Stony Brook University

Front Tracking and Adaptive Mesh Refinement

A Dissertation Presented

by

Brian B. Fix

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Applied Mathematics & Statistics

Stony Brook University

May 2011

Stony Brook University

The Graduate School

Brian B. Fix

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Xiaolin Li – Dissertation Advisor
Professor, Department of Applied Mathematics & Statistics

James Glimm – Chairperson of Defense
Professor, Department of Applied Mathematics & Statistics

Xiangmin Jiao
Professor, Department of Applied Mathematics & Statistics

Foluso Ladeinde
Professor, Department of Mechanical Engineering

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Front Tracking and Adaptive Mesh Refinement

by

Brian B. Fix

Doctor of Philosophy

in

Applied Mathematics & Statistics

Stony Brook University

2011

Multi component fluid instability problems suffer artificial mass diffusion when numerical methods do not correct for numerical dissipation at an interface. Front tracking provides a sharp interface between components but can be computationally intensive. Adaptive mesh refinement (AMR) is a method of increasing resolution only where needed, which can be more computationally efficient. Under certain conditions AMR can achieve a high resolution numerical solution that would be otherwise unattainable on a uniform grid. In this thesis, we combine front tracking and AMR and apply the new algorithm to fluid mixing problems. A series of timed simulations show that this algorithm can be faster then uniform grid front tracking. The front tracked interface is less diffuse than an AMR calculation at equivalent resolution without front tracking. These results show that combining AMR and front tracking is feasible and the strengths of both methods are retained. The combined algorithm assumes front information resides only at the finest level, simplifying the code and easing the way for future modifications.

Contents

List of Figures

Listings

List of Tables

Acknowledgements

I would like to thank my family who is always with me even when separated. Including, my inspiring, happy and supportive wife Jessica, my understanding, encouraging, and loving parents Thomas and Kathleen Fix, my magnetic, lovable, and cool siblings Tracey and Steven and my grandparents, Grandma Fix, Nana and Pop Pop.

I'd like to thank every teacher who has ever helped me.

I'd like to thank my friends Wai, Thomas, Jose, Derrick, Zohar, Rudy, Rob, Samir, Anthony, Jesse, Kealey, Shirley, Joe, Chris, Ilya, Nick, Jonathen, Ben, Jan, Tony, Ryan, Yan, Tom, Cathleen, Sam, Adam, Juanita, and Nancy.

My adviser and friend Dr Xiaolin Li, for helping me become the scientist and code developer I am.

For their time, patience and comments I'd like to thank Drs James Glimm, Xiangmin Jiao and Foluso Ladeinde.

Chapter 1

Introduction

1.1 General Description

Numerical analysis includes the study of discretization methods for the numerical solution of partial differential equations (PDEs). The ability to compute a numerical solution to a PDE with highly complex physics and geometry has applications throughout engineering and science. Finite difference and finite volume methods have been empirically shown to converge to similar answers for well behaved computational fluid dynamics (CFD) problems using many different algorithms. However there are many applications where computed solutions are off by large factors. There are many reasons for the failure of simulations to agree with experiment. These include, but are not limited to, the use of inappropriate physics models, lack of knowledge of initial conditions or material properties, undetected numerical errors (bugs in the code), and in many cases lack of sufficient resolution for the computation. The focus of this research is the latter. Namely, we investigate computational adaptivity to ad-

dress under-resolved physical regimes. Specifically, fluid instability problems suffer from the inability of a numerical implementation to achieve sufficient resolution for the complex interface structure. To attain the full resolution required for a fluid instability problem in 2D could take days on multiple processors. Yet the answer still may not include all important features of the solution. Furthermore, we are limited by available memory to store such a refined solution. Through studying these types of numerical results, it is clear that in many CFD problems there are significant sources of error due to under resolving important features in small areas of a computation, or to the inherent loss of sharp discontinuity over time due to numerical dissipation.

One approach to improving computational resolution in flow regimes dominated by sharp and discontinuous interfaces is front tracking. The basic idea of front tracking is to focus computational resources at wave fronts. There have been a variety of approaches to accomplish this goal. These include explicit interface tracking [1], volume of fluid (volume reconstruction) method [2], and the level set methods [3]. For fluid instability problems involving two or more materials, a sharp interface between components is essential for obtaining a correct answer. Numerical methods that do not try to correct for numerical dissipation at a fluid interface suffer artificial mass diffusion, which results in a growth rate of instabilities slower than that measured in experiments. Front tracking is a method which represents a fluid interface as a dynamically moving surface in a CFD discretization. This surface separates the fluids at an interior boundary for solvers, allowing for the categorization of the numerical stencils into internal stencil points and stencil points near an interface. Special treatments are taken to ensure that the near-interface stencils will not diffuse

the sharp boundary and maintain high solution gradients near the boundary. The solver can also query which of the stencil points has crossed the interface so that one can apply a different equation of state (EOS) and viscosity to the fluid solver stencil. Even surface tension can be added to the interface.

Another approach to enhance local resolution is to use adaptive mesh refinement. In AMR additional cells are inserted in regions where greater resolution is required such as near shock regions, combustion regions, material interfaces, and vortex sheets. The main idea of adaptive mesh refinement is to provide a refined mesh grid for areas where high resolution is needed and use coarse mesh in areas where low resolution is sufficient. There are several ways to carry out the adaptive mesh refinement including block structured [4] and oct/quad tree [5] [6]. All AMR algorithms require interpolation between consecutive levels so that several physical variables are conserved. The refinement is carried out hierarchically, which makes the interpolation of the embedded solution algorithmically and numerically simpler. The decision as to when and where to refine mesh depends on the rate of change of physical variables. For example, Richardson extrapolation can be used to estimate errors in a given level of mesh refinement. Many AMR implementations allow a user-defined refinement in places such as domain boundaries and specialized geometry sections. It may also allow user-specified gradient cut-offs.

Simulations of multi-component fluid instabilities benefit greatly from the ability of front tracking to supply complicated interface geometries. The interface allows the instabilities to be sharp, and may also introduce extra detail in the state solution near the interface that may not be seen with a method where the interface is more diffuse or smeared. CFD simulations such as shocks

and material interfaces with prominent features that don't consume the entire computational domain benefit greatly from adaptive mesh refinement. However, no refinement will be able to treat an interface as sharply as an explicit tracking of the interface. Front tracking may introduce a discontinuous jump in the state solution across the interface. Therefore combining front tracking with refining the mesh around the interface is a natural idea. Enforcing refinement near the fluid interfaces would allow for more exploration of the nature of fluid instabilities. The addition of AMR to front tracking allows for higher resolution studies with more complicated initial geometries and fluid features.

To couple AMR and front tracking many considerations must be taken into account. Tracking a dynamically moving interface using topologically linked marker particles is challenging. Much care is taken in front tracking code to ensure topological consistency. Ensuring the correct behavior near boundaries and across parallel subdomains is also an important in front tracking. Front tracking is tightly coupled with the underlying state grid for interpolation of state values at and around the interface. To ensure that coupling takes place in a logical way, the front tracking algorithms must have proper access to grid states. Adaptive mesh refinement has its own computational difficulties. It is advantageous to use solvers that are well vetted on single grid calculations. These solvers require large grid blocks for maximal efficiency. Block structured AMR is a method which requires that refined regions exist in patches, or blocks larger then a certain size, in order to maintain the efficiency for single grid solvers. A block structured calculation is made up of nested levels of patches of refinement. Patch sizes have restrictions placed on them to ensure efficient calculation times.

Proper coverage of areas where refinement is requested is also enforced. To couple front tracking with AMR, we use block structured AMR. We enforce sufficient refinement near the interface so that the interface always exists inside the finest level of refinement. The front tracking algorithms and related solvers can then operate as they normally would on a regular grid. The only front tracking operation required after this point is to make sure proper data management of the interface is provided at patch boundaries. Extra AMR functionality is needed to ensure that component information on coarser patches is updated from overlapping finest level patches that have an interface. The functionalities needed of AMR and front tracking are briefly described in this introduction. Their implementations requires a detailed code to control several steps of the process. These details will be elaborated upon in this thesis.

The perturbation in a fluid instability starts from a small area in the computational domain with smooth state solutions on either side of the computation. The number of cells of the mixing zone in which the interface exists can grow exponentially. AMR can reduce the amount of calculation in early states of the simulation, when the interface still resides in a fraction of the computational domain. This allows calculations to progress to later time without spending too much resource for early time. Many physical processes important to engineering and science start with the interface residing in a very small region and evolves into complicated late time mixings covering a larger part of the domain. Applications of this code are useful in simulations of supernova, implosions, and injector jet spray. Another area where AMR and front tracking shows promise is shock interface interactions, due to the relatively unperturbed state in the initial condition where only those cells near the shock

and the interface need refinement.

AMR and front tracking are both advanced techniques in CFD in their own right. This thesis proposes algorithms to couple these technologies. We explain how these algorithms are implemented in a software framework. Verification and refinement studies show promise for the coupling of these techniques. Our research opens the door to simulations which could not have been done previously, and should be useful for future numerical experiments.

1.2 Contributions

Work on combining front tracking with adaptive mesh refinement has been worked on by several different investigators. Some basic data structures and initialization exist in the FronTier code, but combining these algorithms is a challenging task. The current work is a continuation of the work done by by Zhiliang Xu. Xu developed FronTier and AMR with a different software package, namely the Overture package [7]. The Xu approach uses the Overture AMR algorithms for the decisions of where to place refinement grids. All other aspects of AMR are taken care of inside the FronTier gas dynamics code. Xu's AMR code was developed in an isolated branch of the FronTier code without version control that was not remerged. It took three years to get the Xu approach working in the current FronTier development tree.

The development of a new light weight compressible solver based on the FronTier-Lite code, cFluid, presented the opportunity for an alternate approach. We combine the cFluid code with the SAMRAI AMR package. The SAMRAI routines take over much of the AMR related work. The new com-

bined front tracking AMR algorithm has proven more robust and easy to use. The combined algorithm changes the interface model by only allowing access to the interface geometry at the finest level. This reduces the amount of specialized AMR routines. This reduced amount of specialty code requires only a few functions that are dimensionally dependent. Much of the code in the combined algorithm is modularized and already usable in two or three dimensions. Due to the new algorithm and modern software design, this work furthers work towards AMR front tracking in three dimensions.

Chapter 2

Background

2.1 Front Tracking

Numerical modeling of interfaces can be done in several ways: particle methods, PLIC-VOF, level sets and capturing. Front tracking relies on marker particles, and of the four groups of methods mentioned above, it is closest conceptually to the marker particle methods. It differs from marker particles in that the particles are located only on the interface, rather than in a volume region near the interface, and in that the particles are connected to each other to form a triangulation (3D) or piecewise linear (2D) description of the interface. It is significantly faster than particle methods, since fewer particles (one or two in 2D) are used per cell in front tracking than the number used (4^D) in typical particle method simulations.

The front tracking method has showed its advantage in the computation of several important physical problems such as the study of fluid interface instabilities [8–12], providing the first or the only physically validated simulation

for some important fluid instability problems.

The front tracking method provides several algorithms for the redistribution of marker points on curves (2D) and surfaces (3D). The most frequently used method in 2D is the equal-bond redistribution. This function measures the total length of the curve, which is then divided by the optimal bond length to obtain N, the total number of bonds. The new bond list is created with equal length starting from the first point (node).

In three dimensions, both the area and the aspect ratio of the triangles are calculated and those with low quality are placed in a queue. A cyclic optimization procedure is called for elements of this queue to insert, delete or re-triangulate the triangles until all the triangles satisfy a preset geometrical criterion. Each of the optimization functions performs a complete set of operations to guarantee that the topological linkage of the interface is correct after its operation.

Front tracking starts from a set of discrete marker points, topologically organized through an interface data structure. The method provides a set of functions to maintain its organization after dynamic propagation and bifurcation [13–15]. For use below, we call this method grid free (GF) tracking, as the interface handling has no logical relation to a finite difference grid. The difficulties associated with topological bifurcations for GF tracking are amplified in 3D. As a result, a robust semi-Eulerian reconstruction method, called grid based (GB) tracking [16], was introduced. GB tracking, although robust, suffered from excessive interpolation and smoothing errors, the same inaccuracies as the level set method. A new method which combines the best features of GF and GB tracking, is called the locally grid based (LGB) front tracking

method.

To reduce the GB interface interpolation error, we use the LGB, or the locally grid based tracking, which combines the advantages of both methods. We use the fully Lagrangian GF method to propagate the interface to obtain an accurate solution of the interface position. Eulerian GB reconstruction of the interface is only used in small regions where topological bifurcation is detected. The detection of topological changes is through a fast algorithm which walks through the Eulerian grid to check the consistency of the indices assigned to grid nodes of each subdomain and the corresponding side of the interface. In the first step of the procedure, the intersections between the interface and cell edges of the Eulerian grid are inserted and the index of every subdomain is assigned to each corner point of the Eulerian grid. We then check the consistency between the indices of each crossing point and the node point it faces. If inconsistency is detected, the corresponding mesh block is recorded. This intersection detection algorithm is approximate in that it will miss bifurcations totally internal to a single mesh cell.

