement

In this chapter we will try to give a short history of and introduction to the world of *Adaptive Mesh Refinement*.

When looking to solve ordinary or partial differential equations, we first proceed by discretizing the domain. This involves choosing a set of points from the domain and defining the unknowns of our systems at those points. This can be achieved by creating a mesh (or grid) or by using mesh-free methods that only involve the points, with no structure.

Even within methods involving meshes there is great variation: the mesh can be **structured** (elements have clear neighbors in each direction) or **unstructured** (sizes and shapes differ between elements), **conforming** (two elements can only share a complete edge) or **non-conforming** (an edge can be shared between more than two elements), etc. Usually, these classic mesh-based methods involve a mesh that, once it is constructed, does not change while the solution is being computed.

Unlike typical meshes, AMR is a technique that permits changing the mesh while the solution to the differential equations is calculated. This can be achieved in multiple ways, two of which we will look at in detail.

The first method is called **block-based** (or **patch-based**) AMR and it involves stacking multiple grids on top of each other in regions where more accuracy is required. The extra grids are always finer than the original grid they are supposed to replace. Once the extra grids are constructed, the equations are solved normally on the finer grids and then interpolated to the coarser ones.

A second method involves modifying the original mesh by splitting its elements into multiple parts and is known as **cell-based** AMR. This method is a larger departure from the usual static meshes as it employs the use of trees to store the mesh information and easily refine and coarsen specific cells.

Cell-based AMR can be applied equally to structured and non-structured meshes and always produces non-conforming elements. This is due to the fact that there will always be a cell that neighbors a refined cell (if there are no such cells, the mesh becomes uniform).

An alternative to AMR is the use of the previously mentioned mesh-free methods such as the **Smoothed Particle Hydrodynamics** [1] method that is used to simulate fluid flows. This type of Lagrangian method is very similar (in results) to *Adaptive Mesh Refinement* because it will naturally gather more points in the regions where the flow changes faster and thus allows for more accuracy where needed. It does have its downsides as well, some of which include the

inherent diffusive nature of the method thanks to the use of *kernel functions*, the problematic handling of contact discontinuities, high numerical viscosity, etc.

Another alternative AMR that has seen quite some attention lately is **Wavelet-based AMR**. Wavelets have been used for a long time in the fields of Signal Processing and Data Compression, in general, to impressive results. One of the main advantages of wavelet-based AMR is its strong mathematical foundations that allow for exact error calculation and performance optimizations. A comprehensive comparison between mesh-based AMR and Wavelet-based AMR is given in [2].

In the subsequent sections we will talk about the two major types of AMR that are of interest to us: block-based and cell-based AMR on structured meshes.

2.1 Block-based AMR

One of the first, if not the first, description of block-based AMR is given by M. J. Berger and J. Oliger in [3] (1984) with an application to hyperbolic partial differential equations. This paper was followed by another from M. J. Berger and P. Collela [4] (1989) which extended the method to take shocks into consideration and greatly simplified the AMR-related algorithms.

Both of these papers start with a regular uniform rectangular mesh. As the solution progresses, we can compute local error estimators that give a criterion for creating new, coarse grids that are contained in the initial domain. In the original article, these grids were allowed to be rotated in 2D space, but in [4] they have been aligned with the coarser grid to ease interpolation, handling of boundary conditions and construction algorithms.

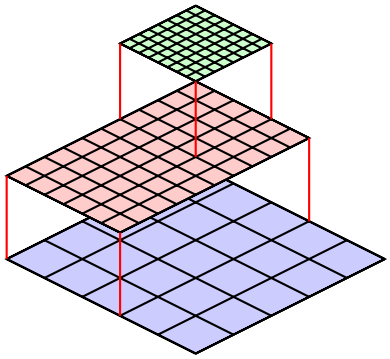


Figure 2.1: Example of overlapped grids.

One example of block-based AMR can be seen in Figure 2.1. The parent-child relationship between the grids is kept using a directed acyclic graph or using linked lists. In the scheme proposed by [4], fine grids are allowed to overlap multiple coarser grids and, of course, a coarse grid can contain any number of finer grids.

Formally, for a set of levels $l \in \{1, \ldots, b\}$ and a maximum refinement level b, we define $G_{l,k}$, the k-th grid at level l. Since the grids are uniform and rectangular, all the cells in the grid have the same size, which is Δx_l. This gives a refining ratio of:

$$r = \frac{\Delta x_{l+1}}{\Delta x_l},$$

which is usually chosen as a power of 2 to simplify calculations. The union of all the grids on a level gives G_l and, by extension:

$$G_1 = \bigcup_k G_{1,k} = D$$

is the whole domain. Furthermore, the grids must be *properly nested*, which is defined in [4] as:

- A fine grid starts and ends at the corner of a cell in the coarser grid.
- Two overlapping cells have to be separated by exactly one level.

Once the boundary conditions are defined from the neighboring grids, a grid at level l is completely independent from all the others and we can solve the equations on it as if it were the only one. This gives block-based AMR the very useful quality of being able to naturally re-use existing codes that do not adapt the mesh. Once the solutions have been independently calculated, additional interpolations and flux corrections are required to transmit the solution to lower grids and ensure qualities such as conservation.

An important part of AMR, which we will only glance at now, is finding a good refining criterion. In [4], M. J. Berger and P. Colella look at the Euler equations and use a finite difference scheme for solving them. Using the finite difference approximation, they define:

$$w(x, t + \Delta t) - Qw(x, t) \approx \tau(x, t) + \Delta t \mathcal{O}(\Delta t^{q+1} + \Delta x^{q+1}),$$

where q is the order of the finite difference approximation and Q is an operator that gives the desired discretization. To compute the local truncation error, they compute the solution at time $t + 2\Delta t$, first by doing two steps of Δt, denoted by Q^2, and then by doing a single time step with a coarsened cell size $2\Delta x$, denoted $Q_{2\Delta x}$. The error is then given by:

$$\frac{Q^2 w(x, t) - Q_{2\Delta x} w(x, t)}{2^{q+1} - 2} = \tau(x, t) + \mathcal{O}(\Delta x^{q+2}).$$

This is easily achieved on the current mesh structure by advancing two steps in time with the regular integration scheme and doing another two with every other point in the grid. Although their use case was rather specific, such a method can be used with any other type of equations and discretization methods. Other a posteriori error estimation methods can also be used to the same effect.

Once the respective cells of the mesh are flagged for refinement, the framework usually constructs bounding boxes over sets of flag cells and then creates new layers of meshes with a refinement ratio r. This method usually leads to refined areas where the solution is smooth

and does not require a finer mesh, thus leading to higher memory consumption than one would get with cell-based AMR where only the required cells are adapted.

The description given in [4] of block-based AMR is still largely in use today. Some of the frameworks that provide this sort of AMR are:

AMRClaw is part of the bigger **Clawpack** framework for solving linear and non-linear hyperbolic systems of conservation laws. **Clawpack** was originally written by R. LeVeque, one of J. Oliger's students, and the introduction of AMR techniques was done after a collaboration with M. J. Berger [5].

BoxLib is another AMR framework that has support for hyperbolic, elliptic and parabolic PDEs. One special advantage of BoxLib is that it has been proven to scale very well up to 200.000 processes.

Paramesh is another general AMR framework developed by NASA. The general structure is describe in [6].

Others. There are many other AMR codes out there, such as **Chombo** (that shares a common history with BoxLib), **AMROC**, **SAMRAI**, etc.

2.2 Cell-based AMR

We have seen in the previous section a short history of block-based AMR, but the focus of this work is, of course, on cell-based AMR. Unlike block-based AMR, where we would refine a bigger portion of a mesh by a factor r, in cell-based AMR each cell is refined independently by a fixed factor of 2 in each direction.

If we consider the 1D case, we will need to successively divide into two parts an interval $[0, L]$. All possible subintervals (or cells) given by this sort of refinement are of the form:

$$I_{k,l} = \left[\frac{L}{2^l}k, \frac{L}{2^l}(k+1)\right], \quad k \in \{0, 2^l - 1\}, \tag{2.1}$$

where $l \in \{0, b\}$ is the level of refinement that is limited to a maximum value b. For a given level of refinement l, we have:

$$\bigcup_k I_{k,l} = [0, L],$$

but, of course, not every cell gets refined to a fixed level l. To handle the individual refinement of each cell, the most natural data structure to use is a tree, more specifically: a **binary tree** for 1D domains, a **quadtree** for 2D domains, an **octree** for 3D.

Unlike block-based AMR, where trees were only used to handle the grid hierarchy, cell-based AMR makes heavy use of tree structures to store the mesh and perform modifications on it. The use of new data structures implies new difficulties in implementing numerical methods (new integration routines, storage strategies and load balancing techniques need to be developed) and integrating them with existing codes.

There has been a lot of previous research into quadtrees and octrees in the world of computer graphics, with the works of R. A. Finkel and J. L. Bentley in [8] in describing quadtree algorithms and D. Meagher in [7] in describing octree algorithms. Although these algorithms are not concerned with numerical simulations, they were still very useful because they provide efficient ways of searching, storing and modifying these data structures.

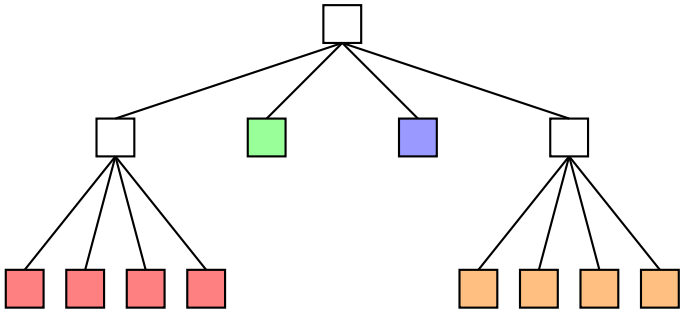


Figure 2.2: Example of a quadtree.

In Figure 2.2, we can see an example of a quadtree. A quadtree, like any other tree structure, is directed (in the sense that there is only one way from one node to another) and does not contain any cycles. Some of the main elements of a tree (quadtree or octree) are:

- The node on top that is called the **root node**.
- Each node can be tied to 0 or 4 (resp. 8 for an octree) other nodes beneath it. This node is called the **parent** and the 4 (resp. 8) nodes beneath it, if they exist, are called **children**. A group of children is sometimes referred to as a **family** and they have **sibling** relationships between them.
- If there are no nodes beneath a certain node, it is called a **leaf**.

This is an abstract definition of a tree (quadtree or octree) that makes no reference to a discretized grid. In the case of numerical simulations, each node is likely to represent a certain subset of the whole domain (see (2.1)).

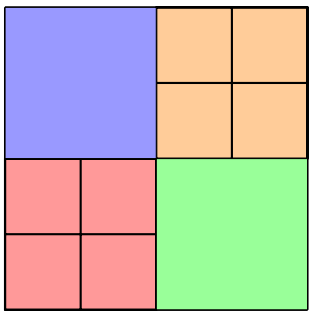


Figure 2.3: Example of a refined square domain that can be represented by a quadtree.

The information that describes the tree needs to be accessible at all times during a simulation. Such a requirement raises several issues, especially in high performance environments:

- Storing the tree. Tree storage can be **linear** (in the sense that it is stored in a linear array) or make heavy use of **pointers** from parent to children. When storing the tree linearly, we have a choice of storing the whole tree or *storing only the leaves.*
- Distributing the tree. In parallel environments, the tree structure that represents a physical domain has to be partitioned between different processes. This can be achieved using **space-filling curves** such as the Hilbert curve or the z-curve.
- Scalable algorithms. The data structures and algorithms used have to consume as little memory as possible to represent the tree structure and as little computations as possible to traverse it.
- Representing complex geometries. Most of the cell-based AMR algorithms and implementations focus solely on **square or rectangular domains** (see Figure 2.3).

Let us now briefly review 3 examples, in chronological order, that have addressed the above issues in different ways, each providing an improvement over the one before.

Example. One of the first attempts to tackle these problems in the context of numerical simulation has been done in [9] with an application to computational fluid dynamics in aeronautics. The work was done on octrees and proposed the following way of describing the data structure: the complete tree is stored (parents pointed to their children) and simple accumulation indexes were used to keep track of all the cells in the mesh (see Figure 2.4).

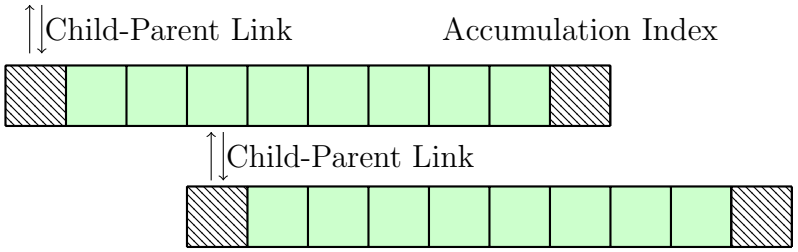


Figure 2.4: Tree Links as described in [9].

Each node in the tree contains the following information: a pointer to its parent, 8 consecutive pointers for each of its children (these can be NULL if the child is not refined), pointers to its neighbors and an accumulation index that stores the number of nodes in the tree up to this point. Besides storing the tree structure, each node also has to contain other information such as coordinates, centroids, size, level, neighborhood, etc. that are very necessary when doing numerical simulations.

Example. Storing the whole tree including neighborhood information is very costly, memory-wise. In fact, the memory requirements are comparable to AMR using non-structured meshes (where all the tree and the mesh connectivity has to be stored explicitly). An improvement to the previous example was given in [10] with the introduction of the **Fully Threaded**

Tree. This new data structure improves the memory requirements of storing an octree and is easier to parallelize.

The main obstacle to parallelization in [9] was the fact that each node contained pointers to its neighbors for easy access. While this had some benefits, in a parallel environment, it is very difficult to guarantee that these pointers stay valid while refining and coarsening different cells. The fully threaded tree stores instead pointers to parents of neighboring nodes that do not change in a refinement or coarsening pass.

The memory improvements come from the following simple observations:

- When a cell is split, all its children are created simultaneously. Thus, they can be stored in a single contiguous array (same as in [9]).
- Neighbor information between siblings is known automatically from the way they are stored in the array.
- The neighbors of a cell are either its siblings, children of its parent's siblings or one of its parent's siblings.

Much like in [9], the nodes of the tree actually hold information about 8 cells of the 3D domain and they are called *octants*. As can be seen in Figure 2.5, an octant contains: a pointer to its parent, its level, its coordinates, 6 pointers to parent cells of neighboring octants and its 8 children. The children usually contain the state vector of the equations we're trying to solve and a pointer to another node, if they are refined.

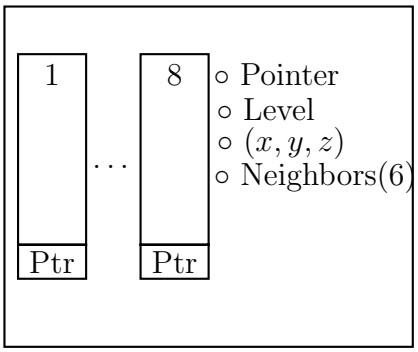
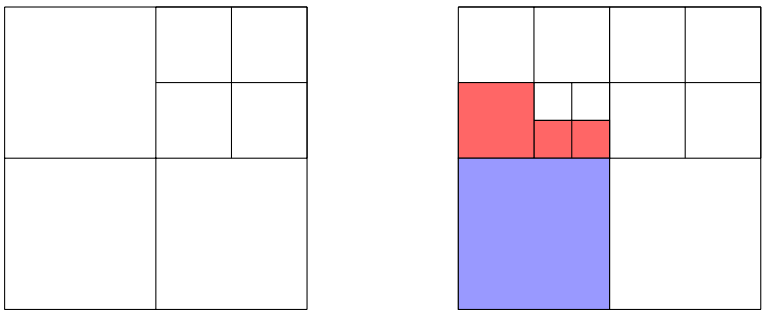


Figure 2.5: An octant as described in [10].

A very strong constraint in cell-based AMR that is mentioned in [10] and [9] is that the level difference between two neighboring octants cannot be bigger than 1 (see Figure 2.6). This is a constraint that is often used in AMR (it also applies to block-based AMR) and it is required mainly to simplify the algorithms. If this constraint is not imposed, finding a neighbor of a cell would require a recursive algorithm that traverses the tree starting from a given sibling and the number of neighbor would increase greatly with the gap between levels.



(a) Valid (b) Invalid

Figure 2.6: Example of (a) a valid refined mesh and (b) a invalid refined meshes.

Example. Another iteration on cell-based AMR has been brought to this data structure in [11]. The main improvement proposed here is based on the simple observation that we could use the triplet (i, j, l), where (i, j) are indexes of each cell in a uniform Cartesian mesh of level l, to describe the tree. This idea is derived directly from the way we subdivide a given interval by successive refinements, as seen in 2.1.

Given the index (i, j) of a cell, we could easily find the coordinates of its children:

$$(i_c, j_c) = \left\{ (2i + m, 2j + n) \mid (m, n) \in \{0, 1\}^2 \right\},$$

its parent:

$$(i_p, j_p) = \left(\left\lfloor \frac{i}{2} \right\rfloor, \left\lfloor \frac{j}{2} \right\rfloor \right)$$

and, using the origin (x_0, y_0), even the coordinates in physical space:

$$(x, y) = \left(x_0 + i\frac{L}{2^l}, y_0 + j\frac{L}{2^l} \right).$$

Notation. $\lfloor x \rfloor$ represents the integer part of x.

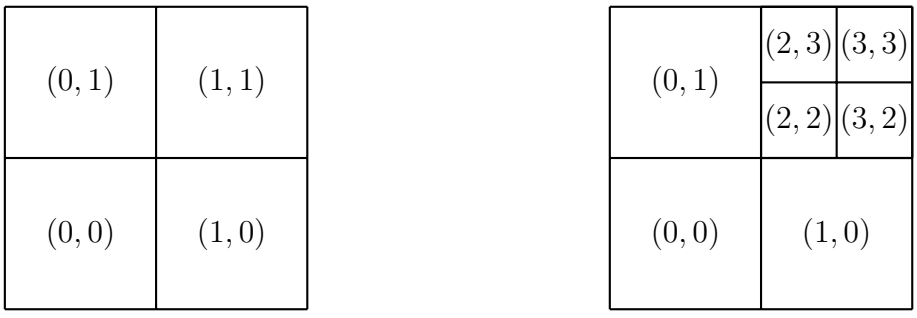


Figure 2.7: Grid indexes as described in [11].

Using this indexing system there are simple ways to find the (i, j) coordinates of any neighboring cell once we know those of the current cell. For example, to find the index of a

neighboring cell to the right that is twice as big, we would simply have to check if the cell with the following coordinates exists:

$$\left(\left\lfloor \frac{i+1}{2} \right\rfloor, \left\lfloor \frac{j}{2} \right\rfloor\right).$$

To optimize the search for the neighboring cell, [11] has proposed the introduction of a hash table where each octant is associated with a key:

$$\text{key} = \sum_{l=0}^{level-1} 2^{2l} + i2^{level} + j,$$

where (i, j) is the index and *level* is the level of a given octant. This implies that the tree is no longer stored in memory using pointers, but linearly using the above mentioned hash table. This hash table allows $\mathcal{O}(1)$ access to all the neighbors of a given octant.

Some of the libraries that use cell-based AMR are:

Ramses uses the fully threaded tree data structure to implement the magnetohydrodynamics equations and the N-body equations for use in astrophysics.

Gerris is a code that does incompressible Euler and Navier-Stokes equations. It has been developed by Stéphane Popinet since 2001 and defines its own C-like language for describing the problem, the boundary conditions, etc.

CLAMR is a code that uses a *hash table*-based approach to doing cell-based AMR on the **GPU** using the `OpenCL` framework. The framework was developed at Los Alamos and is better described in [12].

Others. A review of AMR frameworks and applications using AMR can be found on Donna Calhoun's website [29].

Let us underline that both RAMSES and Gerris are not AMR frameworks, they are *applications that make use of AMR* to solve equations that appear in astrophysics and fluid dynamics, respectively. As a consequence, the AMR functionality is more entangled with the application code and thus harder to separate and understand on its own. Contrary to these approaches, `p4est` [14] offers a pure AMR framework, making no suppositions on the type of equations and methods that the user wants to implement.

3 The p4est Library

p4est [14] is a library that specializes in parallel, *cell-based Adaptive Mesh Refinement*. It uses octrees and quadtrees as its main data structures and groups them together into conveniently called *forests of octrees* or *forests of quadtrees* to construct more complicated geometries. The main goal of the library is to scale the previous attempts at cell-based AMR to hundreds of thousands of processors and provide a simple interface for other projects to make use of. The library is **open-source** and is available at [30].

It does not try to impose any equations or numerical schemes on the user, focusing solely on handling the mesh. This has allowed successful implementations of different methods, such as *Finite Volumes*, *Finite Elements*, *Discontinuous Galerkin*, etc. Some of the mesh-specific functions it offers are: refining and coarsening, partitioning the octants between processors (load balancing), constructing ghost layers, iterating over the elements of the mesh, etc.

We have seen in the previous chapter the evolution of cell-based AMR methods that has culminated with the works in [11]. p4est takes some of the ideas present there, notably in the way it reasons about storing and working with the tree structure. For example, quadrants are represented by their indexes on a uniform mesh and the tree is stored linearly (in a simple list, in the case of p4est). A major difference between the two is that p4est only stores the leaves of the tree and does not group them into octants of 4 (or 8) cells.

p4est also uses a different naming scheme than [10] or [11]. It refers to *octants* (and *quadrants*) as single cells in the domain (not groups) that also represent nodes in the tree. Unless explicitly specified, we will use *quadrant* and *octant* (resp. *quadtree* and *octree*) interchangeably to mean the same thing.

To construct complicated geometries, p4est decomposes the space into K octrees (a *macro* level) that are then further partitioned into octants (a *micro* level). This implies that any domain Ω can be represented as a union of K octrees, where each octree is mapped from a *reference cube* by a smooth function

$$\phi_k : [0, 2^b]^d \to \mathbb{R}^3, \quad \forall k \in [\![0, K]\!],$$

which gives:

$$\Omega = \bigcup_k \phi_k([0, 2^b]^d),$$

where b is the maximum allowed refinement level of an octree.

The only restriction to this scheme is that the macro-decomposition of the domain Ω has to be conforming, meaning two trees can only share a face or edge in its entirety or not at all.