The construction advances through four steps.

1. Those recorded blocks will be assembled to form boxes and overlapping boxes will be merged.

2. Surgery is performed within each box. Triangles crossing the box boundaries are recorded for later use. All the triangles inside the box or attached to the box will be deleted leaving only the crossing points of the interface and grid edges. A grid-based reconstruction is followed to build the new section of the interface inside the box.

3. The triangles totally outside the box form the exterior interface.

4. The region between the exterior and interior interfaces is re-triangulated to join the two smoothly.

Figure 2.1 shows the procedural steps of the local reconstruction of the interface. The reconnection step 4 is the most crucial step in the LGB method. We modify the triangles recorded in step 2 by a series of steps to make the triangles in the reconnection region also grid based relative to the box boundary. These substeps of step 4 are summarized as follows:

4.1 We split triangles crossing the box boundary which are recorded in step 2 using the intersection points between triangles and box boundaries. There are two types of intersection points: one is the intersection points between triangles and grid cell edges on the box boundary (TYPE I), the other is the intersection points between the sides of triangles and the box boundaries (TYPE II). We first divide each triangle by recursively inserting the TYPE I intersection points (if any) inside the triangle. Each triangle will be divided into three smaller triangles by joining the TYPE I intersection point with the vertices of the triangle. Then we insert the TYPE II intersection points recursively. Each triangle is divided into two smaller triangles by joining the TYPE II intersection point and the vertex opposite to the side containing this intersection point. After this step, the triangles which crossed the box boundaries are split into triangles lying either entirely inside the box or entirely outside the box. We keep only the triangles entirely outside the box. These triangles meet

the box surface along a closed curve which is actually a piece-wised linear curve connecting all the intersection points.

4.2 The curve gives an ordering to these triangles, and in this ordering, we merge triangles whose vertices are TYPE II intersection points until there is no TYPE II intersection points in the curve. After this step it is sufficient to examine that each triangle meets the box boundary only as a line joining adjacent grid edge. i.e. the triangle meets the box in a grid based manner.

After this operation, all triangles will meet the box edge only as a line joining adjacent grid edges. That is, all triangles, inside and outside, meet the box only in a grid based manner. The reconnection between the outside triangles and the newly reconstructed triangles inside the box is then a simple match of the triangle sides at the box boundaries.

This method reduces the use of the Eulerian reconstruction to a minimum. It is particularly useful for the computation of interface motion in which the interface has regions of large curvature. It reduces interface interpolation errors and minimizes the unphysical disappearance of the fragmented components of the material after bifurcation.

2.2 Multi Component Fluid Dynamics and Front Tracking

The Euler equations of compressible inviscid fluid dynamics are:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{v} = 0$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot \rho \mathbf{v}\mathbf{v} + \nabla p = 0$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot \rho \mathbf{v} E = -\nabla \cdot p\mathbf{v}$$

Where the dependent variables $\mathbf{v}$ ρ, p, and E denote, respectively, the velocity, density, pressure and total energy. With:

$$\frac{E = e + \mathbf{v}^2}{2}$$

and e the internal energy. For the purposes of our research we are interested is a split solver TVD method [17]. We use this method to solve these equations and we apply the multi component ghost fluid method that follows.

The states at the interface are coupled with the fluid states on the computational grid. In a single time step of a front tracking simulation, there are two major function calls. First the interface is propagated using interpolated velocity from the interior. Second, after the interface propagation, the interior variables are updated. This update is different from an untracked Euler update. During a sweep, the state vector along the direction of the sweep is broken up into subsections based on their components which are separated

by the interface. See figure 2.2. These subsections are solved independently with corresponding boundary conditions applied at each end. These boundary conditions at the ends of the subsections either use traditional boundary conditions such as Neumann or Dirichlet boundary conditions at the domain boundary, or interface boundary conditions from an interface coupling algorithm. In a multi-component fluid simulation, the ghost fluid method (GFM) [18] is applied at the internal interface boundary. These two function calls are repeated as time advances.

2.3 AMR

Adaptive mesh refinement comes in different forms. Different numerical methods employ different algorithms and use different data structures to store data for the discretization. Among all numerical methods for PDEs the idea for AMR is the same: to add more data points/cells/memory for higher resolutions only in regions where it is needed. We use finite difference and finite volume methods on rectangular meshes.

On regular rectangular meshes there are two types of AMR, continuous and patched based. They share many common traits. Both methods start with a regular coarse Eulerian mesh. Finer cells are subdivided from coarser cells. Each cell is divisible into an integer subdivision of finer cells. Cells that share a common cell size are on the same level. In both methods successive levels of refinement are hierarchical. This means that the Nth level is embedded in the N-1st level.

In continuous AMR, the algorithms are designed to make a decision on a

single cell. If a cell needs refinement it will be refined. The process is then repeated in each sub cell at every step [5] [6]. This allows for cell by cell refinement. Refinement rules allow refinement in a cell to be one level higher than its surrounding cells. In combination these rules can give a high quality continuously varying mesh. [See Figure2.3.a] One of the disadvantages of this method is that solvers must be written for the data structures to perform updates to cell variables.

In patch based AMR, the initial coarse grid is swept and the cells which require refinement are tagged with a boolean value. After the sweep, patches are placed over the coarse grid to cover all regions in which refinement is needed. The finer grid placement strategy is meant to optimize patch size and location so that each patch is covers a high ratio of tagged cells to untagged cells. These patches must also obey constraints on size, falling between a minimum and maximum size in each direction. The sweep and placement of finer grids is then repeated on each fine grid.

The decision on where to refine is usually a responsibility of the user of the AMR code, and dependent on what the user of the code is interested in. One could choose to refine cells based on a variation of a single variable, such as the gradient, or simply a threshold value. Error estimates such as Richardson extrapolation can also be used for refinement criteria. In shock physics, the refinement is usually decided based on a cut-off value of the pressure gradient. AMR refinement criteria is an active area of research. Many simulations using AMR do empirical studies at low resolutions using a mixture of the above mentioned methods. Results are compared between AMR and fully refined results in order to determine effective AMR refinement criteria to use for production

runs at higher resolution.

Initialization of the two methods are similar. Because the initial conditions are usually set by analytical functions, the initialized grids contain the exact state information. Given a refinement decision function, initial shocks or interfaces can be fully refined with exact information, along with any other areas of interest.

Given the hyperbolic nature of compressible Euler equations, initial waves will propagate away from the initially refined regions. If the initial condition is refined properly and the refinement options are chosen carefully, areas of interest in calculations can have proper refinement throughout the entire simulation. In these cases it is possible to have an AMR refined calculation containing the same large scale structures, such as shock and interface position and speed, as a fully refined single mesh calculation.

2.3.1 Patch Based AMR

The original descriptions of hyperbolic systems of PDE's and a patch based AMR scheme is given by Collella [4] [19]. In these papers Collella and Berger define a patch description and the integration algorithm. Patches are placed on the grid in a logically nested manner and a level of refinement must be embedded in a patch of refinement only one level coarser.

One advantage of using patch based AMR for this work is that verified solvers can be easily inserted into this system. Patch based AMR allows for a software division of AMR and solver so that AMR libraries can be written independent of solvers. This allows for easy reuse of AMR libraries, which

is another advantage. Patch based AMR libraries include routines for data management, input/output, visualization, and parallelization of patches. On a numerical level these AMR libraries also include functions for interpolation, and integration depending on the type of PDEs and numerical schemes used.

With hyperbolic PDEs of the form:

$$\frac{\partial w}{\partial t} + \sum_{i=1}^{d} \frac{\partial}{\partial x_i} f_i(x,t,w) = g(x,t,w), \qquad w(x,0) = w_0(x)$$

it has been shown that a numerical scheme with mixed cell sizes can have mixed time steps instead of a global bound on timestep size [20]. The Berger Oliger integration algorithm deals with the fact that due to the CFL condition, smaller steps can be taken on finer patches. The algorithm is recursive and insures that a series of substeps are taken on a given level for each step of the next coarser level. This recursive algorithm is first applied on the coarsest level and it dictates the dt calculation and the update of solution on all levels. The outline of this algorithm is shown in Listing 2.1. In Figure 2.4 these different stages and recursion are depicted in a simple example with 3 levels, with half dx on every finer level. Assuming the velocity values on the coarse grid and fine grid are about the same, then for every level to sync with the next finest level, the next level will take approximately two steps. If the coarse grid takes a step of size dt_{coarse}, then for the medium grid to sync with the coarse grid it must take two steps of size approximately $1/2 \times dt_{coarse}$, The finest grid must take two intervals of two steps, of size $1/4 \times dt_{coarse}$ In this case the order in which the steps in our simple example in Figure 2.4 would occur would be A, B, C, C, D, B, C, C, D, E .

The hyperbolic AMR algorithms assume that state data on the finest level is the most accurate, and it is conservatively coarsened to all coarser grids. The coarsening from fine grid state to coarse grid state means that previously calculated coarse state data is replaced with a new fine grid solution . In addition, a flux correction on the coarse grid at the coarse/fine boundary is necessary for the reason of conservation (see [20] and [21]).

```
Recursive_Integrate(level_i)
{
        repeat hj/hc times
        if(regridding_time)
                calculate_errors_for level i and finer
        step dt on all grids at level i
        if level i+1 exists
                Recursive_Integrate(level_i+1)
                update(level_i, leveli+1)
}
```

Listing 2.1: The Berger Oliger Algorithm

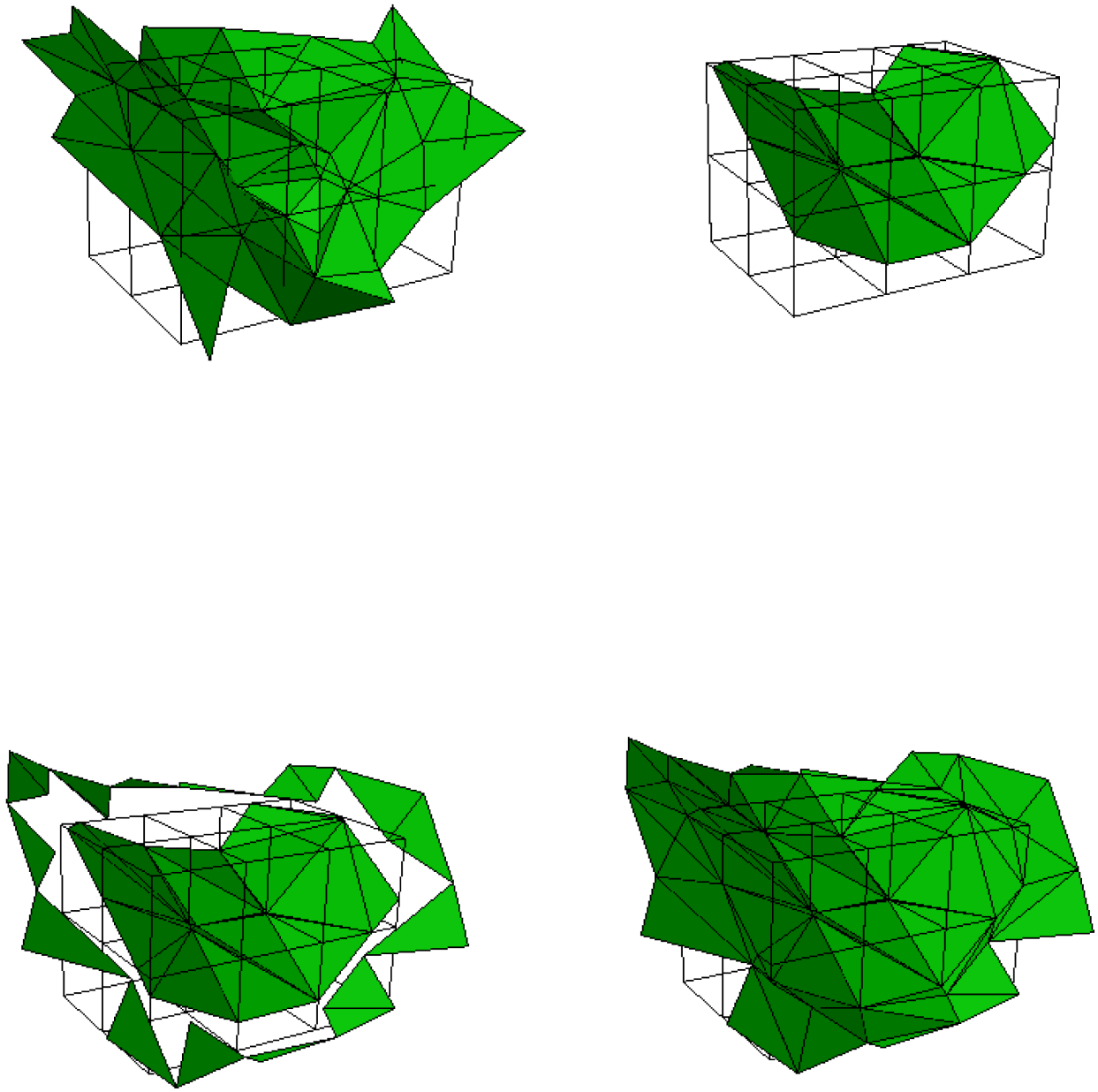


Figure 2.1: LGB Algorithm steps. Steps to reconstruct a tangled section of the three dimensional interface. From left to right and top to bottom: (1) assemble blocks which contain unphysical edges, (2) delete triangles attached to the box and rebuild the interface through the grid-based method, using the grid-based method to reconstruct the interface topology inside the box, (3) align triangles inside the box and outside the box, (4) relink the interface topology for triangles inside and outside the box, and thereby obtain the final interface with new topology. Images courtesy of Yuanhua Li.

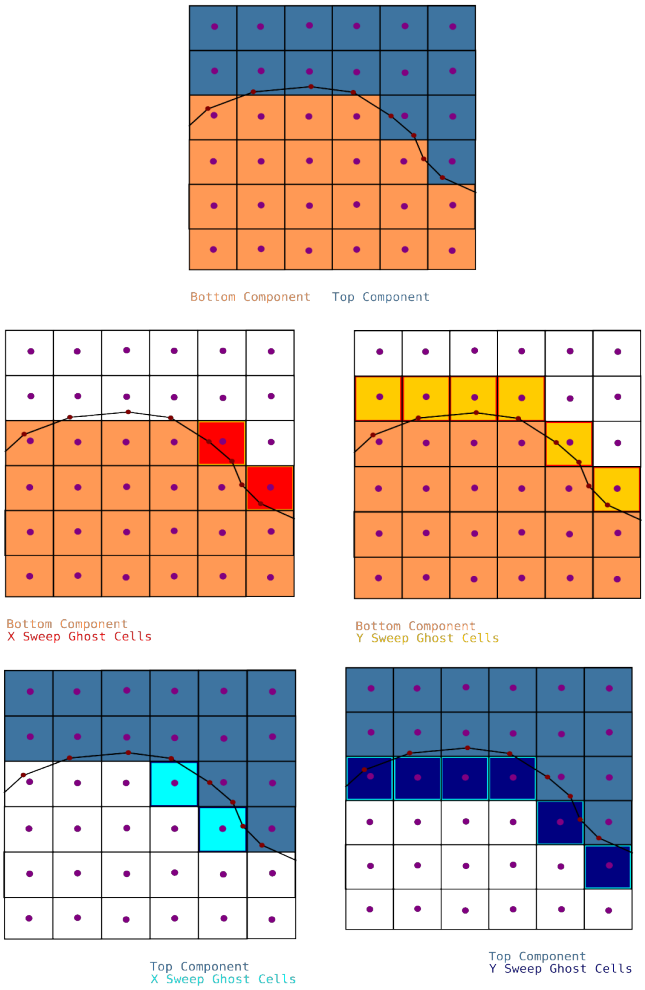


Figure 2.2: Ghost cells of an interface. An interface and ghost cells for a stencil size of one, in different sweep directions.

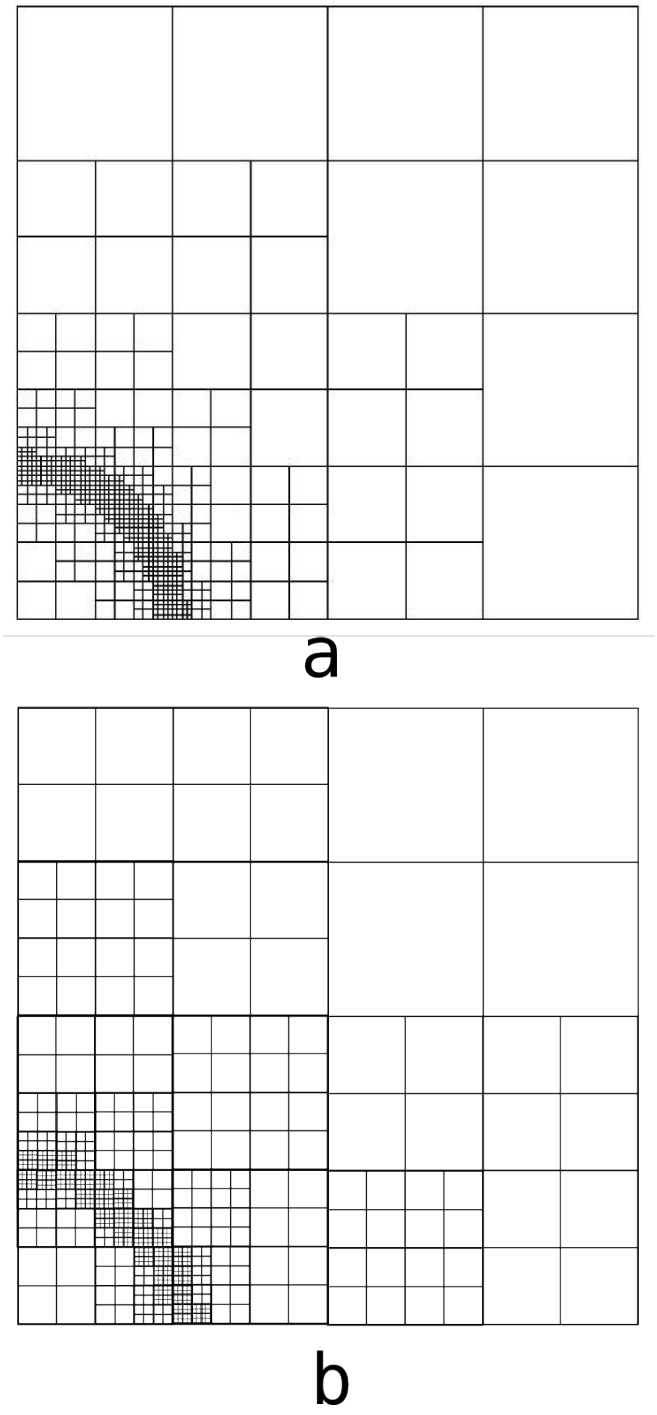


Figure 2.3: AMR types. a. continuous AMR b. patch based AMR (continuous AMR image from [6])

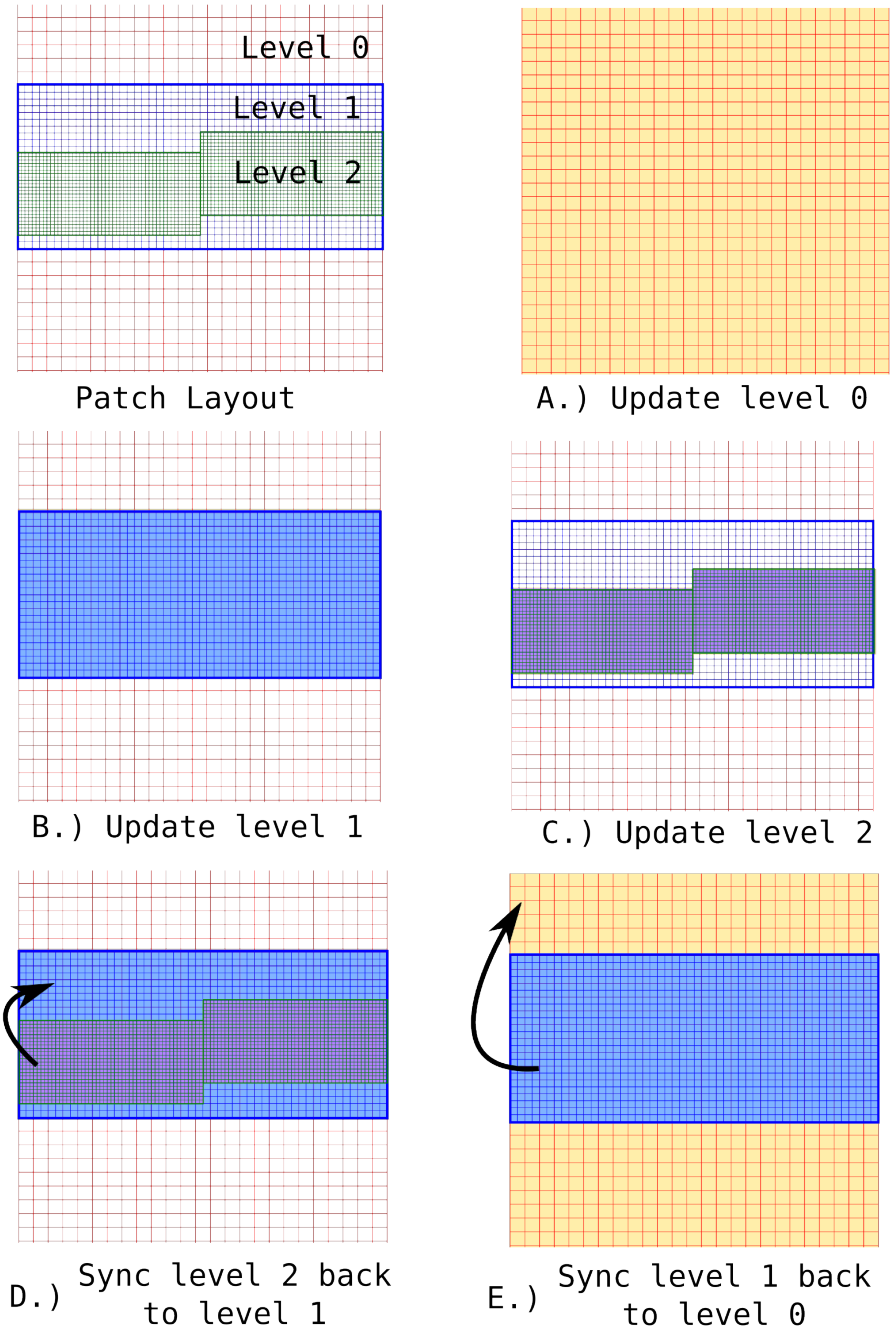


Figure 2.4: Simple AMR example

Chapter 3

Gas-AMR

3.1 Combining Gas and AMR

Chapter 2 gives the history and background of the FronTier code. One of FronTier's premier applications is the `gas` package, designed as three distinct parts, `gas`, `hyp` and `driver` to be used in unison. The `driver` package contains a set of general purpose main loop functions to coordinate different parts of the simulation including several types of solvers, output formats, state storage, and front tracking maintenance calls. The `hyp` package provides the general purpose hyperbolic solvers and state interpolation functions for hyperbolic solvers. The `gas` package initializes compressible fluid dynamics problems, provides EOSs, Riemann solution and ghost-cell method code, and provides other physics options such as mass diffusion, viscosity, phase transition, as well as problem specific extra statistics.

Adding AMR to FronTier's `gas` has been progressing since 2000. This work was carried out mainly by Zhiliang Xu and his co-workers [12, 22, 23].

The approach was to change code in all sections of the `gas` package to make it AMR compatible. This work will be referred to as the `Gas`-AMR approach.

The `Gas`-AMR approach makes three important assumptions:

- The most up to date `front` geometry is needed on all levels.
- A full FronTier GFM calculation is carried out in all patches and on all levels.
- The `front` is propagated with the most accuracy in the finest level of patches.

The decision was made to parallelize the implementation which complicates the data and functional structure of `Gas`-AMR. The parallel model for `Gas`-AMR is an adaptation of the original FronTier data structure. It is essentially a modified spatial domain decomposition approach. If there are P processors in a two dimensional domain, this computational domain is partitioned into $N \times M$ pieces where $P = N \times M$. If the coarsest grid is in a $I \times J$ mesh, then each processor has a coarse grid computational patch $I/N \times J/M$. This coarse grid is also referred to as the base patch. All finer patches are built on top of the base patch. They are created and managed by the base patch's driver function.

After all fine level patch `front`s are advanced via the propagation and redistribution functions, these patch `front`s are assembled to populate the `front`s on all coarser levels. The fragmented interfaces are propagated in the finest patches. They are then sent to the base patch for assembly. Once there they are stitched together to make up the interface in the base patch. The

base patch `front` is then copied and cut to fit all other finer patches in the computational subdomain. When this procedure is complete, every level is updated with a full `front` covering the patch, containing correct geometry and topology information. See Figure 3.1

This procedure is repeated in every timestep and is used for the FronTier GFM calculation. In these procedures, the `Gas`-AMR uses a unified time step across all levels, and all coarser levels take the timestep satisfying the CFL condition of the finest grid. Regridding is enforced after a given number of time steps, which is a user defined input.