This allows p4est to represent a very wide array of geometries: from simple cubes to spheres to the Möbius strip.

3.1 Encoding the Forest

We have seen that p4est uses transformation of reference cubes to define complex geometries. Each $[0, 2^b]^d$ cube represents a tree with a maximum level of refinement b. This implies that, since we can only divide by a factor of 2, we will always get *integer coordinates* for all points in the mesh. These integer coordinates are then used to describe the cells in the mesh: each octant is uniquely described inside a tree by the coordinates $(x, y, z) \in \{0, \ldots, 2^b - 1\}$ of its lower-left corner and its level of refinement $l \in \mathbb{N}$.

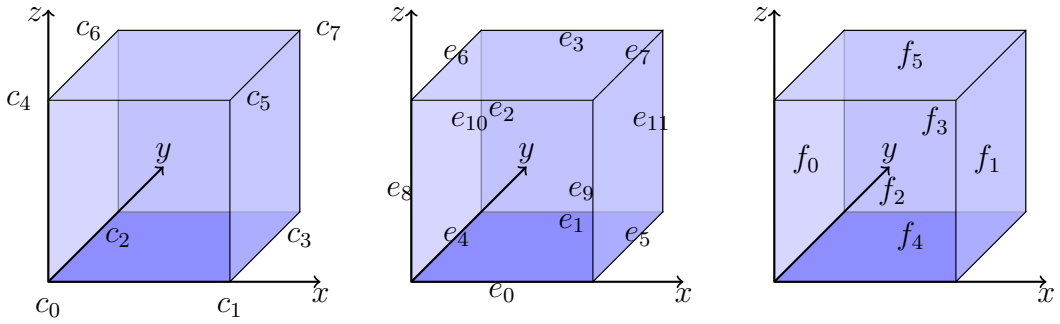


Figure 3.1: Numbering of the corners, edges and faces for an octant.

Definition 3.1. As seen in Figure 3.1, we denote:

- Each corner of a quadrant or octant by c_i, for $i \in \{0, 2d\}$. Each corner of the quadrant corresponds to a point in physical space that we will call $v_i \in \mathbb{R}^d$.
- Each edge of an octant by e_i, for $i \in \{0, 12\}$.
- Each face of a quadrant or octant by f_i, for $i \in \{0, 2d\}$.

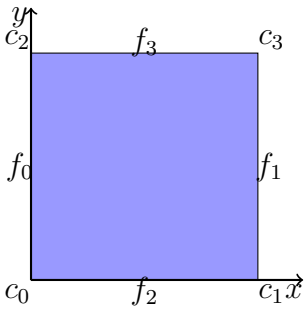


Figure 3.2: The corners and faces of a quadrant.

`p4est` uses a specific ordering for the corners, edges and faces of each octant (see Figure 3.2 for 2D and Figure 3.1 for 3D). They are using the so-called *z-order* numbering that says we first walk in the x direction, then y, then z. Taking as an example the faces, we first take the two faces f_0 and f_1 that are left and right in the x direction, then proceed to f_2 and f_3 that are front and back in the y direction and finally we have face f_4 and face f_5 that are bottom and top in the z direction.

Trees are numbered by their index k and have their own coordinate system and orientation. This means that two trees can be arbitrarily rotated around each other, as can be seen in Figure 3.3.

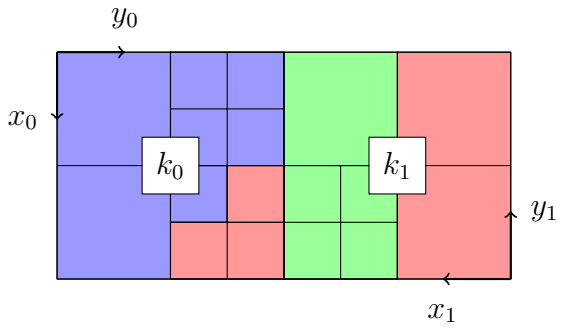


Figure 3.3: Two trees with different orientations.

3.1.1 Morton Index

In `p4est`, octants are stored in per-tree linear arrays. For linear storage, we would require a function that moves from describing each octant by (x, y, z, l) to just (m, l) where m is the linear index and l is the level of the quadrant. The mapping $(x, y, z, l) \to (m, l)$ has to be a 1-to-1 relation, thus a bijection.

A method that is commonly used when working with trees and storing them linearly is **a space filling curve**. Multiple such curves exist, for example: the Hilbert curve, the Peano Curve or the Morton curve. `p4est` uses the **z-curve** (or Morton curve) to construct a linear array from the tree. The name of the curve comes from the **z** shape that it exhibits (see Figure 3.4, quadrants 0-3).

Space-filling curves allow for good load balancing properties (see [11] or [13]), but they are rarely used to also store the tree and provide cache locality, as in `p4est`.

To better understand how a z-curve is constructed, we can look at Figure 3.4. One intuitive way of finding the order of the leaves in the z-curve is to look at the tree structure on the left: we can see here that the z-curve simply does a pre-order traversal of all the leaves in the tree.

Another way of looking at the *z-index* is by thinking of it as the result of applying a function that maps $(x, y, z, l) \to m$, where m is the z-index. In `p4est`, this function takes the binary

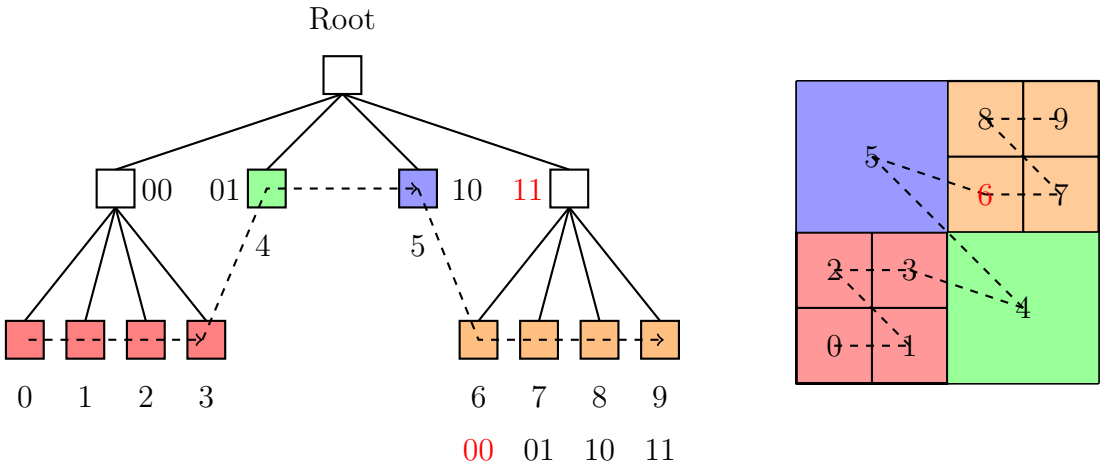


Figure 3.4: z-order traversal of the quadrants in a tree.

representation of the coordinates (x, y, z) and interleaves them to construct the Morton index m whose binary representation is given by:

$$m_{di+2} = z_i, \quad m_{di+1} = y_i, \quad m_{di+0} = x_i, \quad \forall i \in [\![0, b-1]\!],$$

where x_i, y_i and z_i are the i-th bits of each coordinate.

For a concrete example of how to compute the z-index of a quadrant we will take *quadrant 6* in Figure 3.4. Since we are in the reference cube $[0, 2^b]^d$, the coordinates of lower left corner of quadrant 6 are exactly the center of the domain $(x, y) = (2^{b-1}, 2^{b-1})$ and the level of the quadrant 6 is 2. For a maximum level of refinement $b = 5$, the binary representation of the coordinates is:

$$x = 10000 \text{ and } y = 10000.$$

To compute the Morton index, we will simply interleave the individual bits of the binary representation of x and y, as follows:

```
_1 _0 _0 _0 _0 = 16 = x
1_ 0_ 0_ 0_ 0_ = 16 = y
11 00 00 00 00 = 768 = z-index.
```

The Morton index gives a unique ordering of the octants in a single tree. To get a total ordering of all the octants in the forest, we must also introduce the tree index k. As we will see in the next section, the pair (m, l) of an octant can be used to traverse all its ancestors, its children, find its neighbors and so on.

Even though the quadrants are ordered in the array by their z-index, it is very important to remark that the z-index 786 that we have obtained has little to do with the position in the array of the quadrant number 6. The only property that they share is that they both offer an ordering of the quadrants in the tree. This means that the mapping $i \mapsto m$, which given the index in the linear array finds the corresponding Morton index, is strictly increasing.

For example, let's compare quadrant 5 and quadrant 6 with respect to their Morton index. The Morton index of quadrant 5 of coordinates $(0, 2^{b-1})$ and level 1 is:

```
_0 _0 _0 _0 _0 = 0   = x
1_ 0_ 0_ 0_ 0_ = 16  = y
10 00 00 00 00 = 512 = z-index,
```

which is smaller than the Morton index 768 of quadrant 6. However, if the grid is **uniform** at level b, the Morton index and the array index are the same. For a uniform grid of level l, we would have to bit shift the Morton index by $d \times (b - l)$ to get the index in the linear array.

3.1.2 Inter-tree Connectivity

Remark. In what follows, we shall identify a tree with its root quadrant.

We will now try to explain the **macro-connectivity** between the different trees in a forest represented by `p4est`. To completely describe this macro-connectivity, we need to:

1. Match each corner of each tree with a vertex of the physical space.
2. Describe the neighborhood of each tree across its faces.
3. Describe the orientation of each tree that is a face neighbor for a given tree k.
4. Describe the neighborhood of each tree across its corners. Only for corners where such a neighborhood exists.
5. In 3D, also describe the adjacency with respect to edges.

We will take the simple 2D example from Figure 3.5 and describe all the different elements. The domain described by the two trees k_0 and k_1 is periodic in y and has open boundaries in x. Furthermore, we can see that the two trees do not have the same system of coordinates, making for a sufficiently complicated forest to describe all the above mentioned items.

Physical Coordinates

The first thing that needs to be defined are the vertices in physical space. We can see that the corners referred to by c_3 and c'_1 (resp. c_2 and c'_3) represent the same physical vertices, so we only need to define 6 vertices v_i, for $i \in \{0, \ldots, 5\}$, to completely define the position of the two trees in physical space. Once we have defined these points we can assign them to each tree using an array called $\mathcal{NV}$, such that, for a given tree k and a given corner c, we can get the corresponding vertex $v_i = \mathcal{NV}(k, c)$.

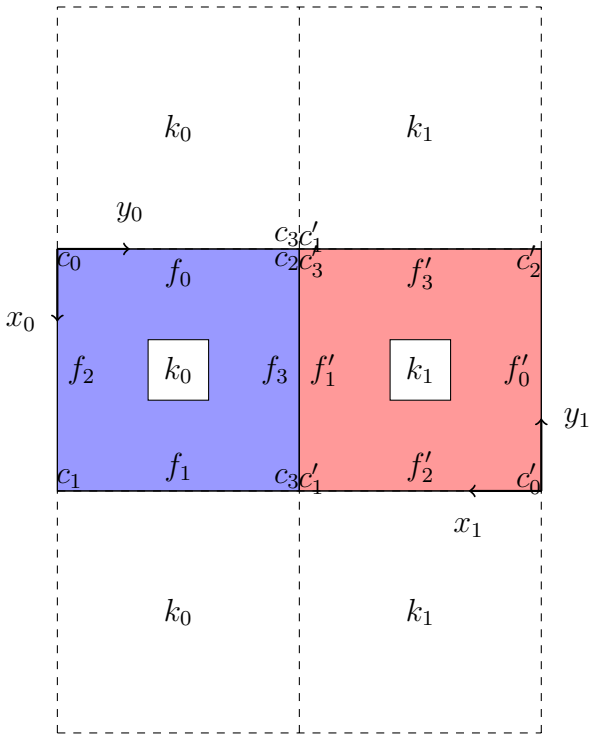


Figure 3.5: Coordinate system, corner and face numbering of two adjacent trees.

Face Neighbors

Next, we define the array $\mathcal{NO}$ which, for each tree k and face f, will give the corresponding tree on the other side. In our case, this is described by:

$$\begin{cases} \mathcal{NO}(k_0, \cdot) = \{0, 0, 0, 1\}, \\ \mathcal{NO}(k_1, \cdot) = \{1, 0, 1, 1\} \end{cases}$$

Indeed, we can see here, as in Figure 3.5, that the tree k_0 touches on the tree k_1 across the face f_3. If there is no other tree across a face boundary, the index of the originating tree is used.

Tree Orientation

The relative orientation between trees and the faces that they share are describe by the array $\mathcal{NF}$. Considering a tree k and a face f, $\mathcal{NF}(k, f) = 4r + f'$ where:

- f' is the index of the connecting face from the perspective of the tree $k' = \mathcal{NO}(k, f)$. The faces f and f' represent the same line (resp. surface) in the physical space.
- r is the orientation of the two trees relative to each other across the face f. If we denote $\{a, b\}$ as the indexes of the two corners that belong to the face f and $\{a', b'\}$ the corners

of the face f', the orientation r is given by:

$$r = \begin{cases} 0, & \text{if } (a-b)(a'-b') > 0, \\ 1, & \text{if } (a-b)(a'-b') < 0. \end{cases}$$

In our case, the array $\mathcal{NF}$ is defined as:

$$\begin{cases} \mathcal{NF}(k_0, \cdot) = \{1, 0, 2, 5\}, \\ \mathcal{NF}(k_1, \cdot) = \{0, 7, 3, 2\} \end{cases}$$

First we will handle the periodic nature of the domain. We see that $\mathcal{NF}(k_0, f_0) = f_1$ and $\mathcal{NF}(k_0, f_1) = f_0$ (same for the faces in the y directions of the tree k_1) which defines the periodic link from the upper boundary of the domain to the lower boundary.

For the domain boundaries in the x direction, as in the case of the $\mathcal{NO}$ array, the face of the originating tree is used. Thus, we have $\mathcal{NF}(k_0, f_2) = f_2$ and $\mathcal{NF}(k_0, f_0') = f_0'$.

The orientation on the previously mentioned faces is always 0 because, as seen in the array $\mathcal{NO}$, the neighboring tree is always the tree of origin which cannot be rotated around itself. The only faces that are an exception to this are face f_3 of the tree k_0 and face f_1' of the tree k_1, where we have:

$$\begin{cases} \mathcal{NF}(k_0, f_3) = 4r + f_1' = 5, \\ \mathcal{NF}(k_0, f_1') = 4r + f_3 = 7, \end{cases}$$

with an orientation $r = 1$.

Corner Neighbors

Next we will look at the connectivity across corners. It is stored in an array $\mathcal{CT}$ which, for a tree k and a corner c, stores the pair (k', c') of the corresponding neighboring tree and its own corner c that represents the same vertex as c.

It is important to note that connectivity across corners is only needed if two trees are connected *only by a corner*, and not a face or edge, like in Figure 3.6. This is the case for us because of the periodicity we introduced in the physical y direction. In Figure 3.5 we can see that, in fact, the corner c_2 of k_0 links the tree k_0 and the tree k_1 across a single point on the diagonal.

So we can define our array as:

$$\begin{cases} \mathcal{CT}(k_0, \cdot) = \{(*, *), (*, *), (1, 1), (1, 3)\}, \\ \mathcal{CT}(k_1, \cdot) = \{(*, *), (0, 2), (*, *), (0, 3)\} \end{cases}$$

The extension of this encoding in 3D is explained in [14] as well as the additional definition of the various transformations that are needed to pass from one tree to another, i.e. from one coordinate system to another. These transformations are done using matrices since all the possible combinations are computable in advance.

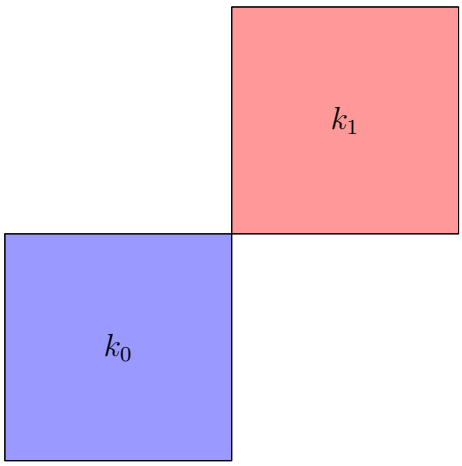


Figure 3.6: Two trees connected solely through a corner.

3.2 Traversing the Tree

In this section we will look over a few algorithms that permit finding the parent of an octant and its children. Other algorithms, for finding face, edge and corner neighbors, finding other descendants and ancestors, dealing with inter-tree boundaries, etc., are described in [14].

Since most of the calculations we will be doing are performed on the binary representation of the coordinates (x, y, z), we introduce the following notation:

- & denotes a **binary AND** that gives 1 only if the two bits are also 1.
- | denotes a **binary OR** that gives 1 if any of the two bits is 1.
- $\neg$ denotes a **binary NOT** that gives 1 if the bit was 0, and 0 otherwise.
- `(condition ? a : b)` is the **ternary operator** commonly used in `C`-type languages. It executes `a` if `condition` is true and `b` otherwise.
- For an octant o, we will denote by $o.l$ its level and by $(o.x, o.y, o.z)$ its coordinates.

Finding the parent of an octant. We will first look at Algorithm 1 that allows us to find the parent of a given octant o by finding its coordinates and level.

Algorithm 1: Get the parent of an octant.

Data: Octant o

1 $h \leftarrow 2^{b-o.l}$;
2 $q.l \leftarrow o.l - 1$;
3 $q.x \leftarrow o.x \ \& \ \neg h$;
4 $q.y \leftarrow o.y \ \& \ \neg h$;
5 $q.z \leftarrow o.z \ \& \ \neg h$;

To understand Algorithm 1 we will take the simple example from Figure 3.7. In this figure we see the parent q on the left getting refined into its 4 children on the right; we will look at the second of these children, namely o (both of them are hatched).

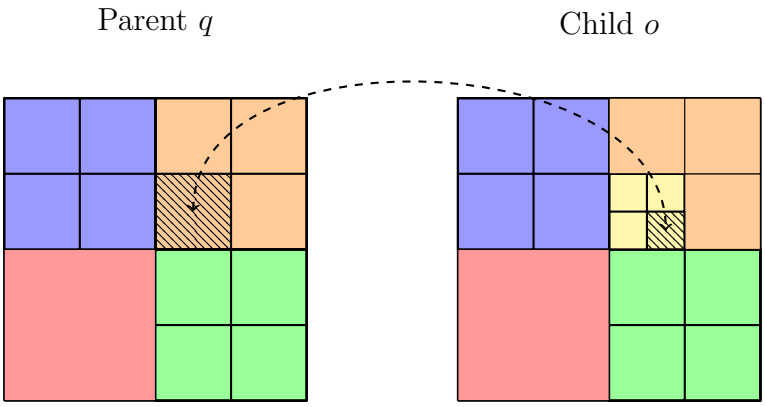


Figure 3.7: Parent-child relationship in a mesh.

Since we already know from computing the Morton index how the coordinates are computed and stored in `p4est`, we can do it ourselves and then see if the results given by Algorithm 1 match.

For a maximum level of refinement $b = 5$, the coordinates in their binary representation are:

- for the parent q of level $q.l = 2$, we have $(q.x, q.y) = (10000, 10000)$.
- for the child o of level $o.l = 3$, we have: $(o.x, o.y) = (10100, 10000)$.

The coordinates can be obtained by correctly subdividing the interval $[0, 2^b]$.

The last variable we need in order to perform the calculation is h. In our case, for the child quadrant o, $h = 2^{b-o.l}$, which in binary notation is 00100. So, according to Algorithm 1, the coordinates of the parent are given by:

$$\begin{cases} q.x = o.x \ \& \ \neg h = 10100 \ \& \ 11011 = 10000, \\ q.y = o.y \ \& \ \neg h = 10000 \ \& \ 11011 = 10000 \end{cases}$$

which is indeed the case.

Remark. If we consider the coordinates of the parent and child quadrant on the uniform mesh of their respective level, we have:

$$(q.x, q.y) = (2, 2) \text{ and } (o.x, o.y) = (5, 4).$$

This representation allows another way of finding the coordinates of the parent q from the coordinates of child o through a simple division by 2:

$$\begin{cases} q.x = \lfloor o.x/2 \rfloor = \lfloor 5/2 \rfloor = 2, \\ q.y = \lfloor o.y/2 \rfloor = \lfloor 4/2 \rfloor = 2, \end{cases}$$

which is the representation given in [11].

Finding the child of an octant. To find the i-th child of a quadrant, we will have to do the inverse operation to that suggested in Algorithm 1. This method is described by Algorithm 2.

Algorithm 2: Get the child of an octant.

Data: Octant q, Child index i

1 $h \leftarrow 2^{b-(o.l+1)}$;
2 $o.l = q.l + 1$;
3 $o.x \leftarrow q.x \mid (i \;\&\; 1 \;?\; h : 0)$;
4 $o.y \leftarrow q.y \mid (i \;\&\; 2 \;?\; h : 0)$;
5 $o.z \leftarrow q.z \mid (i \;\&\; 4 \;?\; h : 0)$;

We will revisit the example from Figure 3.7 with the two quadrants and their coordinates in binary:

$$(q.x, q.y) = (10000, 10000) \text{ and } (o.x, o.y) = (10100, 10000).$$

The variable h in this case will again be `00100` and, since we are looking at the second child of q, $i = 1$ (`00001` in binary). Starting from the coordinates of q, we will use the operations described in Algorithm 2 to find $(o.x, o.y)$. First, we will look at the ternary operator in each *binary OR* (lines 3-4 of Algorithm 2):

- for the x coordinate, we have 00001 & 00001 = 1, which is true, giving h as a result.
- for the y coordinate, we have 00001 & 00010 = 0, which is false, giving 0 as a result.

The coordinates of the child then are:

$$\begin{cases} o.x = q.x \mid 00100 = 10000 \mid 00100 = 10100, \\ o.y = q.y \mid 00000 = 10000 \mid 00000 = 10000, \end{cases}$$

which are indeed the coordinates we have calculated before.

3.3 Parallel Algorithms

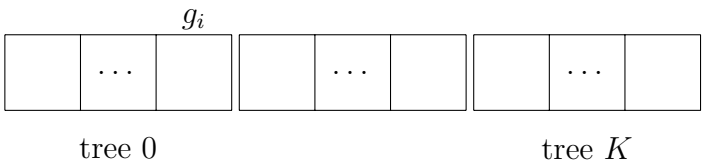


Figure 3.8: The entire forest is stored in a global linear array indexed by g_i, the global index of each octant.