FronTier load balancing is optimized by distributing patches among processors. After each regridding, all data is returned to each base patch and the external AMR library functions are called to make the decisions on patch placement. A FronTier function is then called to balance the computational load by sending patches from heavily loaded processors to lightly loaded processors. See Figure 3.2 for an intuitive explanation.

3.2 The Software Maintenance

The `Gas`-AMR approach started in the early 2000's by Zhiliang Xu [12, 22, 23]. We continued Xu's AMR work by inserting his early implementation into the main FronTier version. We adopted the version control system named Mercurial [24] to track all the changes in both the FronTier and the AMR libraries. The use of Mercurial has accelerated the code development and merging, allowing us to revert changes when an incorrect addition has crept into the system. After the merge was completed, the `Gas`-AMR code was

still not compatible with the new system. We removed many bugs in the interface clipping and parallel communication which occurred in standard test problems. The Gas-AMR code was functional after all these changes. See Figures 3.4 and 3.3 for an example of the Gas-AMR simulation of a jet and an implosion problem.

3.2.1 Software Metrics

The use of a version control system makes it possible to determine precisely what is added to the code. In all, Gas-AMR has added 28 files and changed another 144 files. We have made total of 56,109 line insertions and 8,538 deletions of the code. Another code counting metric was used called Source Lines of Code (SLOC), via a script called SLOCCount[25]. The FronTier code started with 361,859 SLOC before merging and ended with 387,927 SLOC, a difference of 26,068 SLOC and a growth of 7%.

The industry standard for errors per lines of code is 15-50 per 1000 lines of delivered code. Corporations that make their primary profit from software write code that has 10-20 errors per 1000 lines at development stage before testing. After formal bug tracking and unit testing this error rate drops to .5 per 1000 at release [26]. The rate of .5/1000 errors/SLOC implies a tolerance of 13 errors in the Gas-AMR code. With a rate of 4/1000 errors/SLOC the number of errors in the Gas-AMR code is over 100. The number of successful tests and production runs combined with the volume of code added to the software library can be used as a measure for the error rate per SLOC in Gas-AMR section of the FronTier library.

3.3 Development Issues of the Gas-AMR code

3.3.1 Design Issue

The assumptions in section 3.1 on front tracking in AMR led to some difficulties in making full use of the AMR libraries. The requirement that `front` must be included at every level of the refinement patches makes it incompatible with AMR libraries in parallel mode. As the result, the Gas-AMR attempts to re-implement many routines that should be left to the external AMR libraries. AMR libraries are used in the Gas-AMR code only to make decisions such as to where to do the regridding, etc. Many features of the AMR library cannot be used without a full re-implementation. The interoperation is rigid and limited.

The Gas-AMR model contains function calls to clip, send, receive, and stitch patches in its AMR framework. The request for full front geometry at every level requires the code to synchronize interfaces between patch levels. Gas-AMR adds several files containing data structures and functions only to maintain a collections of `front`s in the AMR data model. Much of the code written around this model is not modularized.

3.3.2 Programming Problems

FronTier, when not coupled with AMR, operates on one piece of `interface` at a time with an old copy and a new copy of the `interface`. It swaps between the old and new `interface`s in each time step. To ease the difficulty in accessing the data structure, a pointer to the current interface is set as a global address. Global pointers and flags, such as `cur_intfc`, impose no difficulty when there

is only one piece of `interface` per MPI process. However, when a process is dealing with multiple patches, the program must insure that these global pointers and flags are consistently maintained and set. Any change in the code that does not take into account these issues can lead to serious errors.

The programming features in the `gas` code are useful in the complex FronTier operation, but come at the expense of being too tightly integrated with all aspects of the code. The `hyp, driver, gas` framework imposes programming caveats and extra dependencies, which makes the system less general. Because the `gas, hyp, driver` system was written in C and divided into separate directories, function pointers and nested data structures tie them together. `Gas`-AMR integration introduces more pointers, more levels of nested data structures, and more code to maintain these data structures.

The combined system also contains non unique macro and function names. Name collision on variables and functions need to be detected and resolved. Renaming all non unique function names in the `gas` framework was necessary in order for these functions to work correctly and serve as subroutines to other packages.

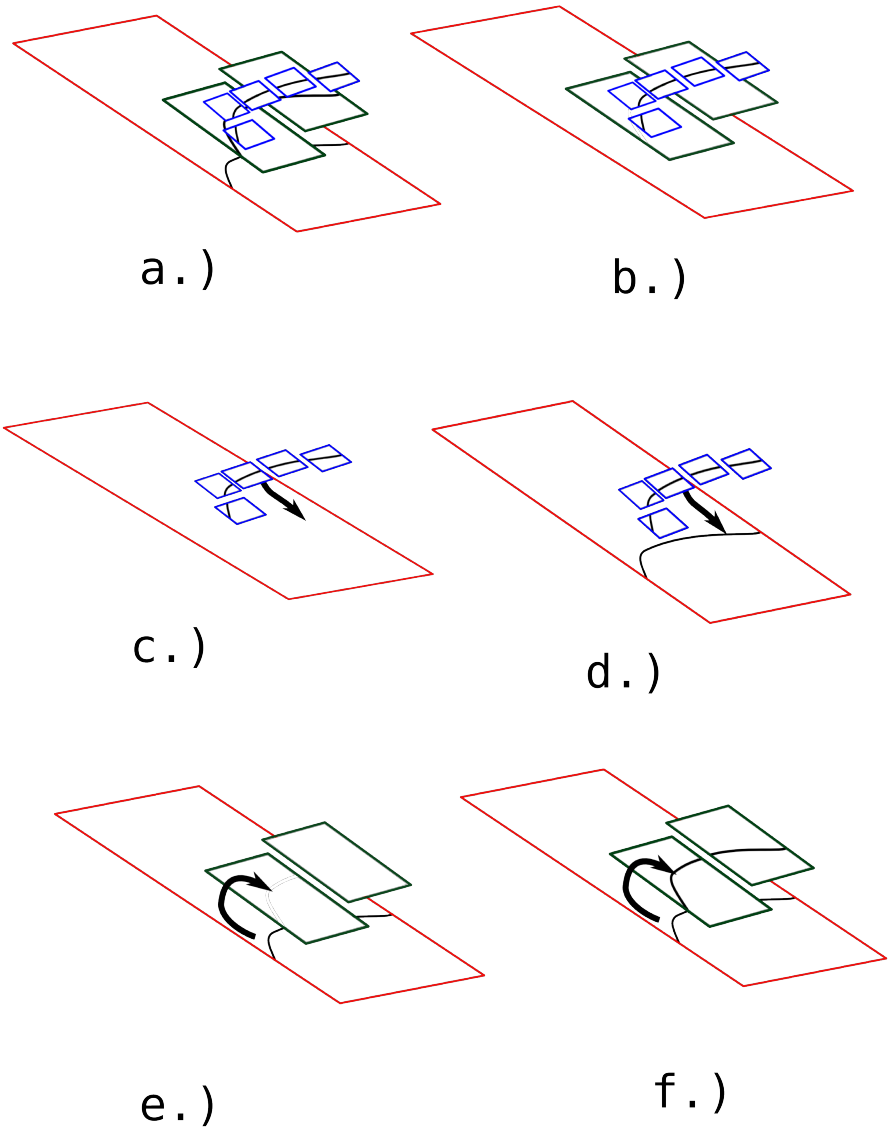


Figure 3.1: Gas-AMR patch communication. a.) Original patch configuration b.) Interface is removed from coarse patches c.) Finest level interface is sent to base patch d.) Finest level interfaces are assembled on the base patch e.) The base patch is copied to all other non finest level patchs f.) The copied base patch is cut to all non finest patches

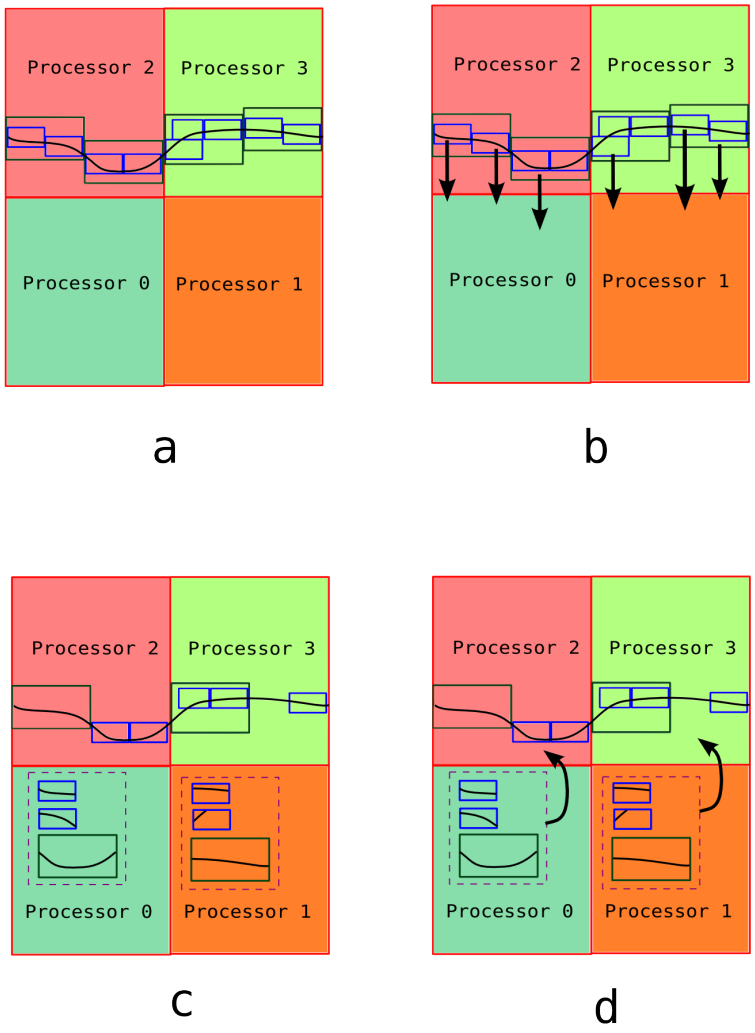


Figure 3.2: Gas-AMR load balancing. a. Original patch configuration b. Load balancing calculation determines where to send send patches c. Patch state update and front propagation occur d. patches are sent back to base processor

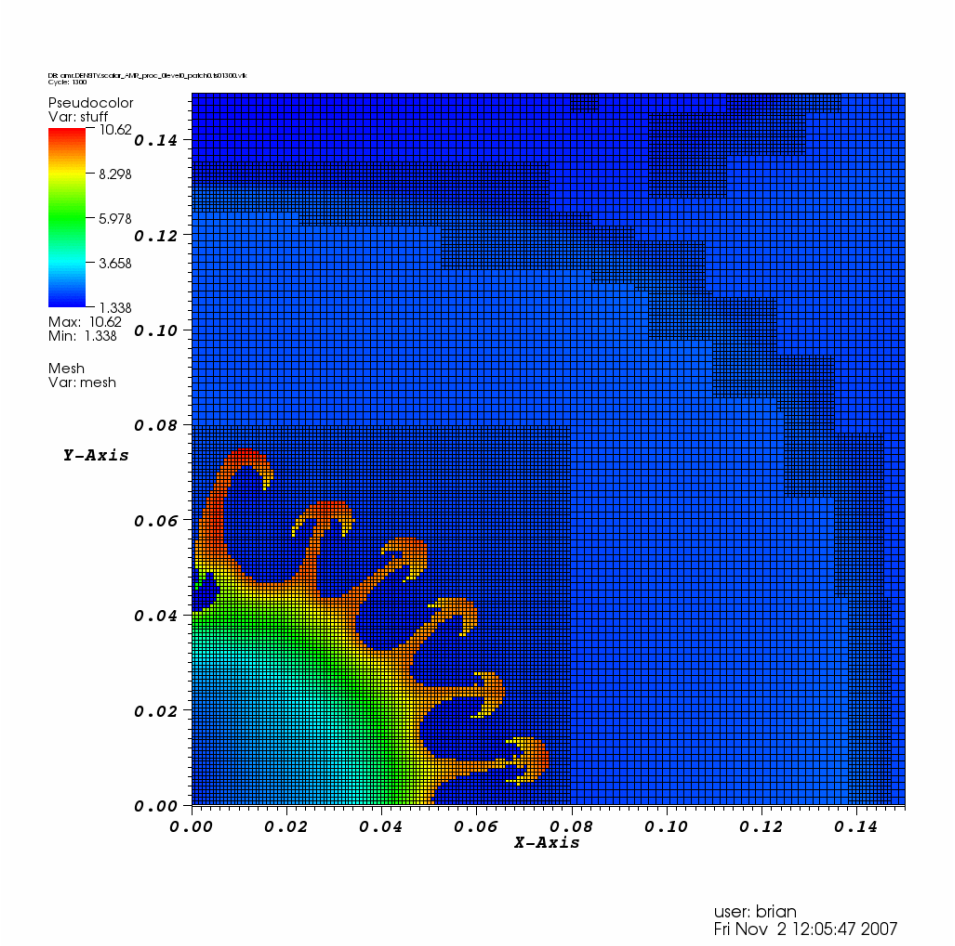


Figure 3.3: Gas-AMR implosion simulation

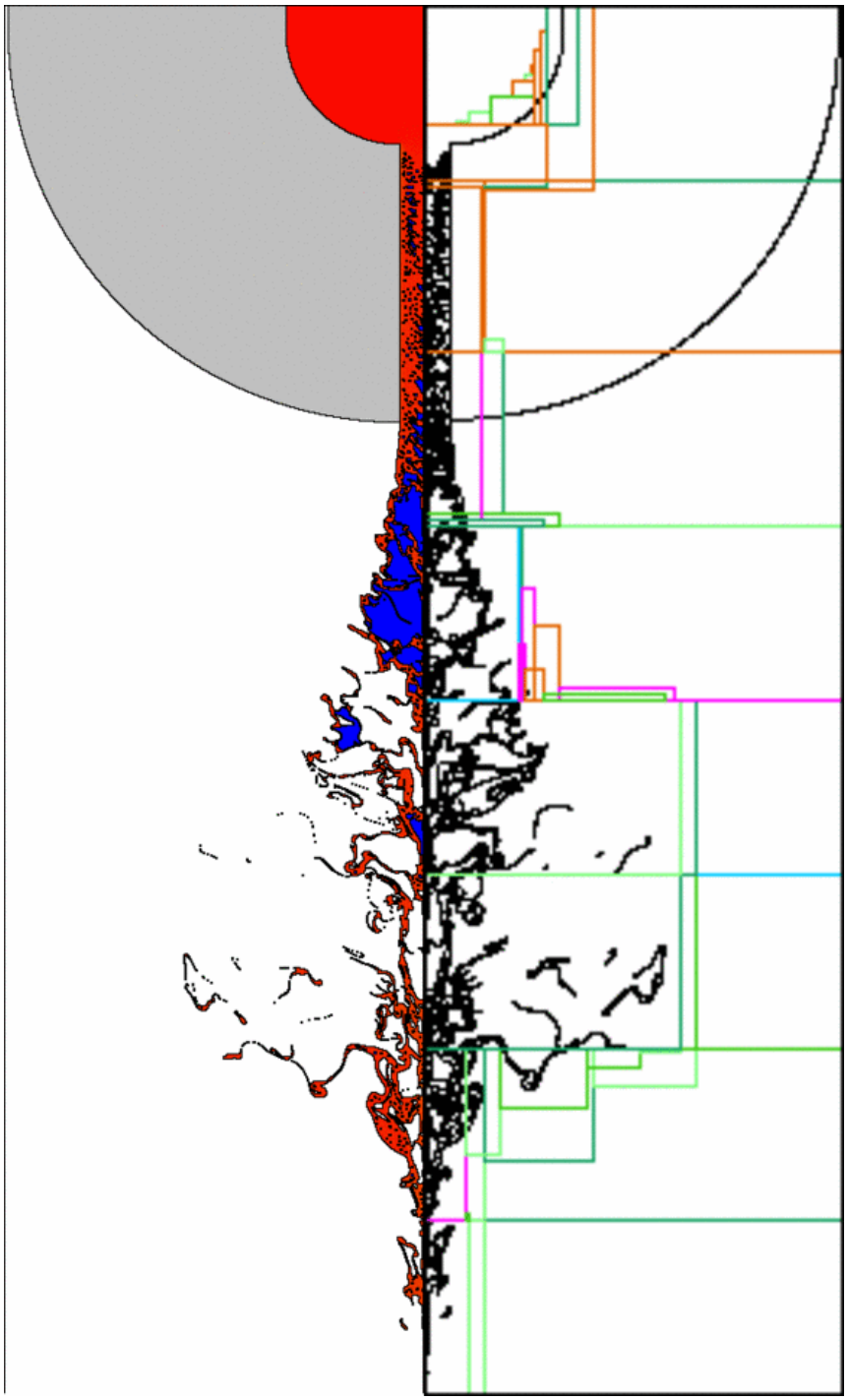


Figure 3.4: Gas-AMR injector jet simulation Image courtesy of Zhiliang Xu

Chapter 4

cFluid & AMR

4.1 The New Algorithm

The key to understanding how the new front tracking algorithm couples with patch based AMR is to have a good understanding of patch based hyperbolic AMR schemes and the integration algorithm.

To do front tracking with AMR we make a few assumptions. One is we want the interface to exist on the finest level all the time. If we do not do this, front tracking becomes extremely difficult, because the front tracking repair and redistribution algorithms assume bond and triangle size that has a minimum and maximum on the same order of magnitude. If the interface was allowed to cross between areas that only had a coarse grid and areas that were fully refined, the difference between minimum and maximum size bonds would be enormous. Specialized and possibly unstable algorithms would have to be developed to deal with this. AMR allows for easy user defined refinement, and it is no trouble to ensure full refinement around the interface. This alleviates

this potential problem and allows the interface to live on a uniform grid size.

Combining the above assumption about the interface always being on the finest level, with our knowledge about AMR hyperbolic integration schemes, we can arrive at another useful assumption. The recursive state update on patch levels means that on every level except the finest, the portions of state information that overlap with finer patch levels are replaced with the fine level state data. The ghost fluid method at a time step only effects the solutions of cells within the stencil length distance from the interface. If region within the stencil length of the interface on all levels is always maximally refined then the ghost fluid computation only has to be done correctly at the finest level, because all other data in this region comes from the finest patch. This allows us to make the interface only accessible on the finest level.

Restricting access to the interface to the finest level is optimal because keeping interface information current on different patch levels requires lots of computation and communication. These computations and communications also require complex interface surgery. Due to the large amount of code required for this functionality, this functionality is very prone to mistakes.

Front information is used elsewhere in the calculation besides the ghost fluid computation. Mainly, the front provides a way to get cell by cell component information for multi component fluid operation. This is required to apply different EOS information or other physics that is component dependent. This component data is needed on all patches even far from the interface. To provide this data we add one extra scalar field for component data to the AMR software framework. To keep the component information correct, it must be initialized correctly. Then when the finest level patches propagate the front,

they also update component information. The AMR package is responsible for data replacement on coarser levels. This replacement also updates the component data on all coarser patch levels, using standard prepackaged data coarsening algorithms.

Instead of doing a normal interior state update on a patch at a time step, front tracking specific control code directs a sequence of events to do a ghost fluid calculation. If on the finest level, the front is propagated. If not on the finest level, AMR component information from the AMR coarsening routines is used. Then the component information is used to calculate the new interior state values with the ghost fluid method. The above description of this leads to a modified FronTier enabled Berger Oliger Algorithm See Listing 4.1.

The AMR libraries are written for parallel operation. Patch data is distributed by the AMR library's load balancing algorithms to different processors. In order for the FronTier interfaces to exist on the finest level, FronTier code must be written to cut the interface into pieces that correspond to the patch domains, and distribute them in the same manner as the patches.

4.2 Software Choices

The ability to combine AMR and front tracking depends on the ease of use of two fairly complex libraries, and making them coexist in one program without much linking and namespace collision trouble. The new approach takes heavy advantage of FronTier-Lite design philosophy, which re-factors a large portion of the FronTier code as a library that is callable by other codes.

The gas package in FronTier does not meet several requirements to be

integrated into another software package. It is not a standalone package, but instead depends on several other directories such as driver hyp and tri. Designing a simple way to make few calls to the gas/driver/hyp system without writing a complex supporting code would be difficult. Function pointers, nested data structures, and legacy code to support unused options, change how the code was designed, and runs. This makes integrating gas into another package challenging.

Instead cFluid was used. cFluid is a light weight compressible CFD package written in C++ which has front tracking capabilities using FronTier. cFluid is written in an modularized object oriented design. It lacks function pointers and associated macros to hide them, as well as the complicated initialization code. cFluid uses standard C++ memory management. cFluid only depends on standard FronTier-Lite with no external dependencies.

SAMRAI is an acronym for Structured Adaptive Mesh Refinement Application Infrastructure. The SAMRAI library consists of a collection of classes that are meant to be combined for use in any numerical PDE calculation requiring Structured Adaptive Mesh Refinement (SAMR) calculations. Structured AMR refers to the fact that the overlapping grids are regular structured, and in most cases cartesian grids. However, cartesian patch based AMR is a sub classification of SAMR. These structured meshes are of different coarseness and are overlapped in a domain to cover it. Figure 4.5 shows an example of such an overlapping grid.

SAMRAI has a rich code base with many useful capabilities. SAMRAI can be used to restart a multi processor MPI simulation with a different number of processors. Class hierarchies exist to facilitate AMR calculations for hy-

perbolic, parabolic and elliptic calculations. SAMRAI provides state of the art load balancing and data placement algorithms that make SAMRAI scalable to thousands of processors. SAMRAI provides easy to use input files and documentation to easily change AMR decisions. User variables include number of levels, clustering efficiency, refinement, ratio per level, min/max patch size, refinement choices such as Richardson extrapolation, and per data variable refinement decisions such as cut-off, gradient cut-off, and shock detection. SAMRAI also provides extensive example codes and documentation that allow for quick development by outside users. SAMRAI is interoperable with plugin solver packages such as SPOOLES, SuperLU, PETSc, SUNDIALS. Performance and tuning is provided via vampir and tau and IO with silo and hdf5. The only package required to build SAMRAI is hdf5, which makes SAMRAI also easy to build and use without problems. Visualization is a simple task with the built in Visit output support.

The combined program of cFluid, SAMRAI and FronTier is called SAM_FT. SAM_FT lives in its own directory, outside SAMRAI and FronTier. As a separate program, it requires FronTier Lite, SAMRAI, and cFluid to be built already. In overview of SAM_FT, the SAMRAI library provides all code related to adaptive mesh refinement and storage of interior state information. FronTier provides front propagation and front query operations, and cFluid provides a front tracking aware solver to update interior state solutions. See Figure 4.2 for a diagram of software dependencies and what they provide.

4.3 FronTier and cFluid

4.3.1 FronTier Modifications

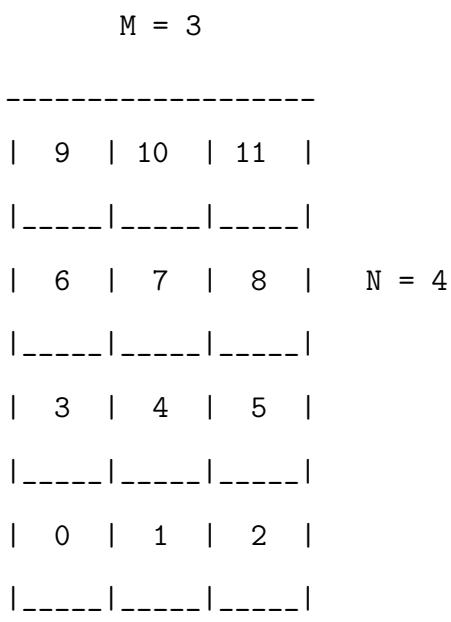
As mentioned in Chapter 3, in the Xu AMR attempt code was changed in many locations throughout `hyp/ driver/ tri/ front/` and `gas/` directories. Much of this code was written around assumptions about the Xu AMR model and was a duplication of code that other AMR packages do more efficiently (load balancing, data placement, coarsening and refining of data, etc). However, some tools needed for the new algorithm were already in place from Xu's work, mostly functions relating to AMR in the `front/` directory. Xu's changes to the `front/` directory that were part of the previous merge work were directly transferred into the new algorithm code. Most changes for the new AMR in the `front/` directory are Xu's. However some of the code is rearranged to keep AMR code separate, and clean.

FronTier's parallel model assumes that there is one front per processes. In Frontier AMR there are multiple fronts per processes. The single front assumption is coded into several FronTier functions involving patch scatters and redistribution. In between different steps in these functions a call to pp_min_status() occurs to ensure all fronts on all processors have done the correct thing. In FronTier AMR these functions are wrapped in a loop:

```
for(i=0; i < numPatches; i++)
{
status = redistribute(front[i]);
}
```

However other functions in AMR do require MPI. Special compile time macro USE_AMR is used to ensure that pp_min_status is not called in these locations.