We have seen until now how to store a tree in linear arrays using the z-index and how to traverse the tree knowing only its leaves. The set of all trees in a forest gives rise to a global linear array where each octant is identified by its global index g_i (see Figure 3.8).

In a parallel environment, the global array of octants has to be distributed between all the processes. In `p4est`, continuous chunks of this array are assigned to each process in increasing order of the process ID (see Figure 3.9).

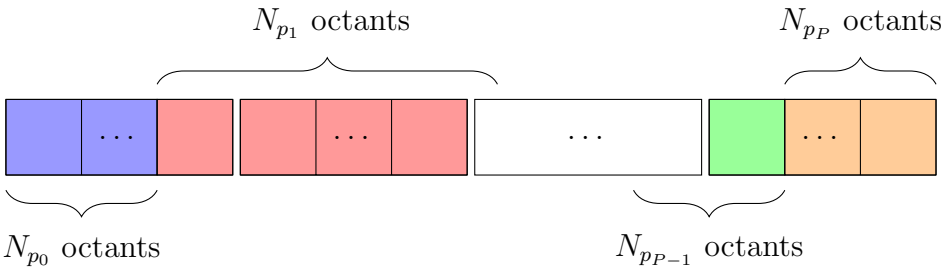


Figure 3.9: Partition between processes of the global linear array.

To ease the access to each individual octant, they are stored in per-tree arrays inside each process. This allows the distribution between processes of the global quadrant array to be completely defined by the triplet (N_p, k_p, o_p), where:

- N_p is the number of octants owned by process p.
- k_p is the first tree that has octants owned by process p.
- o_p is a descendant of level b of the first octant from the tree k_p owned by process p.

Remark. The octant o_p does not need to exist in the tree k_p. It is used as a marker to delimit the octants in the process p and those in process $p-1$, that belong to the tree k_p. This is possible because all the octants from the tree k_p in $p-1$ will have a smaller Morton index than o_p and all the octants in p will have a larger Morton index than o_p.

This information is necessary because a process may own any number of trees (including 0) and any number of octants from these trees. Since each process has this information, they can all compute the global octant count:

$$G = \sum_{p'=0}^{p-1} N_{p'}.$$

If a uniform partition is desired, each process q can also easily calculate the number of octants it has to have, given a total number of octants G:

$$N_p^{uniform} = \frac{G(p+1)}{P} - \frac{Gp}{P} \tag{3.1}$$

where P is the total number of processes.

In the next pages, we will look at some of the basic algorithms for building an efficient cell-based AMR library. These are: finding an element that does not belong to the current process, creating a new forest, refining and coarsening the new forest and doing uniform load balancing.

We have left out the more advanced algorithms that create the ghost layer, balance the forest and iterate over its elements. More information on these algorithms can be found in [17] (for ghost layer construction and advanced iteration) and [16] (for efficient 2:1 balancing).

Finding the Owners of an Octant

The first algorithm that we are going to look at is a basic building block of parallel AMR algorithms: finding the owner processes of a given octant. In [14], we have algorithms that allow easy ways to find the neighbors of an octant in the same tree and more advanced methods of accounting for inter-tree boundaries, but we also need to find to which process these neighboring octants belong.

Algorithm 3: Find the processes that own an octant.

Data: octant o, tree k

1 $\mathcal{P} \leftarrow \emptyset$;
2 **if** *o is outside the tree k across a corner c* **then**
3 $\quad \mathcal{CC} \leftarrow$ `corner_connections(k, c)`;
4 $\quad$ **for** $C \in \mathcal{CC}$ **do**
5 $\quad\quad o' \leftarrow$ `transform_corner(C, o, 1)`;
6 $\quad\quad$ Binary search in (k_p, o_p) for owner process p' of $(C.k, o')$;
7 $\quad\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{p'\}$;
8 $\quad$ **end**
9 **else if** *o is outside the tree k across an edge e* **then**
10 $\quad \mathcal{EC} \leftarrow$ `edge_connections(k, e)`;
11 $\quad$ **for** $E \in \mathcal{EC}$ **do**
12 $\quad\quad o' \leftarrow$ `transform_edge(e, E, o, 1)`;
13 $\quad\quad$ Binary search in (k_p, o_p) for owner process p' of $(E.k, o')$;
14 $\quad\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{p'\}$;
15 $\quad$ **end**
16 **else if** *o is outside the tree k across a face f connected to k'* **then**
17 $\quad o' \leftarrow$ `transform_face(k, f, o)`;
18 $\quad$ Binary search in (k_p, o_p) for owner process p' of (k', o');
19 $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{p'\}$;
20 **else**
21 $\quad$ Binary search in (k_p, o_p) for owner process p' of (k, o);
22 $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{p'\}$;

There are two cases that we will look at:

- The first one involves knowing the global index g_i of an octant o in the linear storage array and is the simple case. Given the triplets (N_p, k_p, o_p) for each process, the process

p that contains the octant o with the global index g_i satisfies:

$$\sum_{p'=0}^{p-1} N_{p'} \leq g_i \leq \sum_{p'=0}^{p} N_{p'}$$

- The second case is when we are only given the tree k and the octant o (and thus its coordinates and level). This case is described in Algorithm 3.

Finding an octant o given a tree k can be useful in many situations. One example is building a ghost layer. When building a ghost layer, we can construct the (x, y, z) coordinates of an octant that neighbors a local octant (see Algorithm 6 and 7 in [14]), but we do not know if it exists in the forest, what tree and what process it belongs to. All this information can be found using Algorithm 3.

The new family of functions `transform_*` have been defined in [14] and are functions that supersede the simple functions to find neighboring octants by taking into account the connectivity of the forest and give back the correct octant in the coordinates of the originating tree.

Similarly, the `*_connections` functions get all the octants that share a corner or edge with the given octant. These functions, however, only give non-redundant connections, meaning that for a corner connection it will not return octants that are also connected though a face or an edge.

Creating a new uniform forest

Algorithm 4: Creating a new forest.

Data: Minimum octants per process n_q, number of trees K

```
1  l ← ⌈log_2 ⌈max(Pn_q/K, 1)/d⌉⌉;                    /* mimimum level */
2  n ← 2^{dl};                                        /* octants per tree */
3  N ← nK;                                            /* global number of octants */
4  g_first ← ⌊Np/P⌋;                                  /* first octant in process */
5  g_last ← ⌊N(p+1)/P⌋;                               /* last octant in process */
6  k_first ← ⌊g_first/n⌋;                             /* first tree in process */
7  k_last ← ⌊g_last/n⌋;                               /* last tree in process */
8  for k ∈ {k_first, ..., k_last} do
9  |  m_first ← (k = k_first ? (g_first − nk) : 0);
10 |  m_last ← (k = k_last ? (g_last − nk) : n − 1);
11 |  O_k ← ∅;
12 |  for m ∈ {m_first, ..., m_last} do
   |  |  /* Octant computes (x, y, z) from the linear index */
13 |  |  O_k ← O_k ∪ Octant(l, m);
14 |  end
15 end
```

In Algorithm 4, we present a simple way of creating a new uniform forest. This algorithm is fairly straightforward. It first computes the number of octants to be created on the current process as well as the global index of the first octants in the process. Then it creates an array with the required number of octants and associates each one with its correct coordinates. The coordinates are computed from the global z-index (there is a bijection between the z-index and (x, y, z)). The coordinates of each octant can be computed directly from their position in the global linear array since all the trees are *uniform*.

`p4est` also provides a way to construct forests that are not uniform. The algorithms used for this are based on the work from [15] with additional improvements to reduce the algorithm to $\mathcal{O}(N)$ time complexity, where N is the global number of octants.

Adapting a forest

Once the forest is created, the next step is usually to refine and coarsen it in such a way that the initial condition is correctly represented. We will look here at the sketch of the refining and coarsening algorithms presented in [14].

A very important feature of the adapting algorithms is that they strive to maintain the z-order while modifying the linear arrays. This permits to only do one pass over all the octants and decide to refine or coarsen, without having to revisit them all and put them in the right order. This gives the algorithms a running time of $\mathcal{O}(N_p)$, but requires additional space, as we will see.

The first algorithm we will look at is the coarsening algorithm (see Algorithm 5). When coarsening, a family of 4 (resp. 8) quadrants (resp. octants) is replaced by their single parent.

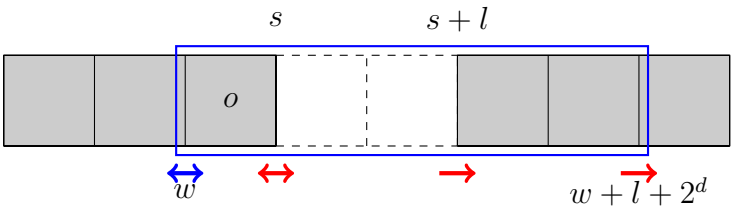


Figure 3.10: Coarsening the octant family starting with o.

To better understand the algorithm described in 5 we will look at Figure 3.10. As in the algorithm, we see our sliding window starting at w and then an empty range of size l starting at the index s. The first thing we will do is look at the octant o and see if it and the other $2^d - 1$ octants at the end of the sliding window form a family that wants to be coarsened.

Once we have a family that desires to be coarsened, the octant o at position w is replaced by its parent (this is done at line 10) and we grow the size of the empty space by $2^d - 1$ (corresponding to the other children) and advance the starting position w.

Algorithm 5: Coarsening the octants in a single process.

```
Data: Boolean recursive, function coarsen
1  for k ∈ T_local do
2      n ← #O_k;                                   /* octants in tree k */
       /* Start and size of the sliding window                          */
3      w ← 0;
4      C ← 2^d;
       /* Start and size of the empty range                             */
5      s ← 1;
6      l ← 0;
7      while w + C + l ≤ n do
8          c ← n;
9          if is_family (w, s, l) and coarsen(k, w, s, l) then
10             O_k(w) ← Parent (O_k(w));
11             c ← child_id (O_k(w));
12             s ← w + 1;
13             l ← l + C − 1;
14         end
15         if c ≤ w and recursive then
16             w ← w − c;                          /* move window backward */
17         else
18             w ← w + 1;                          /* advance window */
19             if w = s and s + l < n then
20                 O_k(s) ← O_k(s + l);
21                 s ← w + l;
22             end
23         end
24     end
25     O_k(s, ..., n − l − 1) ← O_k(s + l, ..., n − 1);
26 end
```

If instead, the octant o and the others do not form a family, we simply advance the starting position w and copy into the new empty space an octant from the end of the empty range, from position $s + l$ (line 20). Additional care is given to recursive coarsening when we can go back and decide if there is a family that is formed by the parent of o and if it desires to be coarsened as well.

Recursive coarsening is not usually desired because, during the procedure, we cannot correctly apply interpolation algorithms and cannot rely on the ever-changing mesh and linear array to get neighborhood information. It is however desired while trying to define the initial conditions from physical data known in advance.

The refinement algorithm is described by Figure 3.11. When refining, we will look at an octant o and see if it desires to be refined. Once that is established, 2^d children are created to take its place.

To achieve refining in a single pass, an additional linked list is used to store the octants that have to be inserted. Briefly, the algorithm goes as follows:

- If an octant o wants to be refined, it is popped from the linear array and its 2^d children are prepended to the linked list.
- If instead it does not want to be refined, it is popped from the linear array and appended to the linked list.
- The first item in the linked list is then added in the place of the popped octant.

If we consider the number of octants that want to be refined N_r, then the size of the linked list will be at most $2^d N_r$ at a time.

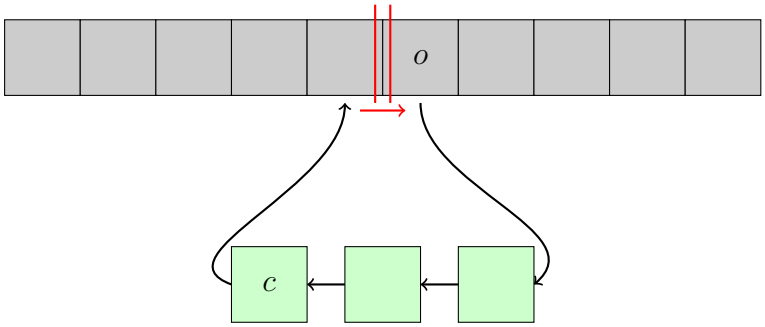


Figure 3.11: Refining the octant q and adding its 4 children.

Partitioning a forest

The last algorithm we will look at is the **load balancing** or partitioning algorithm. This algorithm will allow the forest to have an almost equal number of octants in each process and thus distribute the work that needs to be done in a fair fashion between them. In [14], there is a more general algorithm that allows assigning weights to each octant and doing a partition based on equal weights. We will instead look here at a simple algorithm that assumes each octant has the same weight.

Algorithm 6: Partition forest for equal weights.

Data: N global number of octants
Data: P number of processes
1 **for** $p \leftarrow 0$ **to** P **do**
2 $\quad N_p \leftarrow \left\lfloor \frac{N(p+1)}{P} \right\rfloor - \left\lfloor \frac{Np}{P} \right\rfloor$;
3 **end**
4 `partition_given` (N_p) ; `/* exchange octants */`

The algorithm that describes the uniform partitioning is Algorithm 6. It makes use of an extra function called `partition_given` that, given the desired number of octants in each process,

will exchange octants at the beginning and end of the per-process arrays with neighboring processes so as to achieve the desired partition.

The benefit of this algorithm is that it does not require any sort of inter-process communication to compute the partitions on all existing processes. The calculations are simply based on (3.1).

3.4 Implementation and Data Structures

We will now try to give a high level view of how we have used `p4est` in our tests. `p4est` has quite extensive documentation in its source code (through Doxygen), as well as a few overviews on how the library is supposed to be used. These are all available with the source code on the projects's webpage [30].

The scheme we have used is a very simple one (can be seen in Figure 3.12). Advancing the solution in time happens in the **Advance** stage, while all the interpolations between refined and coarsened elements happen during the **Adapt** and **Balance** phases using `p4est` functions that require simple, per-octant callbacks.

We store the state vector in the octants themselves so that we can get easy access during the various stages of the simulation. This is contrary to the recommended way in the `p4est` documentation, that supposes the user of the library will have their own separate data storage schemes and will only use `p4est` to place them on the mesh and perform interpolations when necessary.

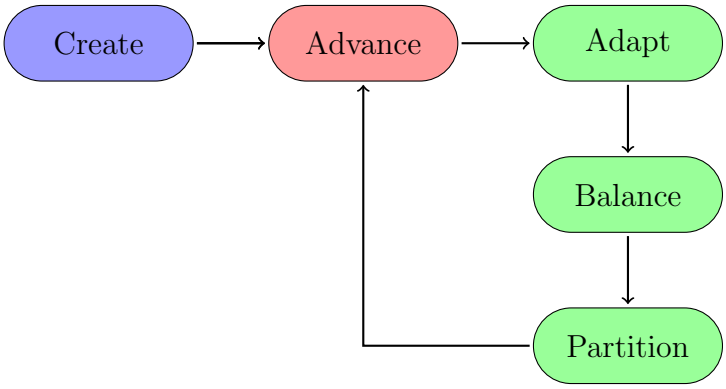


Figure 3.12: Overview of the application workflow.

3.4.1 Creating a Forest

We will now look in more detail at each of those steps and see what functions and data structure are necessary. In the first step, **Create**, we will create the connectivity, the forest and set up an initial condition for our equations.

The base element of p4est is the quadrant or octant. These two data structures are defined in p4est.h and p8est.h and are called p4est_quadrant_t and p8est_quadrant_t, respectively. The 4 and the 8, of course, refer to the quadtree and octree configurations and all p4est_* functions have a p8est_* variant.

The quadrant described in Code 3.1 contains all the fields we would expect from the descriptions we have seen before: the coordinates (x, y, z) and the level. It also contains a special user_data field that can be used by the client of the library to store its own per-quadrant information, which we have used to store our state vectors.

A few new types are introduced as well, these are: p4est_qcoord_t, p4est_topidx_t and p4est_locidx_t and are used for coordinates, tree indexes and local quadrant indexes, respectively. They are all integer types.

Code 3.1: The data structure describing a quadrant.

```
typedef struct p4est_quadrant
{
  p4est_qcoord_t x, y, z;
  int8_t level, pad8;
  int16_t pad16;
  union p4est_quadrant_data
  {
    void *user_data;
    /* ... */
    struct
    {
      p4est_topidx_t which_tree;
      p4est_locidx_t local_num;
    }
    piggy3;
  }
  p;
}
p4est_quadrant_t;
```

A tree structure is described in Code 3.2. Each local tree contains a list of local quadrants in the quadrants array. This list does not contain all the quadrants in the tree, just the local ones starting at quadrants_offset.

Next we have the connectivity in Code 3.3. As seen before, it contains:

- vertices: a list of physical points.
- tree_to_vertex: an array $\mathcal{NV}$ tying each point to a corner of each tree.
- tree_to_tree: an array $\mathcal{NO}$ specifying, for each face f, the tree on the other side.

Code 3.2: The data structure describing a tree.

```
typedef struct p4est_tree
{
  sc_array_t quadrants;
  p4est_quadrant_t first_desc, last_desc;
  p4est_locidx_t quadrants_offset;
  p4est_locidx_t quadrants_per_level[P4EST_MAXLEVEL + 1];
  int8_t maxlevel;
}
p4est_tree_t;
```

- `tree_to_face`: an array $\mathcal{NF}$ specifying, for each face f, the face in the neighboring tree and its orientation.

The corner connectivity is split between multiple arrays. First we have the `tree_to_corner` array that specifies, for each tree and each corner, the type of the corner if it connects two (or more) trees, or -1 if it does not.

Code 3.3: The data structure describing the connectivity.

```
typedef struct p4est_connectivity
{
  p4est_topidx_t      num_vertices;
  p4est_topidx_t      num_trees;
  p4est_topidx_t      num_corners;
  double             *vertices;
  p4est_topidx_t     *tree_to_vertex;
  p4est_topidx_t     *tree_to_tree;
  int8_t             *tree_to_face;
  p4est_topidx_t     *tree_to_corner;
  p4est_topidx_t     *ctt_offset;
  p4est_topidx_t     *corner_to_tree;
  int8_t             *corner_to_corner;
}
p4est_connectivity_t;
```

One important fact about how `p4est` stores the corner connectivity is that it is based on *types* of corners. In our example, we had the two middle corners in both trees in Figure 3.5 that connected multiple trees, but in fact they both represented just a single *type* since they connected the same trees in the same order.

The rest of the arrays describe, for each type of corner, the trees that it connects and the corners in each of those trees that meet there.

The connectivity is initialized using the p4est_connectivity_new function that returns the structure with all the arrays allocated to specified sizes which the user then has to fill in. A few widely used connectivities have already been implemented, like the unit cube with simple and periodic boundaries, a disk and even exotic structures such as the Möbius strip.

Code 3.4: The data structure describing the complete forest.

```
typedef struct p4est
{
  /* ... */
  void *user_pointer;

  p4est_topidx_t first_local_tree;
  p4est_topidx_t last_local_tree;

  p4est_locidx_t local_num_quadrants;
  p4est_gloidx_t global_num_quadrants;
  p4est_gloidx_t *global_first_quadrant;

  /* ... */
  p4est_connectivity_t *connectivity;
  sc_array_t *trees;

  /* ... */
}
p4est_t;
```

The final data structure we will consider at at this stage is the main p4est_t structure that contains all the other data structures and defines the forest completely.

Code 3.5: The function prototypes for initializing a forest and a quadrant.

```
p4est_t *p4est_new (sc_MPI_Comm mpicomm,
                    p4est_connectivity_t * connectivity,
                    size_t data_size,
                    p4est_init_t init_fn, void *user_pointer);

void     quadrant_init (p4est_t * p4est,
                        p4est_topidx_t which_tree,
                        p4est_quadrant_t * quadrant);
```

We can see in Code 3.4, that all the information we have mentioned before while detailing the algorithms and describing the connectivity is contained in this structure. A significant field

is the `user_pointer` that gives the user the possibility of adding their own data structures that will then be available whereever the `p4est_t` object is available.

The `p4est_t` data structure is initialized with the functions from Code 3.5. The constructor takes as arguments the data size of the per-quadrant user data (which can be 0), that is then completely handled by `p4est`, as well as an initializer function that will be called for each quadrant to initialize its user data (by default it is initialized to `NaN`).

3.4.2 The Ghost Layer

Another very important part of a numerical simulation, especially in parallel, involves the ghost layer. Using the ghost layer, we can set boundary conditions for our domain and also communicate with the neighboring processors to get the necessary data to make our calculations.

`p4est` handles creating the ghost layer and exchanging data between processes in an almost invisible manner to the user if the data is stored inside the `user_data` pointer of the quadrants.

Code 3.6: The ghost layer data structure and the functions that perform the data exchange.

```
typedef struct
{
  /* ... */
  sc_array_t ghosts;
  sc_array_t mirrors;
  /* ... */
}
p4est_ghost_t;

void p4est_ghost_exchange_data (p4est_t * p4est, p4est_ghost_t * ghost,
                                void *ghost_data);

void p4est_ghost_exchange_custom (p4est_t * p4est, p4est_ghost_t * ghost,
                                  size_t data_size, void **mirror_data,
                                  void *ghost_data);
```

The ghost layer contains a lot of information about what quadrants are ghosts, to which tree and process they belong, etc., but we are mostly interested in the `ghosts` and `mirrors` (see Code 3.6). The `ghosts` array is a list of all the quadrants that are ghosts for the current processor. Unlike local quadrants, they do not contain the user data from the foreign process, but they do contain information about their tree of origin and index in that tree (see Code 3.1). The user data has to be retrieved using the `p4est_ghost_exchange_data` function.

The mirrors array contains local quadrants that are ghosts for at least one other process. This array is very useful when trying to send per-quadrant custom data to other processes as it permits each process to know what the quadrants that it needs to send are and what quadrants it has to receive from other processes.

3.4.3 Iterating Over the Octants

Once the forest and the ghost layer are created, we can continue to the next stage of our simulation: **Advance**. In this stage, we will iterate over all the local quadrants in the forest and their neighbors to reconstruct whatever information we may need (node-based matrices for Finite Elements, fluxes for Finite Volumes, etc).