In parallel FronTier there is an assumption that processors spatial decomposition follows the regular pattern:



In the boundary condition code an enumerator for processor subdomain was called SUBDOMAIN_BOUNDARY. Periodic boundaries have no enum so a statement such as:

```
  if(pp\_id\%M == 0 || pp\_id\%M == 0)
  {
treat as upper boundary
  }
```

was used to detect if a boundary was periodic, and then take appropriate steps to enforce periodic boundaries. In AMR the spatial decomposition is not regular so it is not possible to infer this information. To reconcile this a boundary type was added `AMR_SUBDOMAIN_BOUNDARY`.

Another change was to an existing function in `front/` called cut_interface(). This function is called to remove the part of the interface that falls along one side of a line. Depending on the type of interface and why the cut is made, special operations can be performed during the the cut. This is an intricate function and care was made to ensure changes made did not affect normal operation.

Three external functions and their sub functions developed by Xu for AMR compatibility were reorganized so that all related code exists in one file famrextra.c:

- clip_patch_front() - a wrapper for cut_interface to ensure front is cut properly during interface communication and regrid steps.
- ng_form_patch_subintfc_2d() - this functions is responsible for reforming the boundary curves after a front has been clipped. Information about the type of interface, the boundary, and buffer types is taken in, and the boundary curves are reinstalled properly.
- setupPatchFront - after a front is clipped and reformed, ensure data structures associated with front such as RECT_GRID and Patch_bdry_flag properly reflect the new front patch configuration.

Xu amr code was removed from many files such as `fadv.c fredist.c fredist2d.c fscat2d.c fsub.c`. Several files were completely removed:

```
front/:  famr\_adv.c  famrscat2d.c
driver/: doverturepatch.c doverturepatch2.c damr.h
 damrpatch2d.c damrpatch3d.c dpatchmesh.c
gas/:    goverinterp.c goverinterp2d.c goverinterp3d.c
hyp/:    hoverture_driver.c hamrscatter.c
tri/:    overture_trigrid1.c
```

4.3.2 G_CARTESIAN cFluid Modifications

Care was taken to ensure that the structure of the cFluid code was not changed. The main data structure for cFluid is the G_CARTESIAN class. Three new data members are added to G_CARTESIAN: `int level, max_level, and bool amr`. These new G_CARTESIAN members serve as a way to make small edits to member functions without interfering with functionality. Without these members, important G_CARTESIAN member functions would need to be copied, renamed and modified for only a small unintrusive change. Searching the G_CARTESIAN code, for these variables will find the only changes made to it.

Member functions that required large edits were made `virtual`, then a new class was made to inherit the G_CARTESIAN class. This class is called G_CARTESIAN_AMR. Listing 4.2 shows the important features of the G_CARTESIAN_AMR header file.

Since AMR libraries are in charge of ensuring patch buffers are up to date cFluid parallel functions to sync buffers were not needed. These functions were also made virtual so they could be set to empty functions: `scatMeshArray()`

`scatMeshFlux()` `scatMeshVst()` `scatMeshStates()` `scatMeshGhost()` .

For every patch operation which requires cFluid code, the state data from SAMRAI is copied into the cFluid state storage area. This reduces indexing differences between SAMRAI and cFluid, and is an important design decision. To prevent memory thrashing, the state storage memory pointer is allocated once at startup and is allocated to the size of the maximum patch size. To facilitate these initial allocations two new functions were added to G_CARTESIAN `ft_dim_size()`, and `ft_vec_size()`. These functions return integers that indicate the number of data points to allocate for the given grid size. In a single grid simulation the G_CARTESIAN version of these functions return the grid size of the simulation. In an AMR simulation the G_CARTESIAN_AMR version of these functions return the maximum patch size.

When SAMRAI calls cFluid code, G_CARTESIAN member functions assume that the grid and domain variables are already set properly. A new member function called G_CARTESIAN_AMR::resize_interior_and_front() does this. It sets patch related variables such as patch size in each direction (top_gmax), domain border coordinates (top_L and top_U), buffer sizes (lbuf, ubuf), and loop begin and end indices (imin imax). This function also sets the proper front for each patch. Only after this call can SAMRAI state data be copied into the G_CARTESIAN storage and other G_CARTESIAN calls can be made.

The most important numerical change to cFluid is the addition of a wrapper function for the ghost fluid code. When addFluxInDirection2dTracked calls appendGhostBuffer() its actually calling a wrapper. The pseudo code for this function is in Listing 4.3. On a fine patch the G_CARTESIAN version of appendGhostBuffer code is called as normal, but on coarse patches, the solver

has component information but no front. In addFluxInDirection2dTracked, the decision to call appendGhostBuffer happens at either a domain boundary, or a cell where the next cell is of a different component. On a coarse patch this code fills in the correct information at the domain boundary, and at a component change, it fills in buffer values with a flow through condition. The choice of using flow through is arbitrary, and could be anything. As long as the area around the interface is fully refined to within the length of the flux solver's stencil size on any level, the data is replaced with real ghost fluid method calculated fluxes after the next fine level update. This data replacement always occurs before the next coarse flux calculation occurs, due to the recursive nature of the Berger Oliger integration scheme.

4.3.3 FTPatches

The Xu front tracking algorithm and parallel scheme did have high level functions that were reusable for the new algorithm, so a new framework was created for tying front tracking and patch based AMR together. FTPatches.cpp and FTPatches.h were written to make collections of fronts on patches do common front tracking tasks. The AMR front tracking algorithm described in Section 4.1 requires front buffer updates between propagates and redistributes and regridding of the front when a new fine patch layer is made. For these high level capabilities low level requirements include multiple front storage per process, box overlap calculations, front clipping, and clip communication between patches.

There is a small amount of box logic code that may duplicate AMR box

algorithms but the code is simple, and avoids too much dependency on an external library. This code exists in the boxFT class. boxFT is then used throughout the FTPatches.cpp code. If future changes occur such as a different index or coordinate system, it is much easier to simply re-implement CoordsOfBoxContainingCells and overlapingCells in this small subsection of boxFt rather then changing internals of other sections of FTPatches.

Built on top of boxFT is several other classes. patchFT is a class with member data describing a patch, its buffer regions, and the front that belongs to it. patchFT also has temporary storage areas for interface clips to be sent and received during communication sections. patchLevelFT is the master class which stores all patchFT objects and their fronts for one process. It has member functions to control operations on all patches including regridding, redistribution, scattering clips, and printing and restart for all patch fronts. See Listing 4.5 for FTPatches.h and associated documentation.

The most complicated part of maintaining a collection of patches is exchanging front information to update patch buffers. The FTPatches code has been written to be as modular as possible, and simple to understand and read. In order to maintain a patch on a buffer correctly after every front modification, the buffer region of a front at the patch boundary must be gotten from the front that is in the interior of the neighboring patch, near the patch boundary. This process can be seen in Figures 4.3 and 4.4. The order in which this happens is as follows:

- fillPatchClipsForScatter() - For each patch, patchLevelFT calculates which other patches it overlaps with, then fills `patchFT.send` and

`patchFT.receive` for each patch.

- clipPatches() - Prepares clips of each patch for communication. For each scheduled clip in `patchFT.send` the interface is copied, clipped and associated.

- exchangeClips() - The communication of clips happens. All patches in send are transmitted to the receive of the patch they are destined for.

- clipPatchesToInterior() - Removes old front geometry in the patch buffer so the new clips can be stitched in.

- mergeClips() - Assembles clips on the new patch. Several steps occur. First the new clip geometry is put into the patch front, then the clip geometry is merged with the patch front, and finally new boundary line segments are installed.

- deleteClips() - The old `patchFT.send` and `patchFT.receive` clips are removed from each `patchFT`.

The other major operation is regridding the fine patch level. This consists of reassembling the interface in the new fine patch structure. The current implementation is for single processor only. The order in which this happens is as follows:

- makeOneBigFront() - Glues all patches in the patchList together.

- cutPatchesFromBigFront() - Cuts out an interface per patch.

4.4 The SAMRAI Code

4.4.1 The main() Function

The code shown in Listing 4.4 is an abbreviated version of the main driver provided by the Euler example code that comes with SAMRAI. This is only a slightly abbreviated version of the actual code with a only few lines missing for MPI initialization and IO setup. No major modifications exist to the Euler driver code, instead modifications occur to the Euler class. Observing the initialization of these classes is a good way to understand SAMRAI inheritance. After all objects are initialized the end product is a simple loop with a call to advanceHierarchy(dt_now).

For time dependent hyperbolic calculations SAMRAI provides an overarching class called the TimeRefinementIntegrator. Into this class is added all objects of the calculation. All objects placed into the TimeRefinementIntegrator are orchastrated by it during the course of a calculation. Figure 4.1 shows the TimeRefinementIntegrator and all the sub classes it uses to perform a hyperbolic simulation. Table 4.1 describes all classes required by the TimeRefinementIntegrator.

4.4.2 Euler Modifications

The Euler class and accompanying main loop code is an example provided by the SAMRAI packages that demonstrates how a code should use the SAMRAI library for a generic hyperbolic simulation. It shows how to write a concrete class to inherit abstract SAMRAI classes in order to fill in all the off the shelf

solver aspects of a cartesian structured AMR simulation. This concrete class implements the inherited virtual members to initialize and solve the generic hyperbolic equations on a single cartesian patch. An aspect that must be implemented by the user is the coarsening and refining of data between grid levels.

The Euler class is a concrete implementation of several abstract SAMRAI classes. The inheritance model allows for the Euler class to accumulate these virtual members in one class. All functions that require user implementation in the class hierarchy tree are concrete members in the Euler class. Functions required for gridding and integration have abstract members that are implemented in the Euler class. The Euler class implements the solvers, and perform data interpolation between levels for the Euler equations. Solving on a single patch involves computing fluxes, using those fluxes to compute the solution for the next time step on that patch, and choosing a dt for that patch.

Most of the changes made to the original Euler code relating to state/interior data were changed in the examples/Euler.C and Euler.h files.

The initialization code also operates in a similar manner to the Berger Oliger algorithm by supplying a virtual member function which the user implements to initialize one patch at a time. After all patches for each level are initialized, the refinement decision code is swept over that level, the next level is created, and the process starts again.

The Euler class was used as a skeleton code in order to install FronTier cFluid solvers and FronTier front tracking calls. The Euler class and where it fits in the SAMRAI class hierarchy can be seen in Figure 4.1. The SAMRAI/examples/Euler directory was copied to a new directory named SAM_FT.

Then, the single patch operations were removed and replaced with FronTier cFluid code. The following single patch operations were completely replaced:

- Euler::initializeDataOnPatch - replaced with `G_CARTESIAN.setInitialStates` the cFluid initialization code for several multimaterial problems
- Euler::computeStableDtOnPatch - replaced with code to take into account front speed
- Euler::computeFluxesOnPatch - replaced with several cFluid calls, and the major source area for the front tracking modified Berger Oliger algorithm. Fluxes are calculated and summed here, and stored for flux summing in the reflux step.
- Euler::conservativeDifferenceOnPatch - work is done here only at the reflux step. Existing Euler code and SAMRAI, calculates proper flux summing, from fluxes given from computeFluxesOnPatch. Fluxes are summed only when the reflux flag is true.

For each of the above mentioned Euler single patch operations, the SAM_FT code copies the necessary state data into the cFluid storage area, along with all grid parameters (via a call to `G_CARTESIAN_AMR::resize_interior_and_front()`). cFluid can then operate on the patch as it would with a normal FronTier/cFluid serial simulation. See 4.3.2 for more details.

The Euler class also was modified to hold a variable for the component information, stored as a double precision variable. By doing so, we can take advantage of SAMRAI's automatic refinement and coarsening code. A few

added lines to the Euler constructor, and one new data member add the `double ftcomp` variable. Since the stored interface is on the finest level, we can update the component values after a propagation, or a redistribute, and the rest of the coarse patches will automatically get the correct component values through the built in coarsening algorithms. The computeFluxesOnPatch code updates the component values on the finest level patches after the front propagation. To ensure refinement always occurs at the interface, we add code to tagGradientDetectorCells to refine when a component change occurs.

4.4.3 HyperbolicLevelIntegrator Modifications

The parent class of the Euler class is the `HyperbolicLevelIntegrator` class (HLI) see Figure 4.1. This class implements the Berger and Oliger hyperbolic AMR recursive integration algorithm[4] [19]. The HLI class is included in the SAMRAI library, rather than the Euler example code, because no modifications are made to it, when incorporating a users single grid patch solver into SAMRAI. It was written to be generally useful for all coupled hyperbolic problems. The SAMRAI Euler class, and corresponding HLI class was designed as a system where the user implements only the code needed to solve for regular cartesian numerical floating point data on a single cartesian patch.

The `patchData` class is designed to be inherited by other classes for the purpose of representing data on a patch. It contains virtual members to be implemented with code relevant to the specific data type being represented. This allows SAMRAI users to add data per patch with a few lines of code.

The HLI class is written to manipulate collections of `patchData` objects.