Code 3.7: General iterator function.

```
void p4est_iterate (p4est_t * p4est,
                    p4est_ghost_t * ghost_layer,
                    void *user_data,
                    p4est_iter_volume_t iter_volume,
                    p4est_iter_face_t iter_face,
                    /* p8est_iter_edge_t iter_edge, */
                    p4est_iter_corner_t iter_corner);
```

Iteration in p4est is done using the p4est_iterate function (see Code 3.7). This one function allows the user to iterate over octants, unique faces, edge and corners using a system of callbacks. For more information about how it achieves all this, see [17]. We will focus here on presenting how to use this function and what all the structures involved do.

Code 3.8: Type definitions for the corner, edge, face and octant callbacks.

```
typedef void        (*p4est_iter_*_t) (p4est_iter_*_info_t * info,
                                       void *user_data);
```

First, we have the 4 types of callbacks that are defined in Code 3.8. Each of the callbacks takes a different data structure as an argument that describes the neighborhood of that type of element. For a volume, we get the information specified in Code 3.9. This data is probably sufficient in most cases; it provides the quadrant with its coordinates as well as the index in the local linear array and the tree of origin. With this we can construct the neighborhood of the octant and traverse the whole tree.

A **face callback** takes a different type of structure, defined in Code 3.10. A face is usually defined by the octants on each side, but because of the non-conforming structure of the mesh, any of these two sides can contain a refined octant. p4est has made the choice, that while iterating over faces and finding an octant that shares a face with two other octants

Code 3.9: Data structure for an octant iterator.

```
typedef struct p4est_iter_volume_info
{
  p4est_t            *p4est;
  p4est_ghost_t      *ghost_layer;
  p4est_quadrant_t   *quad;
  p4est_locidx_t      quadid;
  p4est_topidx_t      treeid;
}
p4est_iter_volume_info_t;
```

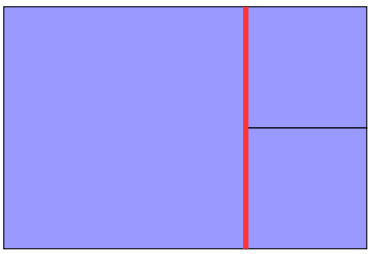


Figure 3.13: A face (marked in red) between quadrants of different levels.

(refined), it will take into account the whole face of the bigger octant instead of taking only half into account with each half-neighbor. We can see this in Figure 3.13: the face is marked in red.

The data structure defined in 3.10 allows us to look at the neighborhood of a face that can be either *hanging* (with two quadrants on one side) or *full* (with only a single quadrant on each side). Additional information is offered about the quadrants, such as their local index, their coordinates and whether they are part of the ghost layer or not.

The corresponding corner and edge data structures are very similar. They both contain the surrounding neighborhood of quadrants that can be used to compute different values on those elements.

3.4.4 Adapting and Load Balancing

Last but not least, we will look at how refining, balancing and partitioning is done in `p4est`. These operations are hidden from the user inside the functions listed in Code 3.11. The only variables that have to be defined by the user are the callback functions that tell `p4est` whether it should refine or not and, in the case of balancing as well, a function that will handle the interpolation between parent and children.

The refining function and the coarsening function have very similar arguments. They both have the option to perform their actions recursively and can initialize newly created quadrants

Code 3.10: Data structure for a face iterator.

```
typedef struct p4est_iter_face_side
{
  p4est_topidx_t      treeid;
  int8_t              face;
  int8_t              is_hanging
  union p4est_iter_face_side_data
  {
    struct
    {
      int8_t              is_ghost;
      p4est_quadrant_t   *quad;
      p4est_locidx_t      quadid;
    }
    full;
    struct
    {
      int8_t              is_ghost[2];
      p4est_quadrant_t   *quad[2];
      p4est_locidx_t      quadid[2];
    }
    hanging;
  }
  is;
}
p4est_iter_face_side_t;
```

with the `init_fn` function.

The balancing function has one special option that is the type of balance it should perform. In 2D, it can balance the forest according to corners or faces and in 3D it can also do it according to edges. The balancing operation is related to the level of neighboring quadrants and constrains the neighbors to not differ by more than one level.

A very important function for both adapting the forest and balancing it is the replacement function, the prototype of which is defined in Code 3.12. This function receives as arguments:

- when refining, the parent quadrant that has been popped out of the array and the 4 (resp. 8) quadrants (resp. octants) that are its children. It is then the job of the user to specify how the information is interpolated from one to the others.
- when coarsening, the inverse is true: we receive the 4 (resp. 8) children that form a family that desires to be coarsened and the new quadrant (resp. octant) that is their parent.

Code 3.11: Functions for adapting and partitioning the forest.

```
void p4est_refine_ext (p4est_t * p4est,
                       int refine_recursive, int maxlevel,
                       p4est_refine_t refine_fn,
                       p4est_init_t init_fn,
                       p4est_replace_t replace_fn);

void p4est_coarsen_ext (p4est_t * p4est, int coarsen_recursive,
                        int callback_orphans,
                        p4est_coarsen_t coarsen_fn,
                        p4est_init_t init_fn,
                        p4est_replace_t replace_fn);

void p4est_balance_ext (p4est_t * p4est,
                        p4est_connect_type_t btype,
                        p4est_init_t init_fn,
                        p4est_replace_t replace_fn);

p4est_gloidx_t p4est_partition_ext (p4est_t * p4est,
                                    int partition_for_coarsening,
                                    p4est_weight_t weight_fn);
```

As we can see, this function does not give any information about the neighborhood of the quadrants that are going to be refined or coarsened, thus greatly limiting the type of interpolations we can do. This is one of the reasons why the `p4est` documentation recommends storing the data ourselves and keeping two versions of the forest: before and after the adaptation, so that we can have complete liberty in how we do the interpolations.

Code 3.12: Prototype for the user defined interpolation function.

```
typedef void (*p4est_replace_t) (p4est_t * p4est,
                                 p4est_topidx_t which_tree,
                                 int num_outgoing,
                                 p4est_quadrant_t * outgoing[],
                                 int num_incoming,
                                 p4est_quadrant_t * incoming[]);
```

3.5 Numerical Results

In this section we will look at some of the numerical results that we have obtained using the `p4est` library.

All the examples that we are about to see have been made using a newly developed code that is available at [31]. Since the present work has started, the p4est developers have released a new version of the library that contains small examples of equations solved using the Finite Element or Finite Volume methods that can also prove invaluable to someone starting out.

Our current code has been developed for numerical simulations that apply the Finite Volume method to different sets of equations. The Finite Volume example in the p4est source code deals with the transport equation, an equation we will also be looking at, in a very basic way. The code we have developed has many extra features:

- Easy way to define initial conditions and choose them at runtime.
- A similar mechanism for boundary conditions. p4est only has periodic and homogeneous Neuman boundary conditions, we have also implemented Dirichlet boundary conditions, walls, etc.
- More advanced error indicators that look at the gradients of different variables.
- Parallel output using the HDF5 library and the XDMF standard.
- etc.

Although there are many tests that could be executed, especially for the more complex two-phase model, that put pressure on the p4est library and the refinement criteria, we have chosen a few that accentuate the quality of the adapted mesh and the way that p4est works in different stages of a simulation.

We will also try to present some very preliminary scalability and performance results for our code and especially for the p4est library. More extensive performance tests of the p4est library can be found in [14], [16] or [17].

3.5.1 Forest Creation, Adaptation and Partitioning

This first suite of tests is going to be targeted specifically at the p4est library and the various functionality that it offers us, such as mesh creation, load balancing, etc.

To test this functionality we have chosen to use a Finite Volume scheme implemented for the transport equation. We will not, at this point, focus on the numerical results, which are available in the upcoming chapters. For the sake of completeness, we will give the parameters used to the simulations. The initial condition for the test is an indicator function in the shape of a disk:

$$c_0(\mathbf{x}) = \begin{cases} 1, & \sqrt{(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2} \leq r, \\ 0, & \text{otherwise}, \end{cases}$$

where (x_c, y_c, z_c) is the center of a sphere of radius r. In our case, we have chosen center $(0.2, 0.5, 0.5)$ and the radius 0.1. The advection velocity is simply $\mathbf{u} = (1, 0, 0)$. We have chosen a final time $T = 1$ and the Courant number $\theta = 1.0$. The boundary conditions on this test are **periodic** in all directions.

On the AMR side, we have limited the maximum and minimum possible level of a quadrant to the interval $[\![4, 7]\!]$ and ran the program on 8 different processes. `p4est` supports a minimum level of 0 and a maximum level of 32. This is mostly a shortcoming of the programming language it is written in, namely `C`, which does not natively support arbitrary sized integers.

The refining criterion that we have used here is a simple normalized gradient estimator on the transported value c. We define the following estimator for the gradient between two neighboring cells:

$$\xi_{ij}^g = \frac{|c_i^n - c_j^n|}{\max(c_i^n, c_j^n)}.$$

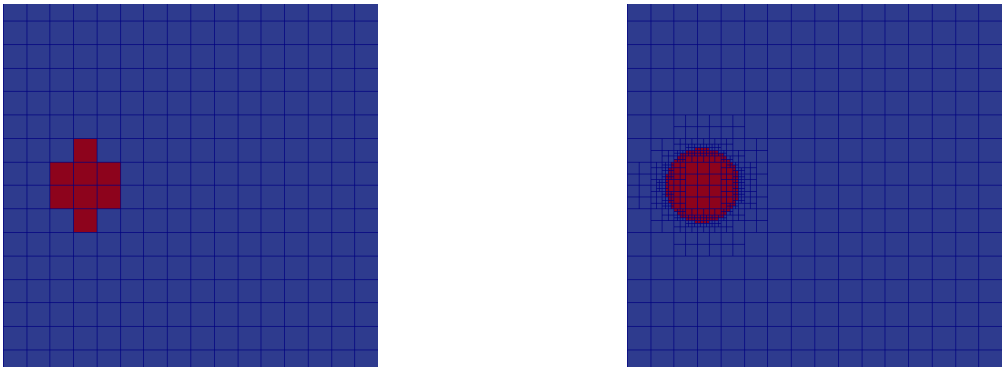
The quadrant is then refined if ξ_i^g is bigger than a certain threshold ϵ, giving the final indicator:

$$\xi_i = \begin{cases} 1, & \xi_i^g > \epsilon, \\ 0, & \xi_i^g < \epsilon, \end{cases}$$

where the value of ϵ is user defined. In our tests we have taken various values for ϵ: in the case of the transport equation, the value of epsilon was set to $\epsilon = 0.1$. These thresholds are *very* specific to the test case at hand.

Creating the Mesh

The first algorithm we will look at is the creation of the forest. We can see an unrefined initial condition in Figure 3.14a that we will then adapt to obtain the new, more exact solution from Figure 3.14b.



(a) (b)

Figure 3.14: (a) Initial solution on the uniform mesh of level 4. (b) Initial solution on a refined mesh.

We can see in Figure 3.14 that the initial solution is correctly refined and the disk gets properly represented. In Figure 3.15, we can see that the 3D case is handled just as easily and the solution gets correctly refined.

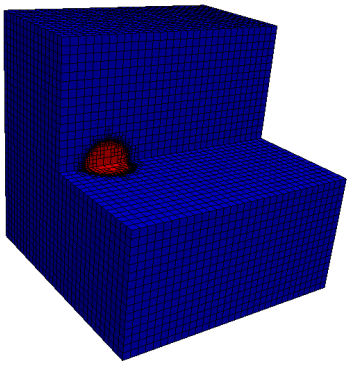
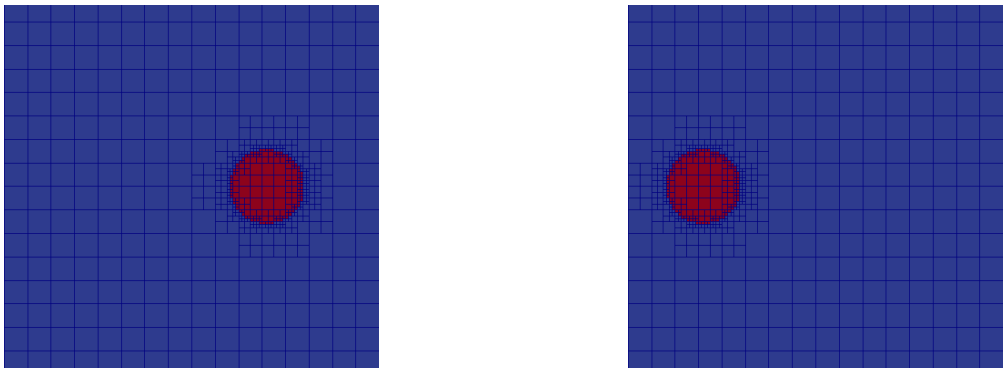


Figure 3.15: Initial solution on a refined 3D mesh.

Adapting the Mesh

Next we will look at how the mesh gets adapted as the solution advances. In the test case that we have chosen, we know that, on a uniform mesh, the solution is exact, so we expect the same results on the adapted mesh.



(a) (b)

Figure 3.16: (a) Mesh and solution at $t = 0.5$. (b) Mesh and solution at $t = 1.0$.

We can see in Figure 3.16a and Figure 3.16b that the refinement follows the solution exactly, to such a degree that no cell is different. We can also see that, even though the mesh is refined and the number of cells is much smaller, we still get the exact solution to this transport equation.

Again, in Figure 3.17, we can see that the solution gets correctly transported and refined during the 3D simulation, just like in the 2D case.

Partitioning the Mesh

In Figure 3.18, we can also see the results of doing load balancing on the mesh. The load balancing in p4est evolves in time with the mesh and always tries to distribute the number of quadrants equally between the processes.

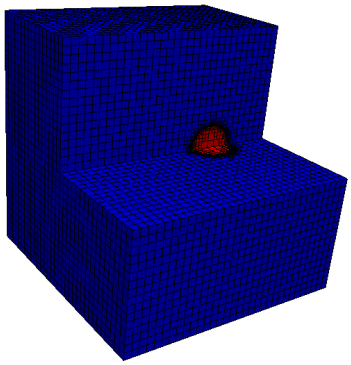
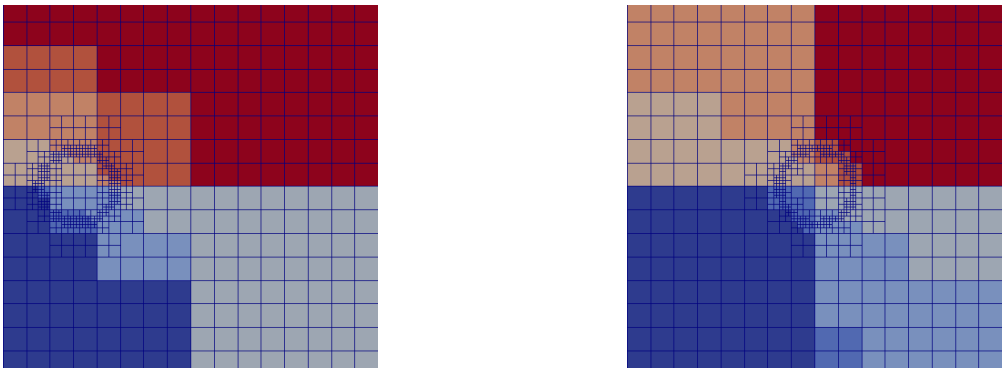


Figure 3.17: Refined solution on a refined 3D mesh at an intermediary time step.



(a) (b)

Figure 3.18: (a) Partition at $t = 0.0$ (b) Partition at $t = 0.3$.

In Figure 3.18 we can see the distribution between the 8 processes that we have ran this simulation on. In the first figure, we have the initial load balancing of the mesh and then, in the second one, we have the partitioning in the middle of the simulation.

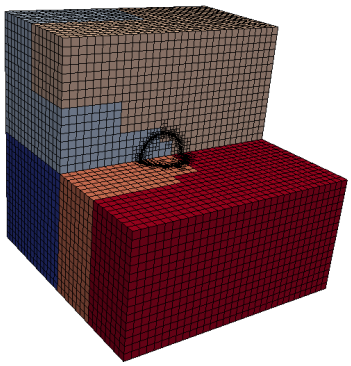


Figure 3.19: Load-balanced partition of the quadrants on 8 processes using a refined 3D mesh.

Since the *z-index* works equally well in 3D, the partitioning can also take place in this case with the same algorithms. We can see in Figure 3.19 the partitions of processes 0 to 5 out of the total 8 at an intermediary step of the simulation.

4 Discretization of the Transport Equation using `p4est`

In this chapter we present a discretization strategy for the transport equation within the p4est AMR framework using a classic *Finite Volume* approach.

We consider a domain Ω of dimension $d \in \{2, 3\}$ and the following *Initial Value Boundary Problem*:

$$\begin{cases} \partial_t c + \mathbf{u} \cdot \nabla c = 0, \\ c(0, \mathbf{x}) = c_0(\mathbf{x}), \end{cases} \tag{4.1}$$

where c is the transported value, $\mathbf{u}$ is a given **smooth velocity field** and $c_0(\mathbf{x})$ is the initial condition. These functions can be described as follows:

$$\begin{aligned} c \quad &: \quad [0,T] \times \Omega \quad \longrightarrow \quad \mathbb{R} \\ &\qquad\quad (t, \mathbf{x}) \quad \longmapsto \quad c(t, \mathbf{x}), \\ \mathbf{u} = (u, v, w) \quad &: \quad [0,T] \times \Omega \quad \longrightarrow \quad \mathbb{R}^d \\ &\qquad\quad (t, \mathbf{x}) \quad \longmapsto \quad \mathbf{u}(t, \mathbf{x}), \\ c_0 \quad &: \quad \Omega \quad \longrightarrow \quad \mathbb{R} \\ &\qquad\quad \mathbf{x} \quad \longmapsto \quad c_0(\mathbf{x}). \end{aligned}$$

The domain Ω is discretized and covered by octants that do not overlap, but are allowed to be non-conforming due to the constraints of the adapted mesh (see Figure 4.1).

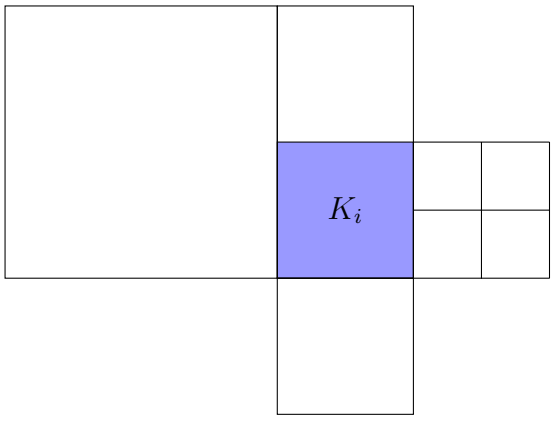


Figure 4.1: A quadrant with its refined and coarsened face neighbors.

Given the heterogeneous form of the neighborhood of an octant, we have chosen to use notations adapted to non-structured meshes. Let:

- K_i be a cell in the mesh, the index i can be considered as the global z-index of the octant.
- $l_i \in [\![0, b]\!]$ be the level of the cell K_i.
- l_{max} be the level of the most refined octant in the mesh and l_{min} be the level of the most coarsened octant in the mesh at a time t_n.
- $\Delta x_i = \Delta y_i = \Delta z_i$ be the size of a cell K_i.
- $|K_i| = \Delta x_i^d$ be the area (resp. volume) in 2D (resp. 3D) of the cell K_i.
- K_j be a neighbor of K_i of level $l_j \in \{l-1, l, l+1\}$.
- $|\Gamma_{ij}|$ be the length (resp. area) of the face between K_i and K_j.
- n_{ij} be the orientation of the normal on the face Γ_{ij} with respect to direction of the axis that is orthogonal to Γ_{ij}:

$$n_{ij} = \begin{cases} -1, & K_j \text{ is to the left of } K_i, \\ 1, & K_j \text{ is to the right of } K_i. \end{cases}$$

- $\mathcal{N}_x(i) = \{\, j \mid K_j \text{ is a neighbor in the } x \text{ direction} \,\}$. This set can contain neighbors of any level, so we do not know its cardinality, but we know that $2 \leq |\mathcal{N}_x(i)| \leq 4$ in 2D (resp. $2 \leq |\mathcal{N}_x(i)| \leq 8$ in 3D). $\mathcal{N}_y(i)$ and $\mathcal{N}_z(i)$ are defined similarly for the y and z directions.

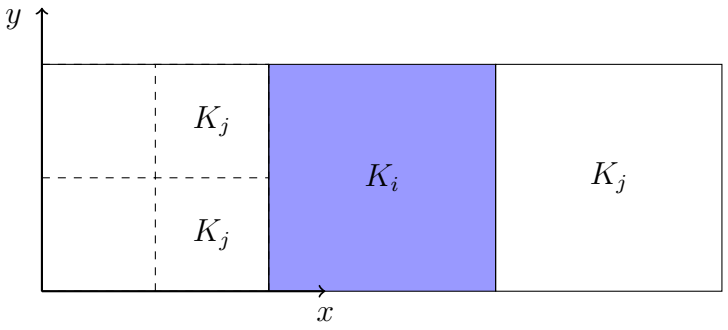


Figure 4.2: Left and right, potentially refined, neighbors of a quadrant.

Remark. In the context of AMR, where neighbors can have different sizes (see Figure 4.2), what we call the face between two octants K_i and K_j can be only a portion of a face of K_i. For this reason, the length (resp. area) $|\Gamma_{ij}|$ of the face Γ_{ij} is given by:

$$|\Gamma_{ij}| = \beta_{ij} \Delta x_i^{d-1},$$

where

$$\beta_{ij} = \begin{cases} 2^{1-d}, & l_j > l_i, \\ 1, & l_j \leq l_i. \end{cases}$$

Remark. $\Delta x_i = \Delta y_i = \Delta z_i$ is a hypothesis we have made for the sake of simplicity. `p4est` has support for more complex geometries that would require the computation of each length. In this simple case, we know that each tree is represented by a domain that is the $[0,1]^d$ cube and each length can be computed using only the level of the octant K_i:

$$\Delta x_i = 2^{-l_i}.$$

4.1 Dimensional Splitting

We choose to adopt a **dimensional splitting** strategy for discretizing (4.1).

For the case of (4.1) using dimensional splitting consists in successively approximating (4.2), (4.3) and (4.4) that ultimately gives an update formula for the values in each octant.

$$\begin{cases} \partial_t c + u\partial_x c = 0, & (4.2)\\ \partial_t c + v\partial_y c = 0, & (4.3)\\ \partial_t c + w\partial_z c = 0. & (4.4) \end{cases}$$

We refer the reader to [27, p. 543] for more details about dimensional splitting. We are left with providing a discretization for a one-dimensional problem that will be successively applied to (4.2), (4.3) and (4.4).

Let us briefly recall a possible guideline for obtaining a Finite Volume approximation of (4.2). Suppose for the sake of simplicity that c is smooth and verifies (4.2), then we have:

$$\partial_t c + u\partial_x c = \partial_t c + \partial_x(uc) - c\,\partial_x u = 0.$$

By integrating over $[t_n, t_{n+1}] \times K_i$ we obtain:

$$\int_{t_n}^{t_{n+1}} \int_{K_i} \partial_t c \,\mathrm{d}\mathbf{x}\,\mathrm{d}t + \int_{t_n}^{t_{n+1}} \int_{K_i} \partial_x(uc)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t - \int_{t_n}^{t_{n+1}} \int_{K_i} c\,\partial_x u\,\mathrm{d}\mathbf{x}\,\mathrm{d}t = 0,$$

namely:

$$\begin{aligned} 0 \quad = \quad & \int_{K_i} (c(t_{n+1},\mathbf{x}) - c(t_n,\mathbf{x}))\,\mathrm{d}\mathbf{x} \;\; + \\ & \sum_{j\in\mathcal{N}_x(i)} n_{ij} \int_{t_n}^{t_{n+1}} \int_{\Gamma_{ij}} (uc)(t,\mathbf{x})\,\mathrm{d}\sigma\,\mathrm{d}t \;\; - \\ & \int_{t_n}^{t_{n+1}} \int_{K_i} c(t,\mathbf{x}), \partial_x u(t,\mathbf{x})\,\mathrm{d}\mathbf{x}\,\mathrm{d}t. \end{aligned} \tag{4.5}$$

We note:

$$\begin{aligned} c_i^n \quad & \text{an approximation of} \quad \frac{1}{|K_i|}\int_{K_i} c(t_n,\mathbf{x})\,\mathrm{d}\mathbf{x},\\ u_{ij}^n \quad & \text{an approximation of} \quad \frac{1}{\Delta t|\Gamma_{ij}|}\int_{t_n}^{t_{n+1}}\int_{\Gamma_{ij}} u(t,\mathbf{x})\,\mathrm{d}\sigma\,\mathrm{d}t,\\ u_{ij}^n c_{ij}^n \quad & \text{an approximation of} \quad \frac{1}{\Delta t|\Gamma_{ij}|}\int_{t_n}^{t_{n+1}}\int_{\Gamma_{ij}} (uc)(t,\mathbf{x})\,\mathrm{d}\sigma\,\mathrm{d}t, \end{aligned}$$

where c_{ij}^n is the numerical flux of the discretization. The Finite Volume approximation we consider consists in setting a recursive update formula for c_i^n that is a discrete equivalent of (4.5). Therefore we require that:

$$c_i^{n+1} = c_i^n + \frac{\Delta t}{|K_i|} \sum_{j \in \mathcal{N}_x(i)} |\Gamma_{ij}| \; u_{ij}^n \; n_{ij} (c_i^n - c_{ij}^n). \tag{4.6}$$

Remark. In the context of dimensional splitting, c_i^n is not the solution at time t_n because intermediate values are computed for each dimension. This is an abuse of notation that we have also used in the case of c_i^{n+1}.

Remark. The update formula can be simplified, using the exact formulas of $|K_i|$ and $|\Gamma_{ij}|$, to:

$$c_i^{n+1} = c_i^n + \frac{\Delta t}{\Delta x_i} \sum_{j \in \mathcal{N}_x(i)} \beta_{ij} \; u_{ij}^n \; n_{ij} (c_i^n - c_{ij}^n). \tag{4.7}$$

The update formula (4.6) for the one-dimensional equation (4.2) can be obtained for deriving update relations that provides an approximation of (4.3) and (4.4). Finally the approximation of the solution of (4.8) from t_n to t_{n+1}, is obtained by performing the three following update steps:

$$\begin{cases} c_i^* = c_i^n + \dfrac{\Delta t}{|K_i|} \displaystyle\sum_{j \in \mathcal{N}_x(i)} |\Gamma_{ij}| \; u_{ij}^n \; n_{ij} (c_i^n - c_{ij}^n), \\ c_i^{**} = c_i^* + \dfrac{\Delta t}{|K_i|} \displaystyle\sum_{j \in \mathcal{N}_y(i)} |\Gamma_{ij}| \; u_{ij}^n \; n_{ij} (c_i^* - c_{ij}^n), \\ c_i^{n+1} = c_i^{**} + \dfrac{\Delta t}{|K_i|} \displaystyle\sum_{j \in \mathcal{N}_z(i)} |\Gamma_{ij}| \; u_{ij}^n \; n_{ij} (c_i^{**} - c_{ij}^n). \end{cases} \tag{4.8}$$

The scheme (4.8) is a *first order* dimensional splitting scheme. Other splitting strategies like the Strang splitting technique [18] can allow to obtain *second order* accuracy.

4.2 Flux Choice: The Upwind Scheme

In order to complete the definition of the numerical scheme (4.8), we only need to propose a definition for the numerical flux c_{ij}^n. We choose here to adopt the simple upwind scheme, as follows:

$$c_{ij}^n = \begin{cases} c_i^n & \text{if } u_{ij}^n \, n_{ij} \geq 0, \\ c_j^n & \text{if } u_{ij}^n \, n_{ij} < 0. \end{cases} \tag{4.9}$$

The upwind flux, defined above, and the update formula (4.6) define a complete scheme that we can use to solve the transport equation.

Property 4.1. The scheme defined by (4.8) and (4.9) is stable under the Courant-Friedrichs-Lewy (CFL) condition:

$$\Delta t \max_{s\in\{x,y,z\}} \max_i \left(-\frac{1}{|K_i|} \sum_{\substack{j\in\mathcal{N}_s(i)\\ u_{ij}^n\, n_{ij}<0}} |\Gamma_{ij}| u_{ij}^n\, n_{ij} \right) \leq 1. \tag{4.10}$$

Proof. For the purpose of this proof, we will look at how to obtain a sufficient condition for stability in the x direction and then, by using similar techniques in the other directions, we will derive (4.10).

A sufficient condition for L^∞ stability can be obtained by imposing a maximum principle on the update from c_i^n to c_i^{n+1}. Such a maximum principle can be obtained by requiring that c_i^{n+1} is a convex combination of c_i^n and its downwind neighbors: c_j^n when j is such that $u_{ij}^n\, n_{ij} < 0$).

We recall the simplified update formula from (4.7):

$$c_i^{n+1} = c_i^n + \frac{\Delta t}{\Delta x_i} \sum_{j\in\mathcal{N}_x(i)} a_{ij}^n (c_i^n - c_{ij}^n),$$

where:

$$a_{ij}^n = \beta_{ij} u_{ij}^n n_{ij}.$$

By using (4.9), we see that the terms with $a_{ij}^n > 0$ cancel out. We are left with:

$$c_i^{n+1} = \left(1 + \frac{\Delta t}{\Delta x_i} \sum_{\substack{j\in\mathcal{N}_x(i)\\ a_{ij}^n<0}} a_{ij}^n \right) c_i^n - \frac{\Delta t}{\Delta x_i} \sum_{\substack{j\in\mathcal{N}_x(i)\\ a_{ij}^n<0}} a_{ij}^n c_j^n.$$

For a convex combination to be possible, we need to satisfy:

$$0 \leq 1 + \frac{\Delta t}{\Delta x_i} \sum_{\substack{j\in\mathcal{N}_x(i)\\ a_{ij}^n<0}} a_{ij}^n \leq 1 \tag{4.11}$$

and, for all $j \in \mathcal{N}_x(i)$ such that $a_{ij}^n < 0$,

$$0 \leq -\frac{\Delta t}{\Delta x_i} a_{ij}^n \leq 1. \tag{4.12}$$

Since (4.11) implies (4.12), we would only need to satisfy:

$$0 \leq -\frac{\Delta t}{\Delta x_i} \sum_{\substack{j\in\mathcal{N}_x(i)\\ a_{ij}^n<0}} a_{ij}^n \leq 1.$$

As $a_{ij}^n < 0$, a *sufficient* condition for stability in a cell K_i is:

$$-\frac{\Delta t}{\Delta x_i} \sum_{\substack{j \in \mathcal{N}_x(i) \\ a_{ij}^n < 0}} a_{ij}^n \leq 1.$$

The condition for all cells in the mesh is then:

$$\Delta t \max_i \left(-\frac{1}{\Delta x_i} \sum_{\substack{j \in \mathcal{N}_x(i) \\ a_{ij}^n < 0}} a_{ij}^n \right) \leq 1. \tag{4.13}$$

From (4.13), we can deduce a sufficient condition for stability in all directions:

$$\Delta t \max_{s \in \{x,y,z\}} \max_i \left(-\frac{1}{\Delta x_i} \sum_{\substack{j \in \mathcal{N}_s(i) \\ a_{ij}^n < 0}} a_{ij}^n \right) \leq 1. \tag{4.14}$$

As a result, we see that c_i^* is a convex combination of c_i^n and the value of its downwind neighbors, c_i^{**} is a convex combination of c_i^* and the value of its downwind neighbors and c_i^{n+1} is a convex combination of c_i^{**} and the value of its downwind neighbors. This implies that c_i^{n+1} is a convex combination of c_i^n and the value of its downwind neighbors at time t_n.

The convex combination that is imposed by (4.14) implies that a local maximum principle has been satisfied. By respecting the maximum principle, we can be sure that our scheme is stable, thus proving that (4.14), that is equivalent to (4.10), is sufficient for the stability of the scheme. q.e.d.

The CFL condition (4.10) allows us to derive a strategy for choosing a time step. We set:

$$\Delta t = \theta \frac{\Delta x}{\max\limits_{s \in \{x,y,z\}} \max\limits_i \left(-\sum_{\substack{j \in \mathcal{N}_s(i) \\ a_{ij}^n < 0}} a_{ij}^n \right)} \tag{4.15}$$

for a *Courant number* $\theta \in [0, 1]$ and a Δx that represents the size of the finest octant in the mesh, given by:

$$\Delta x = 2^{-l_{max}}.$$

4.3 Time Subcycling

We can imagine two different way of advancing the solution of our equation in time:

- By using the time step provided by (4.15) to advance all the cells at all the different levels at the same time. This algorithm is described by 7.

Algorithm 7: Advancing all levels in the x direction by Δt.

Data: Time step Δt

Initialize all c_i^* to c_i^n;
for *All octants* K_i **do**
  for *All* $j \in \mathcal{N}_x(i)$ **do**
    $c_i^* \leftarrow c_i^* + \frac{\Delta t |\Gamma_{ij}|}{|K_i|} u_{ij}^n n_{ij} (c_j^n - c_{ij}^n)$;
  end
end

- By advancing each level of the mesh with a different time step. This strategy is described by Algorithm 8 and 9.

To advance each level at its own pace, we have to define a per-level time step, which we choose to estimate by:

$$\Delta t_l = 2^{l_{max} - l} \Delta t, \tag{4.16}$$

by using the Δt estimation of the time step from (4.10) as a starting point. Since (4.10) gives us a time step that is equivalent to one computed on an uniform mesh of level l_{max}, we use it to estimate the time steps at coarser levels where the size of an octant grows with a factor of 2 in each dimension.

Remark. More advanced methods of computing the per-level time step can be used, but it is always necessary to synchronize them with the levels above. So, if we have a set of n_j time steps Δt_l^i at level l, the following condition has to be respected for a sub-time step at a higher level:

$$\Delta t_{l-1}^j = \sum_{i=1}^{n_j} \Delta t_l^i.$$

In Algorithm 8 and Algorithm 9 we explain how to do subcycling using a per-level time step as given by (4.16).

Algorithm 8 describes how to advance a single level in a specific direction (we have chosen x, but other directions are handled in a similar fashion). The general algorithm that advances *all* the octants of the mesh by $\Delta t_{min} = 2^{l_{max} - l_{min}} \Delta t$ in the x direction is 9. The final step is to repeat 9 for the y and z directions. Note that this is a valid method of advancing in time, but it is not equivalent to advancing all the quadrants using the same Δt.

Remark. The key difference between both methods is that after Algorithm 7, we have advanced by a very small Δt, more precisely Δt is $2^{l_{max} - l_{min}}$ times smaller then the one by which the solution is advanced by subcycling. This difference comes from the difference between the finest and the coarsest levels in the mesh and it is what makes the subcycling algorithm faster in some cases.

The recursive Algorithm 9 is called with the coarsest level in the mesh l_{min} and the time step corresponding for this level:

$$\Delta t_{l_{min}} = 2^{l_{max} - l_{min}} \Delta t.$$

Algorithm 8: Advancing level l in the x direction.

Data: Level l
Data: Time step at current level Δt_l

Initialize all c_i^* to c_i^n;
for *All octants* K_i *of level* l **do**
  for *All* $j \in \mathcal{N}_x(i)$ **do**
    if $l_j \leq l_i$ ***and*** *(K_j to the right of K_i* ***or*** *a boundary)* **then**
      $c_i^* \leftarrow c_i^* + \dfrac{\Delta t|\Gamma_{ij}|}{|K_i|} u_{ij}^n (c_j^n - c_{ij}^n)$;
      $c_j^* \leftarrow c_j^* - \dfrac{\Delta t|\Gamma_{ji}|}{|K_j|} u_{ij}^n (c_i^n - c_{ji}^n)$;
    end
    if $l_j = l_i - 1$ ***and*** *(K_j to the left of K_i* ***or*** *a boundary)* **then**
      $c_i^* \leftarrow c_i^* - \dfrac{\Delta t|\Gamma_{ij}|}{|K_i|} u_{ij}^n (c_j^n - c_{ij}^n)$;
      $c_j^* \leftarrow c_j^* + \dfrac{\Delta t|\Gamma_{ji}|}{|K_j|} u_{ji}^n (c_i^n - c_{ji}^n)$;
    end
  end
end
for *All ghost octants K_i of level l from other processes* **do**
  Let K_j be the local neighbor of K_i;
  if $l_j \leq l_i$ ***and*** *K_j to the right of K_i* **then**
    $c_j^* \leftarrow c_j^* - \dfrac{\Delta t|\Gamma_{ji}|}{|K_j|} u_{ji}^n (c_i^n - c_{ji}^n)$;
  end
  if $l_j = l_i - 1$ ***and*** *K_j to the left of K_i* **then**
    $c_j^* \leftarrow c_j^* + \dfrac{\Delta t|\Gamma_{ji}|}{|K_j|} u_{ji}^n (c_i^n - c_{ji}^n)$;
  end
end

Algorithm 8 is based on previous work from [10] and is very similar to the subcycling algorithm presented there. Two differences set it apart:

- Our proposed algorithm does not compute a *mean* of the values in a family of octants to pass it down to its parents. This is done because, in contrast to [10], where all the tree structure is stored, `p4est` only stores the leaves of the tree so it is not necessary to propagate the information upwards.

- We extend the algorithm to work in a distributed memory environment like the one offered by `p4est`. In `p4est` we have two types of ghost cells: classical ghost cells that are used for *boundary conditions* and ghost cells that are part of the mesh, but are not in the partition of the current process.

Algorithm 9: Advancing by Δt in the x direction.

Data: Level l, time step at level l

1 $\Delta t_{l+1} \leftarrow \frac{\Delta t_l}{2}$;
2 `subcycle` $(l+1, \Delta t_{l+1})$;
3 `subcycle` $(l+1, \Delta t_{l+1})$;
4 Exchange ghost layer with neighboring processes;
5 Advance level l in time by Δt_l using 8;

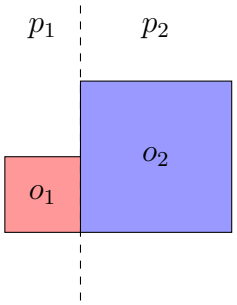


Figure 4.3: Two octants in different processes.

To deal with ghost cells from other processes we have added a second loop to Algorithm 8 to guarantee that all the fluxes at a certain level are correctly calculated. We can take the example from Figure 4.3 where we have two octants of level l and $l-1$ for o_1 and o_2, respectively, in two different processes where each is a face ghost cell of the other. The problem arises when we loop over all cells of levels l in process p_2 and do not, in fact, add the contribution from the ghost cell o_1. To fix this, we have chosen to also loop over all ghost cells and compute the correct interface flux with the local octants. Other solutions are also feasible, such as looping over local cells of level l and $l-1$ during one subcycle, exchanging the fluxes between processes, etc.

An important aspect of Algorithm 8, as of the one proposed in [10], is that it computes the left and right fluxes in such a way that the same flux is not computed twice when recursing between levels. We can see this in Figure 4.4.

Such a subcycling algorithm offers significant speedups because fewer time steps are required when compared to the straightforward time advancing scheme described in Algorithm 7 (coarse cells make big time steps). One drawback of such a scheme is the need to adapt the mesh at each sub-time step. We will see in the numerical results section that the solution will advance too quickly and go into coarser cells if we only adapt once before starting the subcycling procedure, thus leading to unnecessary diffusion and loss of precision.

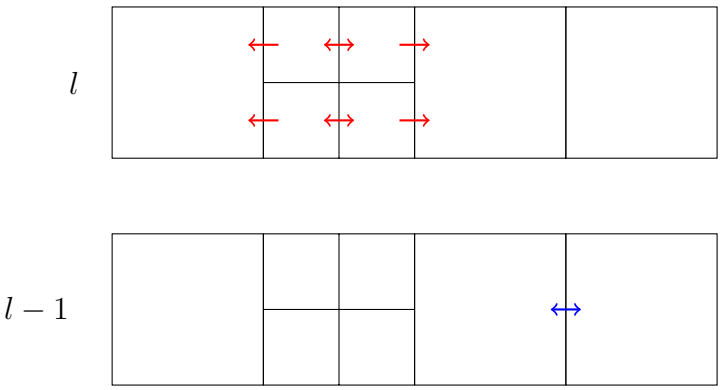


Figure 4.4: Flux computations at level l (in red) and level $l-1$ (in blue).

4.4 Numerical Results

In this section we will look at several numerical results we have obtained for the transport equation using the p4est framework and the presented AMR algorithms. In the tests we also use a second order space discretization for comparison reasons.

4.4.1 Advection on the Diagonal

As a first test for the transport equation we will propose the advection of a star-shaped indicator function on the diagonal of our domain $\Omega = [0,1]^2$.

The initial condition for the star is:

$$c_0(x,y) = \begin{cases} 1, & (x,y) \in \mathcal{A}, \\ 0, & \text{otherwise}, \end{cases}$$

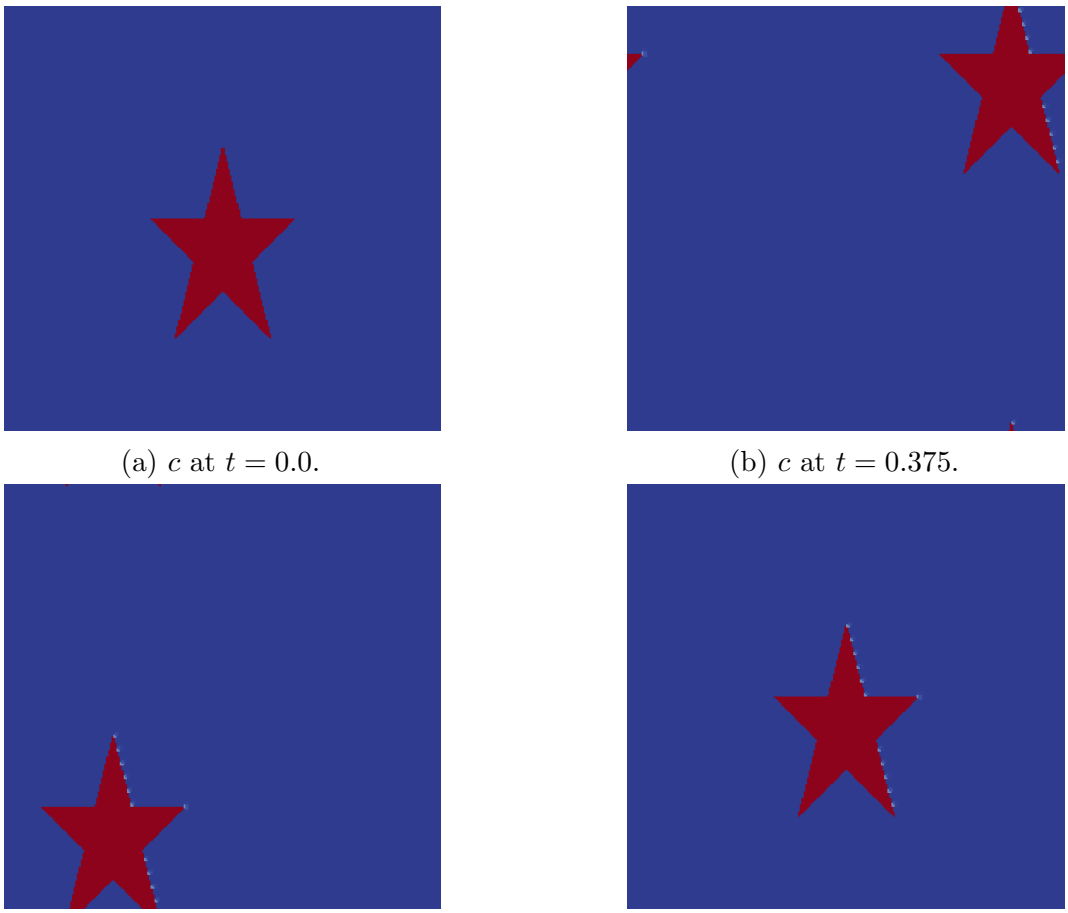
where $\mathcal{A}$ will define our star:

$$\mathcal{A} = \left\{ (x,y) \mid \frac{1}{3} - \left| x - \frac{1}{2} \right| < y < \frac{2}{3} - 4 \left| x - \frac{1}{2} \right| \right\} \cup \left\{ (x,y) \mid \frac{1}{3} + \left| x - \frac{1}{2} \right| < y < \frac{1}{2} \right\}.$$

We also define the velocity vector $\mathbf{u} = (1,1,0)$, the final time $T = 1.0$ and the Courant number $\theta = 1.0$ for the first order scheme and $\theta = 0.5$ for the second order scheme. In the case of the second order scheme we will use the minmod limiter.