Including routines to update patches, and collect time step size for patch levels. HLI also interpolates data between level advances. Inside the HLI class is a advanced collection of MPI and scheduling code that ensures `patchData` objects are operated on in an efficient manner. This communication scheduling does not take into account the interface, or its associated communication requirements. Interface buffers need to be updated after every single patch operation, such as advance and redistribute.

As mentioned in Section 4.3.3, FTpatches was written to allow the flexibility required to easily insert code to operate on collections of interfaces. The FTPatches framework was written to be reusable with other AMR libraries. To operate on interface in the correct order in it was necessary to directly modify the HLI code to add FTPatches calls. This allows us to modify the Berger Oliger Algorithm with Front tracking sepefic calls on the finest level patch. 4.1. The two most important changes to the HLI class are in the following member functions:

- initializeLevelData - on the finest level, add each new patch to `FTPatches_new.addPatchToList()` and then `FTPatchLevel_new.Regrid(FTPatchLevel_old,...);` then destroy FTPatchLevel_old and swap pointers.

- advanceLevel - on the finest level before the next state update exchange interface clips as described in 4.3.3

4.5 Code Metrics

As an analog to chapter 3.2.1 we will inspect the code metrics of the cFluid-AMR code. Version control tracking the changes to FronTier's source reports 36 files changed, with 4990 insertions and 1164 deletions. In addition to the FronTier changes we must also quantify our changes to the Euler example code, as discussed in section 4.4.2. Early development was untracked but since check in 38 files were changed, with 4652 insertions, and 1138 deletions.

Using SLOCCount[25] gives FronTier 499,834 SLOC before merging the cFluid-AMR changes and 502,561 SLOC after, an addition of 2727. The Euler SLOC before changes is 9,063 and 9,195 after, an addition of 132 SLOC.

With the cFluid-AMR changes applied, and all the code associated with the `Gas`-AMR work, that is not used by cFluid-AMR removed, the FronTier code measures 486,798 SLOC a difference of -15763 SLOC. Adding this 15763 to the 26,068 of the changes to make mentioned in 3.2.1 gives us a total of 41,832 SLOC for the `Gas`-AMR functionality.

4.6 Notes on Extending to 3D

Much of the changes mentioned are written in a dimensionally independent way. SAMRAI operates with almost no difference in programing in two or three dimensions.

4.6.1 FTPatches and famextra Changes

The variables TMPDIM and N_PATCH_BUFS are used to do all box calculations in FTPatches in a dimensionally independent way. The only code that needs inspection and replacement are FronTier interface geometry functions that are two dimensional specific. The challenging changes needed will occur in FTPatches.cpp or famrextra.c .

Dimensionally dependent code exists in the following functions: getBufferICoords(), cutPatchesFromBigFront(), makeOneBigFront() clipPatchesToInterior(), clipPatches(), printVTKPatches(). The work in FTPatches.cpp functions listed requires that FronTier 2D calls to be replaced with 3D equivalents for the following functions: cut_interface(), clip_patch_front(), ng_form_patch_subintfc_2d(), remove_patch_all_boundary_curves(), delete_subdomain_curves().

4.6.2 SAM_FT and G_CARTESIAN_AMR Changes

Simply reading through the code in cFcartsnAMR.cpp and Euler.C will show where three dimensional cases need to be filled in. These files deal mainly with solving interior state values so the majority of the work is looping over the state values. In these files the most prevalent change will be a two or three dimension conditional statement where there is a cleanup with a comment on what needs to be changed in three dimensions. In most cases, its a simple loop index modification.

4.7 Installing the Code

SAM_FT depends on SAMRAI and Frontier. Frontier can be compiled with no extra dependencies. SAMRAI however, depends on HDF5. HDF5 and SAMRAI are libraries that are simply downloaded, untared, configured and compiled. SAM_FT links with libFrontier libcfluid and several libsamrai* libraries.

```
To do a majority of the compiling simply run:

untar sam_ft_XXXX_XX_XX.tar.gz
cd sam_ft/
./install

This script will untar, configure and compile
hdf frontier and SAMRAI once these are all build
you can type:

cd samft/
make main2d

to build a 2d binary or

make main3d

to build a 3d binary.
to retar all this code simply commit your changes
in Frontier/ and sam_ft and this dir. Then type:

./package

there are 2 input files to run this code. The main
input file is a SAMRAI style input file. This file
contains all variables related to amr, such as
patch size, patch efficiency, refinement ratio,
load balancing, dt decisions, and boundary conditions.
this file also contains a variable called
```

```
frontier_input and frontier_output.

frontier_input -- is the cFluid file. This contains
                  all solver info, front tracking
                  info and initialization info.

to run the code type:
./main2d sample_input/rm2d.input

make sure that the file sample_input/rm2d.input
has the correct absolute path to the frontier_input file.
```

```
Recursive_Integrate(level_i)
{
        repeat hj/hc times
        if(regridding_time)
                calculate_errors_for level_i and finer
        step dt on all grids at level_i
        if level_i+1 exists
                Recursive_Integrate(level_i+1)
                update(level_i, level_i+1)

        if level_i == finest_level
                propagate interface
                update level_i component field
        else
                create empty front and copy in component info from
                    SAMRAI.
}
```

Listing 4.1: The FronTier Modified Berger Oliger Algorithm

```
class G_CARTESIAN_AMR: public G_CARTESIAN {
// NEW MEMBERS
        //initialization memeber functions
        void resize_interior_and_front(int iminn[],int imaxx[], int ighost[],
                                       double top_LL[], double top_UU[],
                                       int ftlevel, int samid, int center_comp);
        void frontier_init(const char*, const char*, int, bool);
        void setupCellCenters(void);

        void appendGhostBuffer( SWEEP *, SWEEP *, int, int, int*, int, int);

// NEW DATA
        // front level data
        patchLevelFT  *AmrLevelInfo;
        patchLevelFT  *AmrLevelInfo_tmp;

        // new flux storage
        SWEEP *st_fieldamr,st_tmpamr;
        FSWEEP *st_fluxamr;
        double **a,*b;
        void solveFlux(double dt);
        void solveRKFLux(int order,double dt);
        void sumFlux( FSWEEP a_flux, FSWEEP b_flux, double chi);
};
```

Listing 4.2: G_CARTESIAN_AMR

```
void G_CARTESIAN_AMR::appendGhostBuffer(
        SWEEP *vst,
        SWEEP *m_vst,
        int n,
        int nrad,
        int *icoords,
        int idir,
        int nb)
{

        // if on the finest level, we call the regular appendGhostBuffer code
        // then exit this function.
        if(level == max_level && eqn_params->tracked)
                return  G_CARTESIAN::appendGhostBuffer(vst, m_vst, n, nrad,
                                                       icoords, idir, nb);


        // otherwise we are on a coarse patch and we call our the wrapper code below.

        INTERFACE       *intfc = front->interf;
        int             i,j,k,index;
        HYPER_SURF      *hs = NULL; // this should be changed
        COMPONENT       comp;
        double          crx_coords[MAXD],coords[MAXD];
        STATE           *state,st,ghost_st;
        int             ind2[2][2] = {{0,1},{1,0}};
        int             ind3[3][3] = {{0,1,2},{1,2,0},{2,0,1}};
        int             vec_idx,lastorfirst;

        index = d_index_ft(icoords,top_gmax,dim);
        comp = cell_center[index].comp;

        bool ismin = (icoords[idir] == imin[idir]) && (nb == 0);
        bool ismax = (icoords[idir] == imax[idir]) && (nb == 1);
        if (ismin == true ||  ismax == true )
        {

                // if here, we are at a patch boundary, and we call
                // boundary fill in functions.

                switch (rect_boundary_type(intfc,idir,nb))
                {
                        case SUBDOMAIN_BOUNDARY:
                        case AMR_SUBDOMAIN_BOUNDARY:
                          if(ismin == true  )
                                setFromBufferStates(vst,m_vst,hs,icoords,idir,nb,0,1,comp)
                                    ;
                          else
                                setFromBufferStates(vst,m_vst,hs,icoords,idir,nb,n,1,comp)
                                    ;
                         break;
                         break;
                    case NEUMANN_BOUNDARY:
                          if(ismin == true  )
                                setNeumannStates(vst,m_vst,hs,icoords,idir,nb,0,1,comp);
                          else
                                setNeumannStates(vst,m_vst,hs,icoords,idir,nb,n,1,comp);
                         break;
                     case DIRICHLET_BOUNDARY:
                          if(ismin == true )
                                setDirichletStates(vst,m_vst,hs,icoords,idir,nb,0,1);
                          else
                                setDirichletStates(vst,m_vst,hs,icoords,idir,nb,n+nrad,0);
                         break;
                }
         }
         else
         {
                // for NOT boundary, we assume that this data will be over
                // written by finer level patches, so this could be anything
                // that is relativly inert.
                //
                // what we fill these buffers with is a crude flow through,
                // that simply takes the last valid state on the side
                // we are sweeping and duplicates it to all stencil points
                // that are past the non existant interface (because were are on a
                // coarse grid)
                //
                if(nb == 0)
                        lastorfirst = nrad;
                else
```

```
                        lastorfirst = n+nrad - 1;

                ghost_st.dens = vst->dens[lastorfirst];
                ghost_st.engy = vst->engy[lastorfirst];
                ghost_st.pres = vst->pres[lastorfirst];

                if(dim == 2)
                    for(j=0; j<2; j++)
                        ghost_st.momn[ind2[idir][j]]= vst->momn[j][lastorfirst];
                else if(dim == 3)
                    for(j=0; j<3; j++)
                        ghost_st.momn[ind3[idir][j]]= vst->momn[j][lastorfirst];
                for (i = 0; i < nrad; ++i)
                {
                        if(nb == 0)
                            vec_idx = nrad-i;
                        else
                            vec_idx = n+nrad+i;

                        vst->dens[vec_idx] = ghost_st.dens;
                        vst->engy[vec_idx] = ghost_st.engy;
                        vst->pres[vec_idx] = ghost_st.pres;

                        for(j=0; j<3; j++)
                            vst->momn[j][vec_idx] = 0.0;
                        if(dim == 2)
                            for(j=0; j<2; j++)
                                vst->momn[j][vec_idx] =
                                    ghost_st.momn[ind2[idir][j]];
                        else if(dim == 3)
                            for(j=0; j<3; j++)
                                vst->momn[j][vec_idx] =
                                    ghost_st.momn[ind3[idir][j]];
                }
        }
}
/* end G_CARTESIAN_AMR::appendGhostBuffer */
```

Listing 4.3: appendGhostBuffer wrapper code.

```
// Package:     SAMRAI application
// Copyright:   (c) 1997-2008 Lawrence Livermore National Security, LLC
// Description: Main program for SAMRAI Euler gas dynamics sample application

int main( int argc, char *argv[] )
{
   // Create major algorithm and data objects which comprise application.
   geom::CartesianGridGeometry grid_geometry = new geom::CartesianGridGeometry(...));

   hier::PatchHierarchy patch_hierarchy = new hier::PatchHierarchy(..., grid_geometry);

   Euler* euler_model = new Euler(... , grid_geometry);

   algs::FronTierHyperbolicLevelIntegrator hyp_level_integrator =
      new algs::FronTierHyperbolicLevelIntegrator(..., euler_model, true,
          use_refined_timestepping);

   mesh::StandardTagAndInitialize error_detector =
      new mesh::StandardTagAndInitialize(..., hyp_level_integrator));

   mesh::BergerRigoutsos box_generator = new mesh::BergerRigoutsos();

   mesh::LoadBalancer load_balancer =
      new mesh::LoadBalancer(...));

   mesh::GriddingAlgorithm gridding_algorithm =
      new mesh::GriddingAlgorithm(..., error_detector, box_generator, load_balancer);

   algs::TimeRefinementIntegrator time_integrator =
      new algs::TimeRefinementIntegrator(..., patch_hierarchy, hyp_level_integrator,
          gridding_algorithm);

   // Initialize hierarchy configuration and data on all patches.
   double dt_now = time_integrator->initializeHierarchy();

   // Time step loop.  Note that the step count and integration
   // time are maintained by algs::TimeRefinementIntegrator
   double loop_time = time_integrator->getIntegratorTime();
   double loop_time_end = time_integrator->getEndTime();

   while ( (loop_time < loop_time_end) &&
          time_integrator->stepsRemaining() ) {
                int iteration_num = time_integrator->getIntegratorStep() + 1;
                double dt_new = time_integrator->advanceHierarchy(dt_now);
                loop_time += dt_now;
                dt_now = dt_new;
        }
```