Using these parameters, we know that the discrete *first order* approximation of the transport equation is exact and we can indeed see this in Figure 4.5. Note that the solution is exact when transporting on the diagonal because of the dimensional splitting technique we have used. Without directional splitting, the result would be exact only if the velocity was one of the standard basis vectors $\mathbf{e}_i$.

Unfortunately, given the refining criterion we have used, there are several cells that have been diffused in the first step of the simulation (on the right side of the star). A cure to this



(a) c at $t = 0.0$.

(b) c at $t = 0.375$.

(c) c at $t = 0.75$.

(d) c at $t = 1.0$.

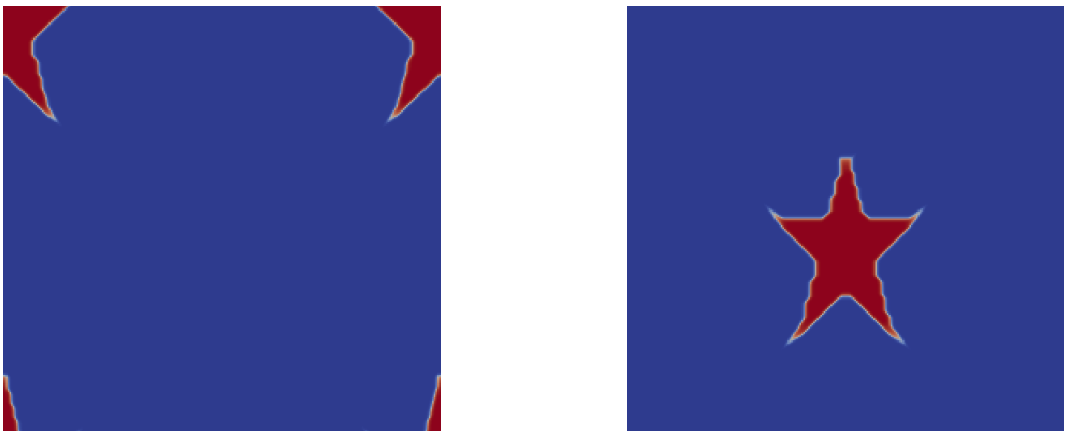
Figure 4.5: First order approximation of the advection of a star shape at different times.

issue could consists in adopting a different refinement criterion that refines more cells around the star.

We can see that by using a second order approximation we loose the match with the exact solution. Nonetheless, the second order scheme is more accurate than a the first order scheme (easily tested with $\theta \neq 1$ for the first order scheme). We can see the results for the second order approximation in Figure 4.6.

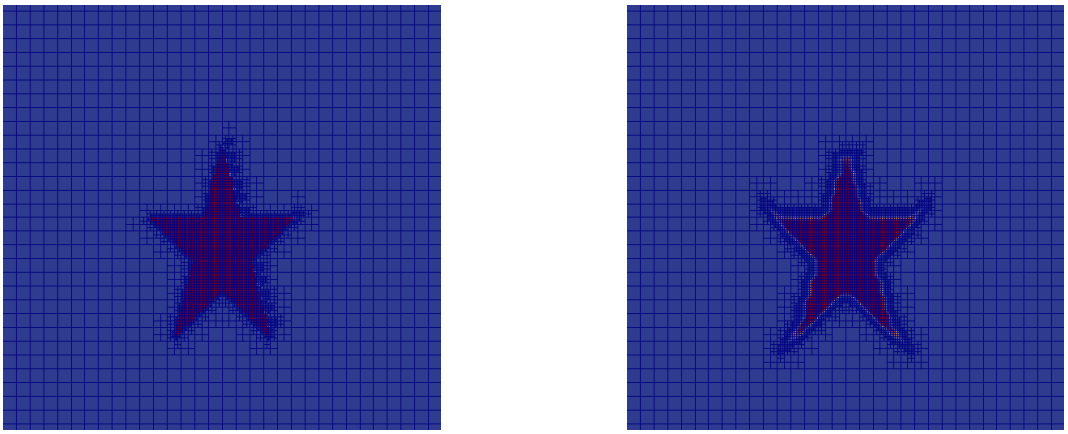
In Figure 4.7 we can also see the way in which the refinement was performed. Since both the first order and the second order approximations have barely any diffusion, the number of quadrants necessary to represent the two solutions are comparable.

Remark. The refinement criterion for the star was changed to refine any cell where the value is greater than 0.001. This is done to limit the dissipation problem that has appeared in the second order approximation due to passing into coarser levels while doing dimensional splitting.



(a) c at $t = 0.5$. (b) c at $t = 1.0$.

Figure 4.6: Second order approximation of the advection of a star shape at different times.



(a) First order. (b) Second order.

Figure 4.7: Refinement at the end of the simulation for the first order and second order approximations.

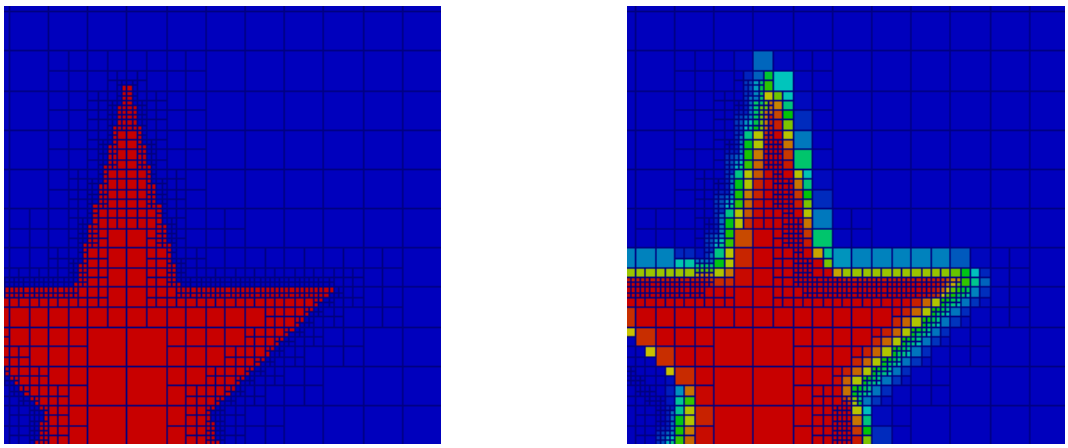
Advection with subcycling

We have seen that the solution is correctly transported when using a normal scheme with a single time step. This is no longer the case when doing time subcycling.

In the case of subcycling, the time step at which we advanced the whole mesh is given by the coarsest level. Considering the fact that the velocity is $\mathbf{u} = (1, 1, 0)$, we know the time step given by the CFL conditions is

$$\Delta t = \theta \Delta x,$$

where Δx is the size of a cell at a certain level. Given the minimum level l_{min} and maximum level of refinement l_{max} at a certain time step, we know that the Δt when doing subcycling is $2^{l_{max} - l_{min}}$ times larger than the time step we would obtain with the normal time stepping scheme.



(a) c at $t = 0$.

(b) c at $t = 0.015625$.

Figure 4.8: Solution after a single time step using subcycling.

However, the fact that we are advancing several steps at finer levels has its downsides. As we can see in Figure 4.8, the solution advances quicker than the refinement leading to a lot of diffusion. This is obviously incorrect because the upwind scheme for the transport equation is exact for the chosen velocity and Courant number.

A few ways to circumvent the problem come to mind, such as:

- refining the mesh at each *sub-time step* during subcycling. While this would solve our problem, it would greatly slow down the computation since adapting the mesh is the most costly operation.
- adding extra layers or refined cells to the existing mesh to be sure that the fine levels don't overflow into the coarser levels. This has the downside of potentially adding many more cells to the mesh.

Even though advancing in time with larger steps potentially speeds up the computations a great deal, the downsides of subcycling and the proposed solutions even out the gains. Further study is needed into the method of subcycling.

4.4.2 Rotation of Zalesak's Disk

The second example we will look at is a disk in solid body rotation. In this example we can expect a lot of numerical diffusion from the upwind scheme. We will try to compare the solutions using a first order approximation and a second order approximation with the `minmod` limiter.

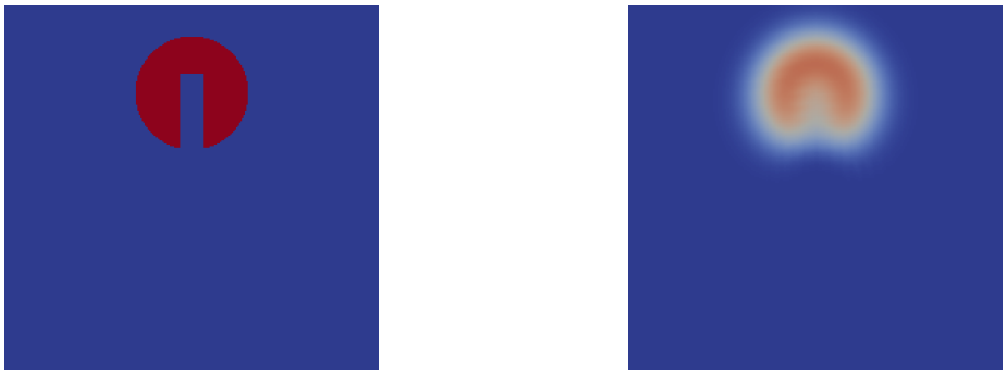
The initial solution we have chosen is a famous test called the *Zalesak Disk* that was first presented by S. T. Zalesak in [19]. It is given by:

$$c_0(\mathbf{x}) = \begin{cases} 1, & \sqrt{(x-0.5)^2 + (y-0.75)^2 + (z-0.5)^2} \leq 0.1, \\ 0, & \text{otherwise} \end{cases}$$

and the velocity vector is:

$$\mathbf{u} = (-(y - 0.5), x - 0.5, 0).$$

We will use a final time $T = 6.283$ (to get a complete rotation) and a Courant number $\theta = 1.0$ for the first order approximation and $\theta = 0.5$ for the second order approximation. Note that for the second order approximation, $\theta = 0.5$ is the maximum allowed value.



(a) c at $t = 0$.

(b) c at $t = 6.283$.

Figure 4.9: First order approximation of the advection of the Zalesak Disk.

We can see in Figure 4.9 that the disk is correctly transported and the refinement correctly follows and grows with the dissipation. The dissipation is, however, very pronounced and we can hardly distinguish the solution at the final time (Figure 4.9b). The second order approximation (see Figure 4.10) gives a significantly improved result with hardly any dissipation. We can see however that the solution gets deformed in other ways.

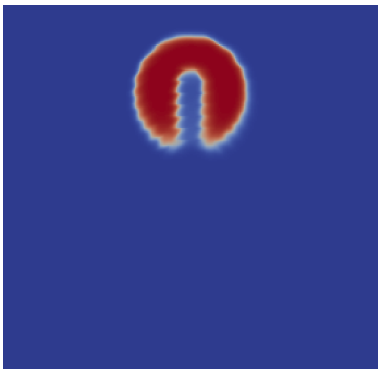
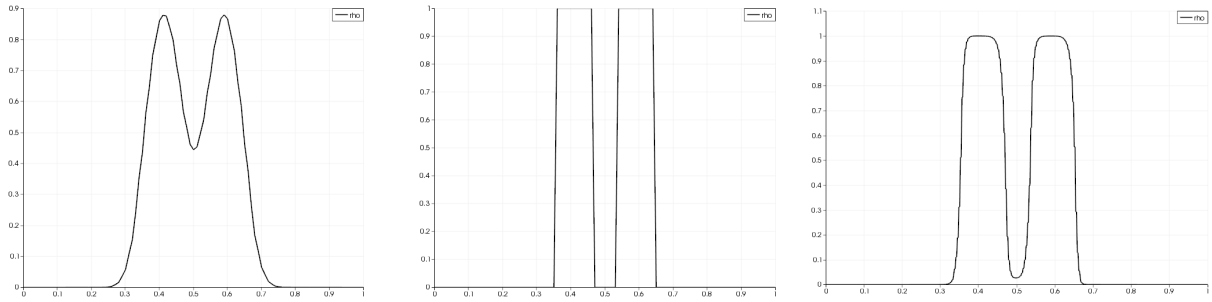


Figure 4.10: Second order approximation of the advection of the Zalesak Disk at time $t = 6.283$.

In Figure 4.11, we have taken a slice at $y = 0.25$ of the first and second order approximations at time $t = 0.5T$. We can indeed see the difference between the exact solution and the two approximations. The first order approximation is the worse of the two, as expected and seen in Figure 4.9.

As for the mesh refinement, we can see in Figure 4.12 that the area containing most of the dissipation from the first order scheme has been correctly captured by our refining



(a) First order. (b) Exact Solution. (c) Second order.

Figure 4.11: Comparison of the first and second order approximation at time $t = 3.1415$.

criteria.

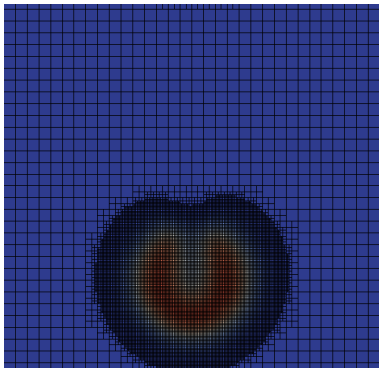


Figure 4.12: Refinement of the first order approximation at time $t = 3.1415$.

4.4.3 Deformation in 2D

The last test case that we will study for the transport equation is a test proposed by R. J. LeVeque in [24]. The test consists of stretching and un-stretching a disk in a vortex from which we will look only at the stretching of the disk and the resulting refinement.

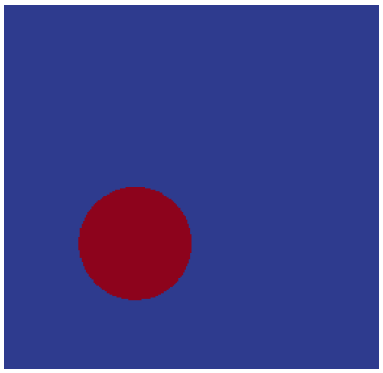


Figure 4.13: Initial condition for the vortex, as given by $c_0(\mathbf{x})$.

The initial condition is a disk centered in $(0, 0.75)$ with a radius of 0.15:

$$c_0(\mathbf{x}) = \begin{cases} 1, & \sqrt{x^2 + (y - 0.75)^2} \leq 0.15 \\ 0, & \text{otherwise} \end{cases}$$

with a rotating velocity such as:

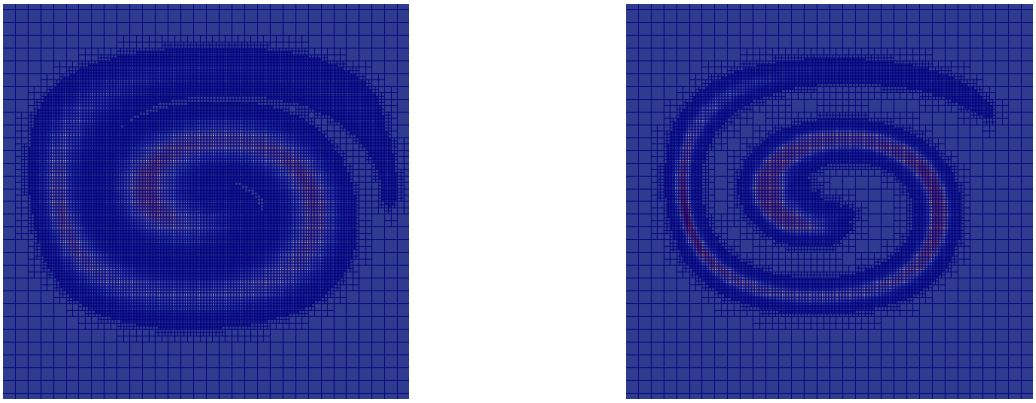
$$\mathbf{u} = (2\sin^2(\pi x)\sin(2\pi y), -\sin(2\pi x)\sin^2(\pi y)).$$

The final time for the simulation is $T = 3.0$ and the Courant number is $\theta = 0.5$. The test has been carried out with a second order scheme and the `minmod` limiter.

The results of a second order approximation can be seen in Figure 4.14. The results are not very diffusive and the vortex forms as one would expect (given previous results from [24]).

(a) c at $t = 1.5$. (b) c at $t = 3$.

Figure 4.14: Second order approximation of the vortex at intermediate time steps.



(a) First order. (b) Second order.

Figure 4.15: Refinement of the solution at time $t = 1.5$ for the first and second order approximations.

We can see in Figure 4.15 that the refinement correctly follows the disk as it deforms. In this case, the diffusion of the first order approximation is so great that the fine quadrants eventually cover most of the domain, making AMR a lot less useful.

5 Simulation of a Two-Phase Model using `p4est`

In this chapter we will consider a hyperbolic system of equations that describes a simple two-phase model. The model has been studied in [20].

5.1 Model Description

We suppose given two compressible materials $k = \{0, 1\}$ that are equipped with an equation of state (EOS) in the form of a **barotropic pressure law**:

$$\rho_k \to p_k(\rho_k),$$

where ρ_k is the and p_k are respectively the density and the pressure of component k. We make the assumption that:

$$\frac{\mathrm{d}p_k}{\mathrm{d}\rho_k} > 0$$

and denote by c_k the sound velocity of the fluid k that verifies:

$$c_k^2 = \frac{\mathrm{d}p_k}{\mathrm{d}\rho_k}.$$

The *global density* of the medium is defined by:

$$\rho = \alpha_1\rho_1 + \alpha_2\rho_2,$$

where α_k is the *volume fraction* of fluid k. We supposed both fluids to be **immiscible**, which implies that:

$$\alpha_1 + \alpha_2 = 1.$$

Notation. We will denote $\alpha = \alpha_1$ and, by definition, $1 - \alpha = \alpha_2$.

The pressure p of the medium as well as the volume fraction α are defined thanks to the hypothesis that for a given value of $\alpha\rho_1$ and $(1 - \alpha)\rho_2$ the following closure relation stands:

$$p_1(\rho_1) = p_2(\rho_2).$$

Consequently, p can be expressed as a function of ρ_1, ρ_2 and α. The sound velocity c of the two-phase medium is defined by:

$$c^2 = \frac{\alpha\rho_1}{\rho}\left(\frac{\partial p}{\partial(\alpha\rho_1)}\right)_{(1-\alpha)\rho_2} + \frac{(1-\alpha)\rho_2}{\rho}\left(\frac{\partial p}{\partial((1-\alpha)\rho_2)}\right)_{\alpha\rho_1}.$$

The pressure equilibrium hypothesis yields then that:

$$c^2 = \frac{1}{\rho}\left(\frac{\alpha}{\rho_1 c_1^2} + \frac{1-\alpha}{\rho_2 c_2^2}\right),$$

which is usually referred to as the *Wallis formula*.

From here on we will suppose that each component of our system is a *stiffened gas* material. This implies that the pressure laws are:

$$p_k(\rho_k) = p_{k,0} + c_k^2(\rho_k - \rho_{k,0}), \tag{5.1}$$

where $p_{k,0}, \rho_{k,0}$ and c_k are positive constants that are characteristic of the fluid k. For such materials, we can see that α is given as the root of a second order polynomial. Indeed, if we denote $m_1 = \alpha\rho_1$ and $m_2 = (1-\alpha)\rho_2$, we have:

$$p_1(\rho_1) = p_2(\rho_2) \iff p_1\left(\frac{m_1}{\alpha}\right) = p_2\left(\frac{m_2}{1-\alpha}\right)$$

which can be expanded using (5.1), to:

$$p_{1,0} + c_1^2\left(\frac{m_1}{\alpha} - \rho_{1,0}\right) = p_{2,0} + c_2^2\left(\frac{m_2}{1-\alpha} - \rho_{2,0}\right),$$

giving the following order 2 polynomial in α:

$$q_1\alpha^2 - (q_1 + q_2)\alpha + c_1^2 m_1 = 0.$$

Out of the two roots of the polynomial, the one that respects the constraint $\alpha \in [0,1]$ is:

$$\alpha = \frac{(q_1+q_2) - \sqrt{(q_1+q_2)^2 - 4q_1c_1^2m_1}}{2q_1}, \tag{5.2}$$

where:

$$\begin{cases} q_1 = p_{20} - p_{10} - (c_2^2\rho_{20} - c_1^2\rho_{10}), \\ q_2 = c_1^2 m_1 + c_2^2 m_2. \end{cases}$$

The kinematics of the medium is determined by a velocity field $\mathbf{u}$ that is shared by both materials.

The evolution of the two-phase system is governed by the following equations:

$$\begin{cases} \partial_t(\alpha\rho_1) + \nabla\cdot(\alpha\rho_1\mathbf{u}) = 0, \\ \partial_t((1-\alpha)\rho_2) + \nabla\cdot((1-\alpha)\rho_2\mathbf{u}) = 0, \\ \partial_t(\rho\mathbf{u}) + \nabla\cdot(\mathbf{u}\otimes(\rho\mathbf{u})) + \nabla p = 0. \end{cases} \tag{5.3}$$

The system (5.3) can also be written in vector form:

$$\partial_t\mathbf{W} + \partial_x\mathcal{F}(\mathbf{W}) + \partial_y\mathcal{G}(\mathbf{W}) + \partial_z\mathcal{H}(\mathbf{W}) = 0, \tag{5.4}$$

where $\mathbf{W}$ is the state vector containing the conservative variables $(\alpha\rho_1, (1-\alpha)\rho_2, \rho u, \rho v, \rho w)$ and the fluxes in each direction are defined by:

$$\mathcal{F}(\mathbf{W}) = \begin{pmatrix} \alpha\rho_1 u \\ (1-\alpha)\rho_2 u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \end{pmatrix}, \mathcal{G}(\mathbf{W}) = \begin{pmatrix} \alpha\rho_1 v \\ (1-\alpha)\rho_2 v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \end{pmatrix} \text{ and } \mathcal{H}(\mathbf{W}) = \begin{pmatrix} \alpha\rho_1 w \\ (1-\alpha)\rho_2 w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \end{pmatrix}.$$

5.2 Numerical Scheme

In order to approximate the solutions of (5.4), we present a Finite Volume scheme based on a dimensional splitting strategy.

Following the outline presented in Chapter 4, the splitting strategy consists in advancing (5.4) in time, with a time step of Δt, by successively approximating the solutions of (5.5), (5.6) and (5.7).

$$\begin{cases} \partial_t \mathbf{W} + \partial_x \mathcal{F}(\mathbf{W}) = 0, & (5.5) \\ \partial_t \mathbf{W} + \partial_y \mathcal{G}(\mathbf{W}) = \mathcal{S}(\mathbf{W}), & (5.6) \\ \partial_t \mathbf{W} + \partial_z \mathcal{H}(\mathbf{W}) = 0. & (5.7) \end{cases}$$

In (5.6), we have added an additional source term that, in our case will correspond to the **gravity** in the y direction:

$$\mathcal{S}(\mathbf{W}) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -\rho\, g \\ 0 \end{pmatrix}.$$

Let us consider a one-dimensional problem set along the x axis and we integrate (5.5) over $[t_n, t_{n+1}] \times K_i$. This yields the integral balance relation:

$$\int_{t_n}^{t_{n+1}} \int_{K_i} \partial_t \mathbf{W} \, \mathrm{d}\mathbf{x} \, \mathrm{d}t + \int_{t_n}^{t_{n+1}} \int_{K_i} \partial_x \mathcal{F}(\mathbf{W}) \, \mathrm{d}\mathbf{x} \, \mathrm{d}t = 0.$$

Once again, the Finite Volume approximation is obtained by proposing a discrete approximate formula of this integral balance law. We set:

$$\mathbf{W}_i^{n+1} = \mathbf{W}_i^n - \frac{\Delta t}{|K_i|} \sum_{j \in \mathcal{N}_x(i)} |\Gamma_{ij}| n_{ij} \mathcal{F}_{ij}^n, \tag{5.8}$$

where $\mathbf{W}_i^n$ is a approximation of $\mathbf{W}$ within the cell K_i at instant t_n and $\mathcal{F}_{ij}^n$ is an approximation of the flux $\mathcal{F}$ along the face Γ_{ij} at instant t_n.

The full scheme for advancing (5.4) with a source term is:

$$\begin{cases} \mathbf{W}_i^* = \mathbf{W}_i^n - \dfrac{\Delta t}{|K_i|} \displaystyle\sum_{j \in \mathcal{N}_x(i)} |\Gamma_{ij}| n_{ij} \mathcal{F}_{ij}^n, & (5.9) \\ \mathbf{W}_i^{**} = \mathbf{W}_i^* - \dfrac{\Delta t}{|K_i|} \displaystyle\sum_{j \in \mathcal{N}_y(i)} |\Gamma_{ij}| n_{ij} \mathcal{G}_{ij}^n, & (5.10) \\ \mathbf{W}_i^{***} = \mathbf{W}_i^{**} + \Delta t \mathcal{S}_i^n, & (5.11) \\ \mathbf{W}_i^{n+1} = \mathbf{W}_i^{***} - \dfrac{\Delta t}{|K_i|} \displaystyle\sum_{j \in \mathcal{N}_z(i)} |\Gamma_{ij}| n_{ij} \mathcal{H}_{ij}^n. & (5.12) \end{cases}$$

We note that each equation uses the solution from the previous one as an initial condition. As in the case of the transport equation, this leads to a first order scheme since both the directional splitting method and the operator splitting methods we have used are of first order.

The flux $\mathcal{F}_{ij}^n$ between two cells K_i and K_j is given a function $\Phi(\mathbf{W}_L, \mathbf{W}_R)$, where $\mathbf{W}_L$ and $\mathbf{W}_R$ are the values on the left and right, respectively, of the interface between K_i and K_j. In our case, the definition of Φ is given by an approximate Riemann solver obtained by a Suliciu-type relaxation scheme (see [20] and [21]).

Let a be a positive constant whose choice will be specified later, the Suliciu relaxation approximate Riemann solver we have used gives 3 waves, as seen in Figure 5.1.

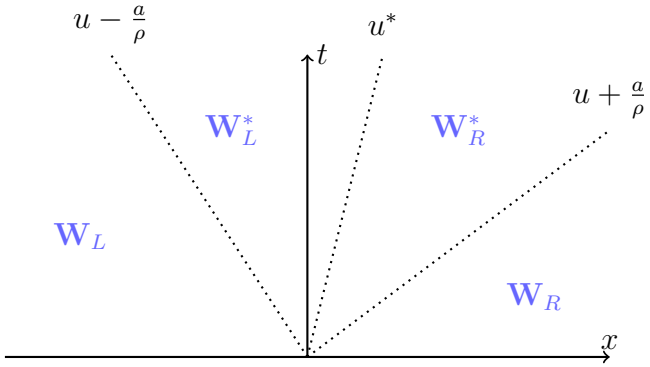


Figure 5.1: The three waves of the approximate Riemann solver.

The states delimited by the contact discontinuities in Figure 5.1 are given by:

$$\mathbf{W}^{RP}(\frac{x}{t}; \mathbf{W_L}, \mathbf{W_R}) = \begin{cases} \mathbf{W}_L = ((\alpha_1\rho_1), (\alpha_2\rho_2), (\rho u), (\rho v), (\rho w))_L, & \text{if } \dfrac{x}{t} \leq u - \dfrac{a}{\rho_L}, \\ \mathbf{W}_L^* = ((\alpha_1\rho_1), (\alpha_2\rho_2), (\rho u), (\rho v), (\rho w))_L^*, & \text{if } u - \dfrac{a}{\rho_L} < \dfrac{x}{t} \leq u^*, \\ \mathbf{W}_R^* = ((\alpha_1\rho_1), (\alpha_2\rho_2), (\rho u), (\rho v), (\rho w))_R^*, & \text{if } u^* < \dfrac{x}{t} \leq u + \dfrac{a}{\rho_R}, \\ \mathbf{W}_R = ((\alpha_1\rho_1), (\alpha_2\rho_2), (\rho u), (\rho v), (\rho w))_R, & \text{if } \dfrac{x}{t} \geq u + \dfrac{a}{\rho_R}, \end{cases} \quad (5.13)$$

where the densities of each of the two materials in the intermediate states $\mathbf{W}_L^*$ and $\mathbf{W}_R^*$ are defined as:

$$\begin{aligned}
(\alpha\rho_1)_L^* &= \left(\frac{1}{(\alpha\rho_1)_L} + \frac{u^* - u_L}{a}\right)^{-1}, \\
((1-\alpha)\rho_2)_L^* &= \left(\frac{1}{((1-\alpha)\rho_2)_L} + \frac{u^* - u_L}{a}\right)^{-1}, \\
(\alpha\rho_1)_R^* &= \left(\frac{1}{(\alpha\rho_1)_R} + \frac{u_R - u^*}{a}\right)^{-1}, \\
((1-\alpha)\rho_2)_R^* &= \left(\frac{1}{((1-\alpha)\rho_2)_R} + \frac{u_R - u^*}{a}\right)^{-1}
\end{aligned}$$

with the velocity in the two intermediate states:

$$u_L^* = u_R^* = u^* = \frac{1}{2}\left(u_R + u_L - \frac{p_R - p_L}{a}\right).$$

The total mass density in the medium in the intermediate states is:

$$\begin{cases} \rho_L^* = (\alpha\rho_1)_L^* + ((1-\alpha)\rho_2)_L^*, \\ \rho_R^* = (\alpha\rho_1)_R^* + ((1-\alpha)\rho_2)_R^*, \end{cases}$$

which permits to compute the momentum in the normal direction:

$$\begin{cases} (\rho u)_L^* = \rho_L^* u^*, \\ (\rho u)_R^* = \rho_R^* u^*, \end{cases}$$

The momentum in the two tangential directions is simply:

$$\begin{cases} (\rho v)_L^* = \rho_L^* v_L, \\ (\rho v)_R^* = \rho_R^* v_R, \end{cases} \quad \text{and} \quad \begin{cases} (\rho w)_L^* = \rho_L^* w_L, \\ (\rho w)_R^* = \rho_R^* w_R, \end{cases}$$

because we are considering only the sweep in the x direction which simply transports the velocities in the tangential y and z directions.

Using the approximate solution to the Riemann problem given by (5.13), we can define our flux as:

$$\mathcal{F}_{ij}^n = \Phi(\mathbf{W}_L, \mathbf{W}_R) = \mathcal{F}(\mathbf{W}^{RP}(x/t = 0; \mathbf{W_L}, \mathbf{W_R})). \tag{5.14}$$

Finally, we recall that in order to avoid numerical instabilities, the parameter a should be chosen in agreement with the *Whithman's subcharacteristic condition*:

$$\rho\, c(\alpha\rho_1, (1-\alpha)\rho_2) < a.$$

In practice we define a by setting:

$$a = K \max_i(\rho_i c_i),$$

where $K > 1$ is a constant.

Stability and Time Step Choice

The stability of the scheme is assured by the following CFL condition:

$$\Delta t \leq \frac{1}{2} \min_i \left[\Delta x_i \left(\|\mathbf{u}_i^n\| + \frac{a}{\rho_i^n} \right)^{-1} \right]. \tag{5.15}$$

A simple way to understand the CFL condition is by looking at Figure 5.2 and Figure 5.1. The main argument for stability is that the waves at the left and right of the cell K_i (considering a one dimensional problem) do not intersect during the time step from t_n to t_{n+1}.

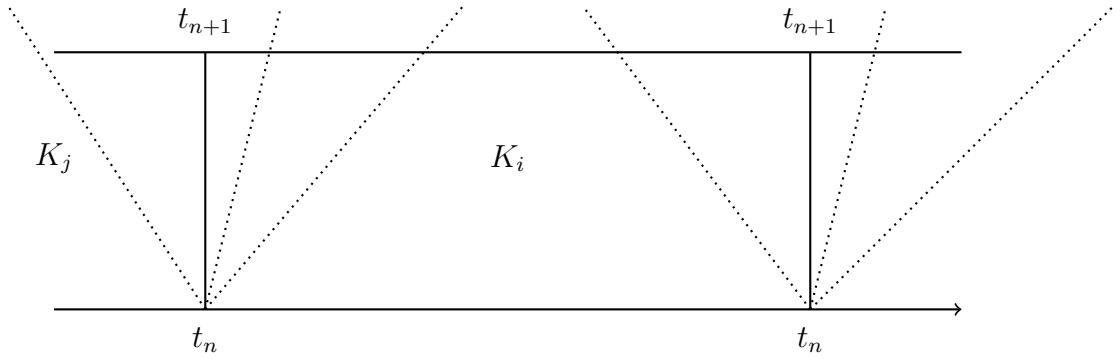


Figure 5.2: Waves at the left and right interface of a cell K_i.

This sufficient stability condition provides a strategy for choosing the time step:

$$\Delta t = \theta \frac{\Delta x}{\max_i (\|\mathbf{u}_i^n\| + \frac{a}{\rho_i^n})}, \quad \theta \in [0, 0.5]$$

where Δx is the size of the octant at the finest level l_{max}, given by:

$$\Delta x = 2^{-l_{max}}.$$

Body Forces Term

In the numerical tests we shall consider we will need to account for gravity terms $\mathcal{S}$. We shall achieve thanks to a simple two-step operator splitting as follows:

$$\begin{cases} \partial_t \mathbf{W} + \partial_x \mathcal{G}(\mathbf{W}) = 0, \\ \partial_t \mathbf{W} = \mathcal{S}(\mathbf{W}). \end{cases}$$

The second step does not involve any partial differential and thus is a simple ODE that will be solved by a cell-centered explicit approximation.

Algorithm

We have seen in the previous paragraphs how to advance (5.4) in time and we will now present a method for advancing the whole system, which includes computing the volume fraction α (using (5.2)) and the pressure p (using (5.1)).

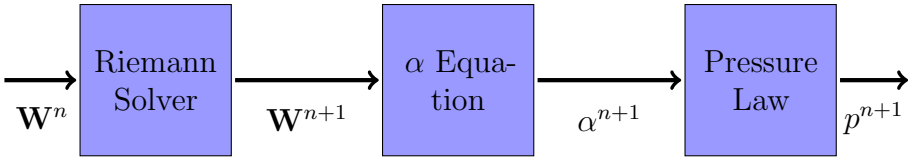


Figure 5.3: Steps for advancing the solution in time.

The algorithm advances as follows (also see Figure 5.3):

1.1. Advance $\mathbf{W}_i^n$ in time by successively applying (5.10), (5.11) and (5.12) under the stability condition (5.15) to solve:

$$\begin{cases} \partial_t \mathbf{W} + \partial_x \mathcal{F}(\mathbf{W}) = 0, \\ \partial_t \mathbf{W} + \partial_y \mathcal{G}(\mathbf{W}) = 0, \\ \partial_t \mathbf{W} + \partial_z \mathcal{H}(\mathbf{W}) = 0. \end{cases}$$

1.2. Use (5.12) if necessary to introduce the source term.

2.1. Compute the volume fraction α_i^{n+1} according to (5.2) using the updated values $(\alpha\rho_1)_i^{n+1}$ and $((1-\alpha)\rho_2)_i^{n+1}$.

2.2. Compute the new pressure p_i^{n+1} according to (5.1) and the updated values α_i^{n+1}, $(\alpha\rho_1)_i^{n+1}$ and $((1-\alpha)\rho_2)_i^{n+1}$.

5.3 Numerical Results

In this section we will try to present several results for the two-phase model using the approximate Riemann solver resulted from a Suliciu-type relaxation scheme. As in the previous chapter, all the tests have been done in parallel using the `p4est` library and our own code that is available at [31].

For the two-phase model we require several parameters to be set. These are available in Table 5.1. We will note in each particular example when the source term using gravity is used and when it is not.

Parameter	Value
$\rho_{1,0}$	$1\ \mathrm{kg\,m^{-3}}$
$\rho_{2,0}$	$1000\ \mathrm{kg\,m^{-3}}$
$p_{1,0}$	10^5 Pa
$p_{2,0}$	10^5 Pa
$c_{1,0}$	$340\ \mathrm{m\,s^{-1}}$
$c_{2,0}$	$1500\ \mathrm{m\,s^{-1}}$
g	$9.81\ \mathrm{m\,s^{-2}}$

Table 5.1: Table of parameters for the two-phase model.

Refinement Criteria

For the two-phase model we have chosen a more advanced refining criterion to take into account more of the variables in the system. In particular, we looked at two different choices:

- Computing the gradient of the pressure and the total mass density of the medium.
- Or computing the gradient of the velocity in each direction and the total mass density of the medium.

The gradient estimator for the pressure is:

$$\xi_i^p = \max_{j \in \mathcal{N}_x(i)} \frac{|p_i^n - p_j^n|}{\max(|\,p_i^n\,|, |\,p_j^n\,|)}$$

and the gradient estimator for the total mass density is:

$$\xi_i^r = \max_{j \in \mathcal{N}_x(i)} \frac{|\rho_i^n - \rho_j^n|}{\max(|\,\rho_i^n\,|, |\,\rho_j^n\,|)}$$

where:

$$\rho_i^n = (\alpha\rho_1)_i^n + ((1-\alpha)\rho_2)_i^n.$$

For the velocity vector, we have used the following indicator for a single component:

$$\xi_i^u = \max_{j \in \mathcal{N}_x(i)} \frac{|u_i^n - u_j^n|}{\max(|\,u_i^n\,|, |\,u_j^n\,|)}$$

and then approximated an indicator by taking the maximum of all the components of the velocity vector $\xi_i^{\mathbf{u}} = \max(\xi_i^u, \xi_i^v, \xi_i^w)$.

An octant is then refined if $\max(\xi_i^p, \xi_i^r) > \epsilon$, where ϵ is a user defined threshold. In our case, we have noticed that the best results were obtained using $\epsilon = 0.005$. Similarly, an octant gets coarsened if $\max(\xi_i^p, \xi_i^r) < \epsilon$.

5.3.1 SOD: Shock and Rarefaction Wave

The first test we will be looking at is the classical *SOD shock tube*. The test was first introduced by G. A. Sod in [25] as a test for the accuracy of several finite difference methods and has since been used for testing many fluid dynamics schemes.

The modified test that we have used is defined on a domain $\Omega = [0, 2] \times [0, 1]$. We have then defined two states, on the left and right of $x = 1$, as follows:

$$\rho_1(0, \mathbf{x}) = \begin{cases} 100, & x < 1, \\ 1, & x > 1, \end{cases} \quad \text{and} \quad \alpha(0, \mathbf{x}) = \begin{cases} 1 - \lambda, & x < 1, \\ \lambda, & x < 1, \end{cases}$$

where $\lambda = 10^{-7}$. The velocity on the domain is uniformly null $\mathbf{u}(0, \mathbf{x}) = (0, 0, 0)$. The other variables are computed starting from ρ_1 and α.

- p is computed from the barotropic pressure law (5.1).
- ρ_2 is then computed from the constraint of pressure equilibrium and the barotropic pressure law (5.1).
- The conservative variables that form the vector $\mathbf{W}$ can now be computed using these values.

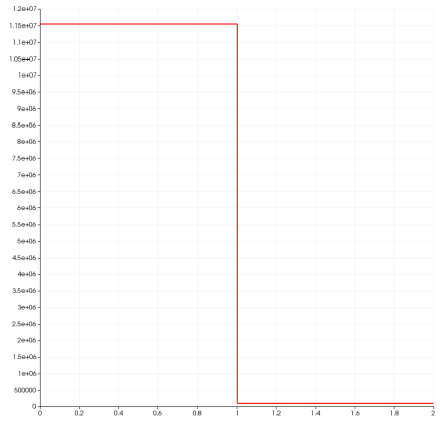
We do not explicitly set the mass densities in both materials, because we cannot be sure that the given values will automatically satisfy the pressure equilibrium.

A first test has been done with the first order scheme with a final time $T = 0.001$ and a Courant number $\theta = 0.5$. We can see in Figure 5.4 the initial state and an intermediary state for the pressure and the velocity in the x direction. Since this is a 1D problem, we only need to look in one direction, namely x, because the solution is constant in all the others.

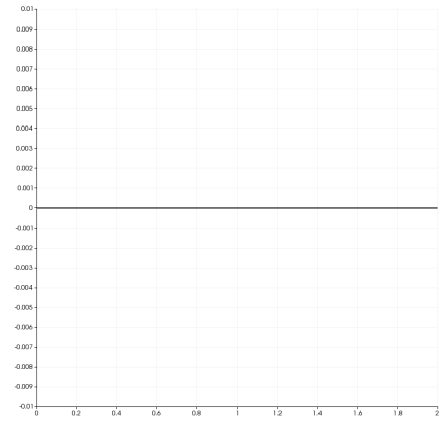
We can see in Figure 5.4d that the profile of the velocity matches that in [20] and, furthermore, the shock on top and the rarefaction wave at the bottom have developed as one would expect.

The second test we have performed is using a second order space discretization and the `minmod` limiter to limit oscillations. The simulation has a final time $T = 0.001$ and a Courant number $\theta = 0.25$. Again, we will only look at the result in the x direction.

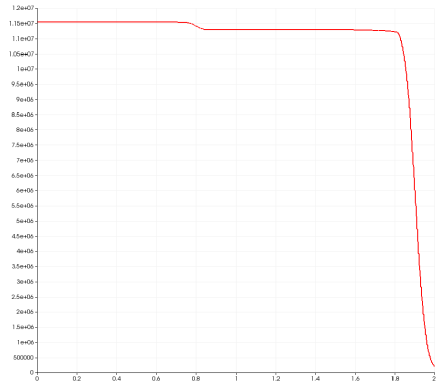
We can see in Figure 5.5 that the profile is the same, but the solution has improved. The shock on the right side is better represented with a steeper slope than in the more diffusive first order simulation. As for the refinement, we can see in Figure 5.6a that the mesh was only refined near $x = 0$ where there is a discontinuity in the mass density. Later, we can see that the refinement follows the shock and the rarefaction wave produced by the velocity in the x direction very closely (see Figure 5.6b).



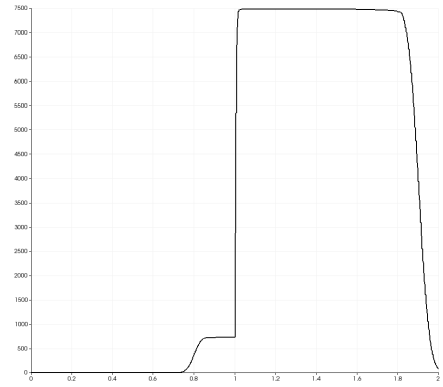
(a) p at $t = 0$.



(b) u at $t = 0$.

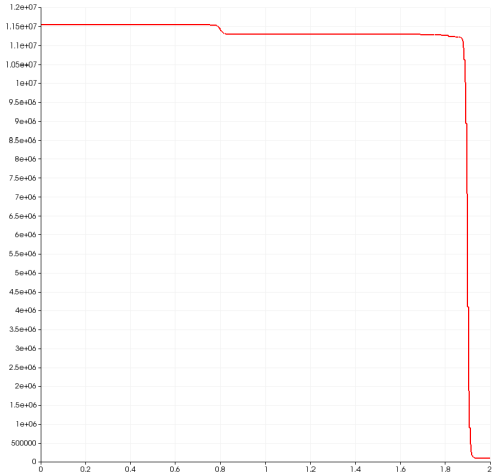


(c) p at $t = 0.0006$.

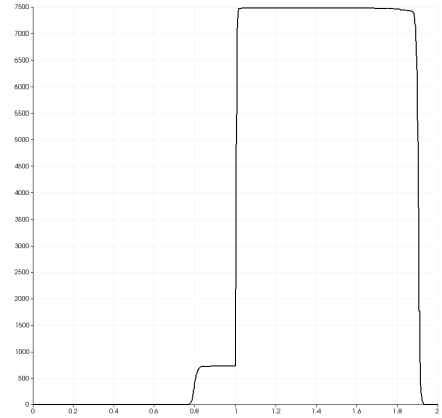


(d) u at $t = 0.0006$.

Figure 5.4: Simulation results from the first order scheme.

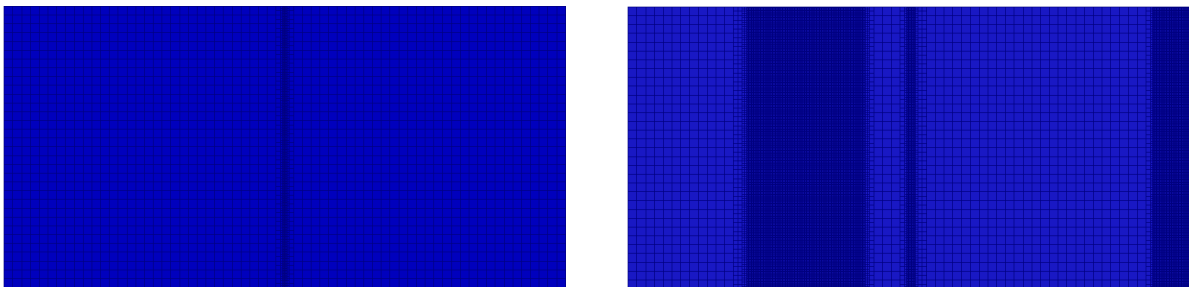


(a) p at $t = 0.0006$.



(b) u at $t = 0.0006$.

Figure 5.5: Simulation results from the second order scheme.



(a) Refinement at $t = 0$.

(b) Refinement at $t = 0.0006$.

Figure 5.6: Mesh refinement during the simulation of the second order scheme.

5.3.2 Two Rarefaction Waves

The second test we will be looking at is also a classical test in fluid dynamics. It involves creating two rarefaction waves in the middle of our domain that can be compared to exact solutions.

The initial condition for this test is described by $\rho_1(0, \mathbf{x}) = 1.0$ and $\alpha(0, \mathbf{x}) = 0.1$, for all $\mathbf{x} \in \Omega$. The velocity field has opposite sides on each half of the domain:

$$\mathbf{u}(0, \mathbf{x}) = \begin{cases} -2, & x < 1, \\ 2, & x > 1. \end{cases}$$

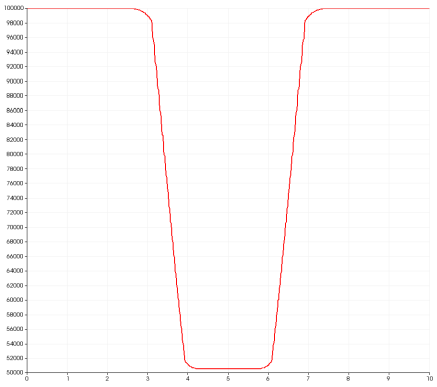
As in the previous case, these variables will allow us to compute the mass density in the second fluid, as well as the pressure of the medium and all the conservative variables at time $t = 0$.

The solution was simulated on the same domain $\Omega = [0, 2] \times [0, 1]$ with a final time $T = 0.05$ and a Courant number $\theta = 0.5$ for the first order simulation and $\theta = 0.25$ for the second order simulation. The second order scheme used the `minmod` limiter as before.

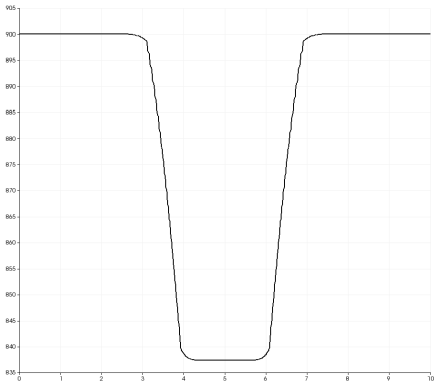
We can see in Figure 5.7a and Figure 5.7b the results for the simulation at final time $T = 0.05$ for the pressure and the total mass density. The two rarefaction waves are symmetric around $x = 1$ and they have evolved correctly for both the first order and the second order simulations.

In the case of the second order simulations, we can see at the top of each rarefaction wave (e.g. Figure 5.7d) there is a change in slope. We suspect that this is happening because the scheme is second order in space and not in time. The simple second order space discretization and the use of limiters seems to have a stiffening effect on the result.

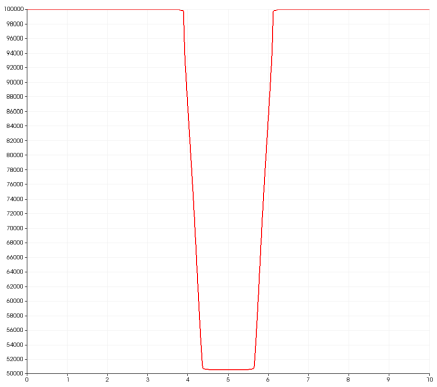
In Figure 5.8, we can also see that, in the case of the simulation done with the second order scheme, the mesh is correctly refined in the areas where the rarefaction waves are developing, thus correctly capturing the phenomenon.



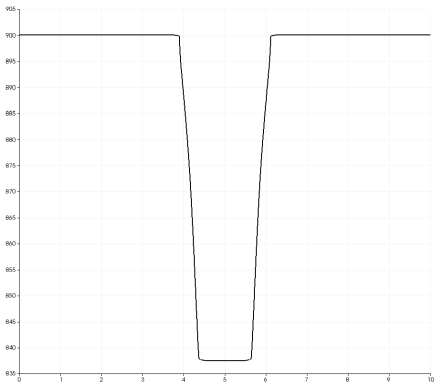
(a) p at $t = 0.05$.



(b) ρ at $t = 0.05$.



(c) p at $t = 0.05$.



(d) ρ at $t = 0.05$.

Figure 5.7: Simulations with a (a)(b) First order scheme (c)(d) Second order scheme.

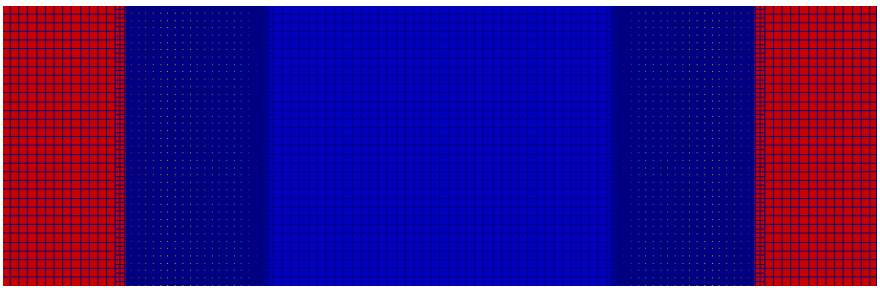


Figure 5.8: Refinement at the end of the simulation, at $t = 0.05$.

5.3.3 Dam Break

The third test we will be looking at is a sightly more complicated fluid dynamics example, namely a dam break. A dam break simulation has **gravity** as a driving force because the initial condition is at rest.

The dam break test case is defined by the following mass density for the gas:

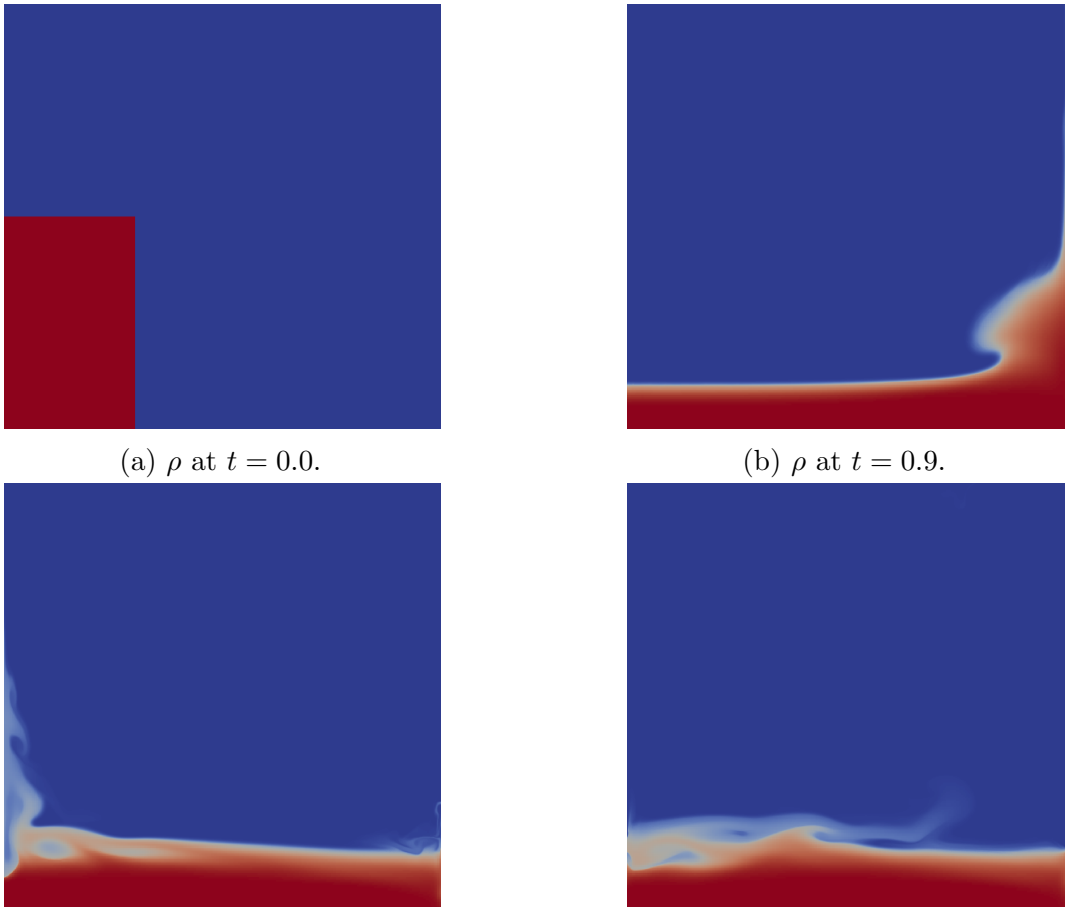
$$\rho_1(0, \mathbf{x}) = \rho_{1,0} \left(1 + g \frac{0.5 - y}{c_1^2}\right)$$

and the volume fraction given by:

$$\alpha(0, \mathbf{x}) = \begin{cases} 1, & (x, y, z) \in [0, 0.3] \times [0, 0.5] \times [0, 0.3], \\ 0, & \text{otherwise.} \end{cases}$$

The initial velocity is $\mathbf{u} = (0, 0, 0)$. This test has only been performed using the first order scheme with a final time $T = 2$ and a Courant number $\theta = 1.0$. The result of the simulation can be seen in Figure 5.9.

This simulation was performed in parallel on 256 CPUs with a minimum allowed refinement level of 7 and a maximum level of 11. It is the most massive of the tests we have performed in this paper, taking slightly over 3 hours.



(a) ρ at $t = 0.0$.

(b) ρ at $t = 0.9$.

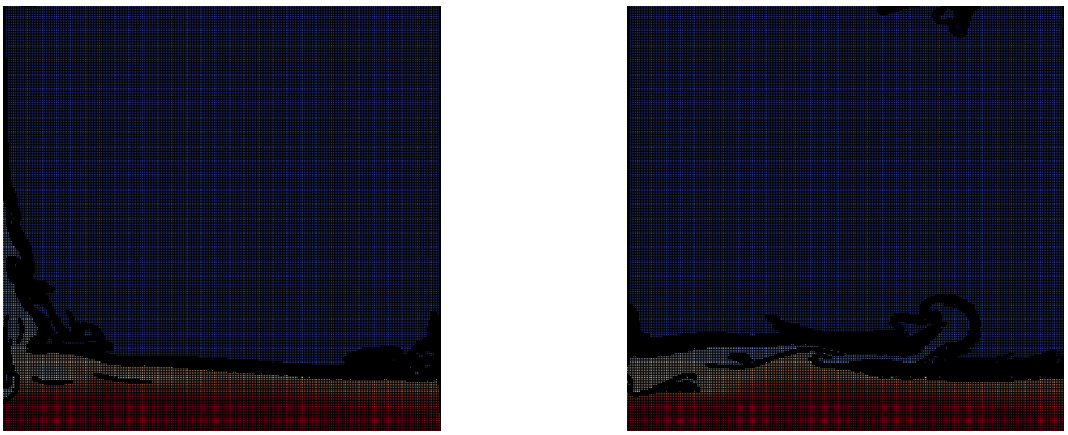
(c) ρ at $t = 1.5$.

(d) ρ at $t = 2.0$.

Figure 5.9: First order simulation of the dam break at different times. Plotting the total mass density of the medium.

We can see in the images from Figure 5.9 that the simulation is very diffusive due to the first order approximation. This dissipation also leads to a *numerical viscosity* that makes the fluids act in non-physical ways. Nonetheless, the simulation has evolved in predictable ways and can be easily improved by using the second order scheme.

Figure 5.10 contains examples of the mesh refinement during the simulation. We can see that it correctly follows the interfaces between the two fluids even as the liquid collapses on itself.



(a) Refinement at $t = 1.5$.

(b) Refinement at $t = 2.0$.

Figure 5.10: The refinement in two stages of the simulation.

6 Conclusions

In this document we have presented a short overview of the existing Adaptive Mesh Refinement technologies, with a specific focus on cell-based AMR. The present document was concerned with adding further documentation and information to the existing body of knowledge surrounding the `p4est` library.

The `p4est` library has proved to be a very promising new project in the complex world of cell-based AMR implementation. The qualities that we feel set it apart are the novel parallel algorithms and data structures that have been developed and detailed as part of the `p4est` implementation and the focus on code quality. The `p4est` code is open source and tries to allow easy and fast access to new contributors by having a documented code base, strict coding guidelines and version control.

During this internship, we have been in contact with the authors of `p4est` and have even contributed a small new feature to the library. This alone gave us a very positive view on the library and possible future collaborations or projects that could involve `p4est`.

As many open source products, the `p4est` library is still a work in progress. There are several cases where the implementation has not caught up to the advanced corner and edge-based cases presented in the algorithms from [14], the input / output routines are lacking, etc. However, this has not stopped other projects from successfully using `p4est`. Notable examples are the `deal.II` Finite Element library and the newly developed, and yet to be released, ForestClaw [26] Finite Volume code.

The main element that we have left out of our analysis of `p4est` has been performance and scalability studies. Even though the library has been proven to scale up to over 220,000 CPUs, replicating such results (even at a smaller scale) would be invaluable. To make this possible, a new code with a more strict view on performance and optimization would likely have to be developed, but we are optimistic that, after the results we have obtained in this work, it is a worthwhile endeavor.

List of Figures

2.1 Example of overlapped grids. 12
2.2 Example of a quadtree. 15
2.3 Example of a refined square domain that can be represented by a quadtree. . 15
2.4 Tree Links as described in [9]. 16
2.5 An octant as described in [10]. 17
2.6 Example of (a) a valid refined mesh and (b) a invalid refined meshes. 18
2.7 Grid indexes as described in [11]. 18

3.1 Numbering of the corners, edges and faces for an octant. 22
3.2 The corners and faces of a quadrant. 22
3.3 Two trees with different orientations. 23
3.4 z-order traversal of the quadrants in a tree. 24
3.5 Coordinate system, corner and face numbering of two adjacent trees. 26
3.6 Two trees connected solely through a corner. 28
3.7 Parent-child relationship in a mesh. 29
3.8 The entire forest is stored in a global linear array indexed by g_i, the global index of each octant. 30
3.9 Partition between processes of the global linear array. 31
3.10 Coarsening the octant family starting with o. 34
3.11 Refining the octant q and adding its 4 children. 36
3.12 Overview of the application workflow. 37
3.13 A face (marked in red) between quadrants of different levels. 43
3.14 (a) Initial solution on the uniform mesh of level 4. (b) Initial solution on a refined mesh. 47
3.15 Initial solution on a refined 3D mesh. 48
3.16 (a) Mesh and solution at $t = 0.5$. (b) Mesh and solution at $t = 1.0$. 48
3.17 Refined solution on a refined 3D mesh at an intermediary time step. 49
3.18 (a) Partition at $t = 0.0$ (b) Partition at $t = 0.3$. 49
3.19 Load-balanced partition of the quadrants on 8 processes using a refined 3D mesh. 49

4.1 A quadrant with its refined and coarsened face neighbors. 51
4.2 Left and right, potentially refined, neighbors of a quadrant. 52
4.3 Two octants in different processes. 59
4.4 Flux computations at level l (in red) and level $l - 1$ (in blue). 60
4.5 First order approximation of the advection of a star shape at different times. 61
4.6 Second order approximation of the advection of a star shape at different times. 62

4.7 Refinement at the end of the simulation for the first order and second order approximations. 62
4.8 Solution after a single time step using subcycling. 63
4.9 First order approximation of the advection of the Zalesak Disk. 64
4.10 Second order approximation of the advection of the Zalesak Disk at time $t = 6.283$. 64
4.11 Comparison of the first and second order approximation at time $t = 3.1415$. . 65
4.12 Refinement of the first order approximation at time $t = 3.1415$. 65
4.13 Initial condition for the vortex, as given by $c_0(\mathbf{x})$. 65
4.14 Second order approximation of the vortex at intermediate time steps. 66
4.15 Refinement of the solution at time $t = 1.5$ for the first and second order approximations. 66

5.1 The three waves of the approximate Riemann solver. 70
5.2 Waves at the left and right interface of a cell K_i. 72
5.3 Steps for advancing the solution in time. 73
5.4 Simulation results from the first order scheme. 76
5.5 Simulation results from the second order scheme. 76
5.6 Mesh refinement during the simulation of the second order scheme. 77
5.7 Simulations with a (a)(b) First order scheme (c)(d) Second order scheme. . . 78
5.8 Refinement at the end of the simulation, at $t = 0.05$. 78
5.9 First order simulation of the dam break at different times. Plotting the total mass density of the medium. 79
5.10 The refinement in two stages of the simulation. 80

Bibliography

[1] R. A. Gingold, J. J. Monaghan, *Smoothed Particle Hydrodynamics: Theory and Application to Non-spherical Stars*, Monthly Notices of the Royal Astronomical Society, Vol. 181, p. 375-389, 1977.

[2] A. Cohen, *Adaptive Methods for PDE's Wavelets or Mesh Refinement?*, Proceedings of the ICM, Vol. 1, p. 607-620, 2002.

[3] M. J. Berger, J. Oliger, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, Journal of Computational Physics, Vol. 53, No. 3, p. 484-512, 1984.

[4] M. J. Berger, P. Collela, *Local Adaptive Mesh Refinement for Shock Hydrodynamics*, Journal of Computational Physics, Vol. 82, No. 1, p. 64-84, 1989.

[5] M. J. Berger, R. J. LeVeque, *Adaptive Mesh Refinement Using Wave-Propagation Algorithms for Hyperbolic Systems*, SIAM Journal on Numerical Analysis, Vol. 35, No. 6, p. 2298-2316, 1998.

[6] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, C. Packer, *PARAMESH: A Parallel Adaptive Mesh Refinement Community Toolkit*, Computer Physics Communications, Vol. 126, No. 3, p. 330-354, 2000.

[7] D. Meagher, *Geometric Modeling Using Octree Encoding*, Computer Graphics and Image Processing, Vol. 19, p. 129-147, 1982.

[8] R. A. Finkel, J. L. Bentley, *Quad Trees: A Data Structure for Retrieval on Composite Keys*, Acta Informatica, Vol. 6, p. 1-9, 1974.

[9] D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant, J. E. Bussoletti, *A Locally Refined Rectangular Grid Finite Element Method: Application to Computational Fluid Dynamics and Computational Physics*, Journal of Computational Physics, Vol. 92, No. 1, p. 1-66, 1991.

[10] A. M. Khokhlov, *Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations*, Journal of Computational Physics, Vol. 143, p. 519-543, 1998.

[11] H. Ji, F.-S. Lien, E. Yee, *A New Adaptive Mesh Refinement Data Structure with an Application to Detonation*, Journal of Computational Physics, Vol. 229, p. 8981-8993, 2010.

[12] D. Nicholaeff, N. Davis, D. Trujillo, R. W. Robey, *Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units*, 2012.

[13] R. Teyssier, *Cosmological Hydrodynamics with Adaptive Mesh Refinement: A New High Resolution Code Called RAMSES*, Astronomy & Astrophysics, Vol. 385, p. 337-364, 2002.

[14] C. Burstedde, L. C. Wilcox, O. Ghattas, *p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees*, SIAM Journal on Scientific Computing, Vol. 33, No. 3, p. 1103-1133, 2011.

[15] H. Sundar, R. S. Sampath, G. Biros, *Bottom-up Construction and 2:1 Balance Refinement of Linear Octrees in Paralle*, SIAM Journal on Scientific Computing, Vol. 30, No. 5, p. 2675-2708, 2008.

[16] Tobin Isaac, Carsten Burstedde, and Omar Ghattas, *Low-Cost Parallel Algorithms for 2:1 Octree Balance*, Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium, 2012.

[17] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas, *Recursive Algorithms for Distributed Forests of Octrees*, http://arxiv.org/pdf/1406.0089v1.pdf, 2014.

[18] G. Strang, *On the Construction and Comparison of Difference Schemes*, SIAM Journal on Numerical Analysis, Vol. 5, p. 506-517, 1968.

[19] S. T. Zalesak, *Fully Dimensional Flux-Corrected Transport Algorithms for Fluids*, Journal of Computational Physics, Vol. 31, p. 335-362, 1979.

[20] G. Chanteperdrix, P. Villedieu, J. P. Vila, *A Compressible Model for Separated Two-Phase Flows Computations*, ASME Fluids Engineering Division Conference, Vol. 1, p. 809-816, 2002.

[21] I. Suliciu, *On Modelling Phase Transitions by Means of Rate-Type Constitutive Equations. Shock Wave Structure*, International Journal of Engineering Science, Vol. 28, p. 829-841, 1990.

[22] B. Van Leer, *Towards the Ultimate Conservative Difference Scheme V: A Second-Order Sequel to Godunov's Method*, Journal of Computational Physics, Vol. 32, p. 101-136, 1979.

[23] P. L. Roe, *Some contributions to the modeling of discontinuous flows*, Journal of Computational Physics, Vol. 22, p. 163-193, 1986.

[24] R. J. LeVeque, *High-Resolution Conservative Algorithms for Advection in Incompressible Flow*, SIAM Journal of Numerical Analysis, Vol. 33, p. 627-665, 1996.

[25] G. A. Sod, *A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws*, Journal of Computational Physics, Vo. 27, p. 1-31, 1978.

[26] C. Burstedde, D. Calhoun, K. Mandli, A. R. Terrel, *ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws*, Proceedings of the International Conference on Parallel Computing, 2013.

[27] E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, Springer, Third Edition, 2009.

[28] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, 2002.

[29] Donna Calhoun's website, http://math.boisestate.edu/~calhoun/www_personal/research/amr_software/.

[30] `p4est` website, http://p4est.github.io/.

[31] Source code repository for the code used in this paper, https://bitbucket.org/alexfikl/p4estest.