Listing 4.4: An abreviated SAMRAI Euler main driver code

```
enum SHIFT{ up, down, none };
class boxFT
{
        public:
        int iL[3]; //lower cell centered index of patch
        int iU[3]; //upper cell centered index of patch
        double gL[3]; // global lower domain coordinate
        double dx[3]; // cell size
        void CoordsOfBoxContainingCells(double L[3], double U[3]); // give the upper and
             lower coordinates of the box from the stored index.
        boolean overlapingCells(boxFT *boxOther, boxFT *boxOverlap); // return true and
             set boxOverlap if boxOther overlaps with this boxFT.
};

// ClipOfThisPatch contains information about a clip to send.
// Collections of these objects are put in a container inside
// of patchFT. This collection of clips is taken of that patchFT.
class ClipOfThisPatchFT: public boxFT
{
        public:
        int patchImGoingTo; //!< in the patchList
        INTERFACE *intfc; //!< the cliped interface
        boolean saveInterior[2][TMPDIM]; //!< instructions on how
                                         //!< to clip the patch
        boolean setCutNoneLocal[2][TMPDIM];
        enum SHIFT shift[TMPDIM]; //!< determines if the patches
                                  //!< are shifted for periodic BCs.
        int uid; //!< for comunications purposes.
};

// ClipOfOtherPatchFT contains information about a clip to receive.
// Collections of these objects are put in a container inside of
// patchFT. This collection of clips is given to that patchFT,
// from other patches.
class ClipOfOtherPatchFT: public boxFT
{
        public:
        int patchICameFrom; //!< in the patchList
        INTERFACE *intfc; //!< the received clip from the other patch
        int uid;//!< for comunications purposes.
};

// patchFT contains all info about a patch. It also contains info on
// clips of this patch, that are to be cliped and sent, and clips of
// other patches, which are received from other patches, to be
// attached here. This class also contains memeber functions to query
// info about intersections with other patches.
class patchFT
{
        public:
   patchFT(int iLL[3], int iUU[3],  int lbuff[3],
           double dxx[3],
           double LL[3], Front *frontt, int samid);  //!< constructor

   ~patchFT();
   void print();
   boolean bufferIntersection(patchFT *p,
                              int bufferNum,
                              boxFT *boxOverlap);//!< query if this patch intesects
                                                 //!< another patchFT *p, on a given
                                                 //!< bufferNum.

   boolean periodicBufferIntersection( patchFT *p,
                                       int bufferNum,
                                       boxFT *domain,
                                       boxFT *boxtmp,
                                       enum SHIFT ClipShift[2]); //!< query if this patch
                                          //!< intesects another patchFT *p, on the other
                                          //!< side of a periodic Domain, on a given
                                          //!< bufferNum.

   boolean getBufferICoords(int buffer, boxFT *boxBuff);
   Front *front; //!< the front for this patch
   list< ClipOfThisPatchFT > send; //!< clips of this patch to be sent.
   list< ClipOfOtherPatchFT > receive; //!< clips of other patches to be received.
   boxFT box; //!< the geometry of this patch.
   int buf[3];//!< buffer info
   int samidx;
};

/// This class contains all data, and to methods operate on a collection of
/// interfaces on patches, on a single level in an AMR simulation.
```

```
class patchLevelFT
{
   public:
   patchLevelFT();
   void addPatchToList(int iLL[3], int iUU[3],  int lbuf[3],
                       double dx[3],
                       double gL[3], int samidx); //!< addPatchToList is used to
                                                  //!< add patches to the patchList
                                                  //!< vector, which serves as the
                                                  //!< container of all patch info on
                                                  //!< the current level.
   vector< patchFT > patchList;
   vector< string > restartList;
   vector< Front* > fronts;
   vector<int> i_comp;
   G_CARTESIAN_AMR *g_cartesian;

   boxFT Domain; //!< the entire computational domain boxFT.

   //major functions
   void Regrid(patchLevelFT*,bool); //!< removes interface from the buffers of all patches
       .
   void redistributePatches(); //!< redistributes

   // tools for used for buffer clip exchange
   void clipPatchesToInterior(); //!< removes interface from the buffers of all patches.
   void fillPatchClipsClassForScatter(); //!< fills the ClipOfThisPatchFT and
                                         //!<  ClipOfOtherPatchFT of each patchFT in
                                           patchList
   void clipPatches(); //!< after fllPatchClippsClassForScatter is called, the patches
                       //!< interface on that patch is copied and cliped for each clip in
                          the
                       //!< send vector.
   void exchangeClips(); //!< after clipPatches is called, the clips can be sent from the
                         //!< patches that they originated from, to the patches they are
                            going to.
   void mergeClips(); //!< after the exchangeClips happens, the clipsOfOtherPatches is
       filled,
                      //!< and the clips from the other patches can be merged.
   void deleteClips(); //!< after the merge takes place, all leftover interface peices in
       the
                   //!< send and receive vectors are deleted.

   // tools used for Regrid
   void makeOneBigFront(); //!< glues all patches in the patchList together
   void cutPatchesFromBigFront();

   // print  and restart functions
   void printVTKPatches(string FTout);
   void printRestart(string FTout);
   void readRestart( Front *main_front);
   void printPatchClipsClassForScatter();
   void printPatchList();
   void printSinglePatch(string FTout,INTERFACE *intfc);
   void simplePrint(char *name);

   // misc functions
   void setStep(int step);
   void setRestart(bool restar){restart = restar;}
   bool getRestart(){return restart;}
   private:
        bool restart;
};
```

Listing 4.5: FTPatches.h Header File

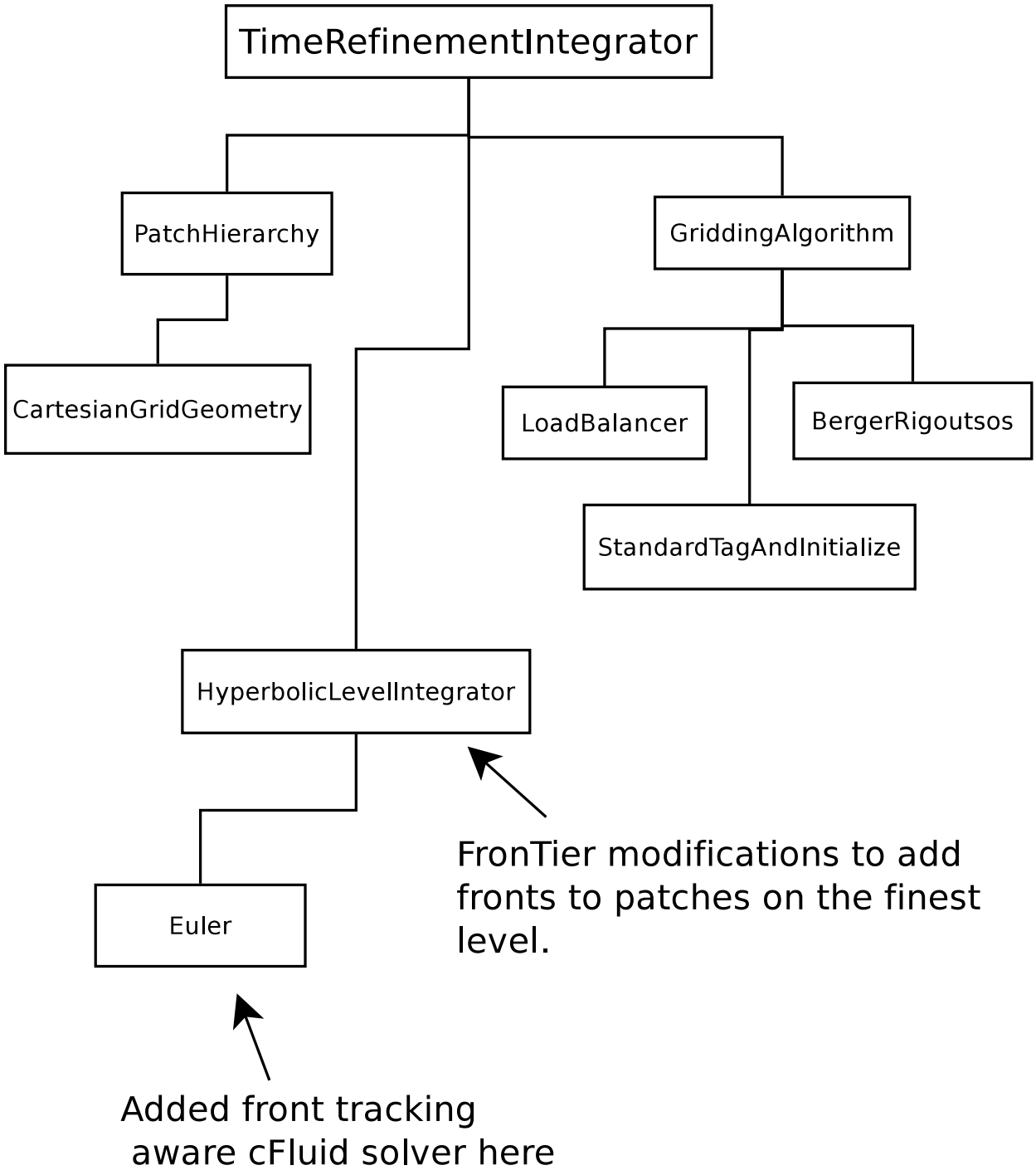


Figure 4.1: Graph of SAMRAI class nesting and inheritance, with notes on where changes were made.

PatchHierarchy	A container for the AMR patch hierarchy and the data on the grid.
CartesianGridGeometry	Defines and maintains the Cartesian coordinate system on the grid. The PatchHierarchy maintains a reference to this object.
TimeRefinementIntegrator	Coordinates time integration and adaptive griding procedures for the various levels in the AMR patch hierarchy. Local time refinement is employed during hierarchy integration; i.e., finer levels are advanced using smaller time increments than coarser level. Thus, this object also invokes data synchronization procedures which couple the solution on different patch hierarchy levels. The time refinement integrator is not specific to the numerical methods used and the problem being solved. It maintains references to two other finer grain algorithmic objects, more specific to the problem at hand, with which it is configured when they are passed into its constructor.
HyperbolicLevelIntegrator	Defines data management procedures for level integration, data synchronization between levels, and tagging cells for refinement. These operations are tailored to explicit time integration algorithms used for hyperbolic systems of conservation laws, such as the Euler equations. This integrator manages data for numerical routines that treat individual patches in the AMR patch hierarchy. In this particular application, it maintains a pointer to the Euler object that defines variables and provides numerical routines for the Euler model.
Euler	Defines variables and numerical routines for the discrete Euler equations on each patch in the AMR hierarchy.
GriddingAlgorithm	Drives the AMR patch hierarchy generation and regriding procedures. This object maintains references to three other algorithmic objects with which it is configured when they are passed into its constructor.
BergerRigoutsos	Clusters cells tagged for refinement on a patch level into a collection of logically-rectangular box domains.
LoadBalancer	Processes the boxes generated by the BergerRigoutsos algorithm into a configuration from which patches are constructed. The algorithm we use in this class assumes a spatially-uniform workload distribution; thus, it attempts to produce a collection of boxes each of which contains the same number of cells. The load balancer also assigns patches to processors.
StandardTagAndInitialize	Couples the gridding algorithm to the HyperbolicIntegrator. Selects cells for refinement based on either Gradient detection, Richardson extrapolation, or pre-defined Refine box region. The object maintains a pointer to the HyperbolicLevelIntegrator, which is passed into its constructor, for this purpose.

Table 4.1: Descriptions of classes in the SAMRAI hierarchy, from SAMRAI manual

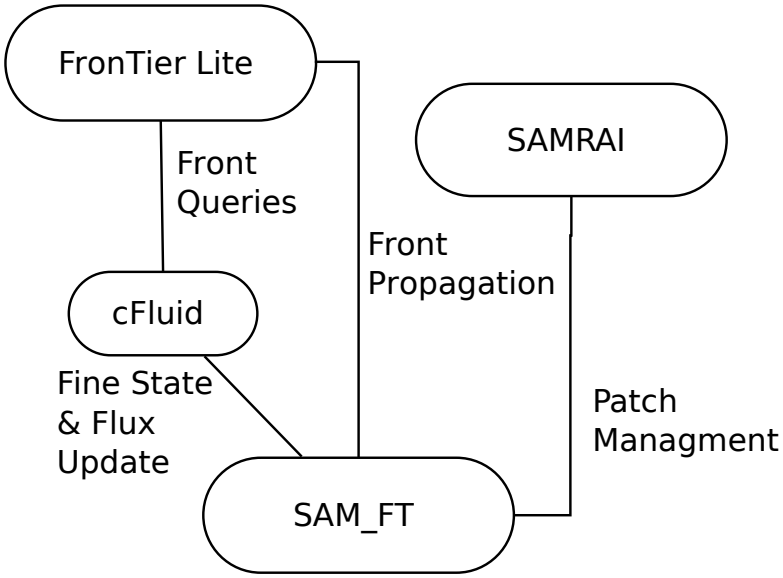


Figure 4.2: Showing software dependencies between FronTier cFluid and SAMRAI

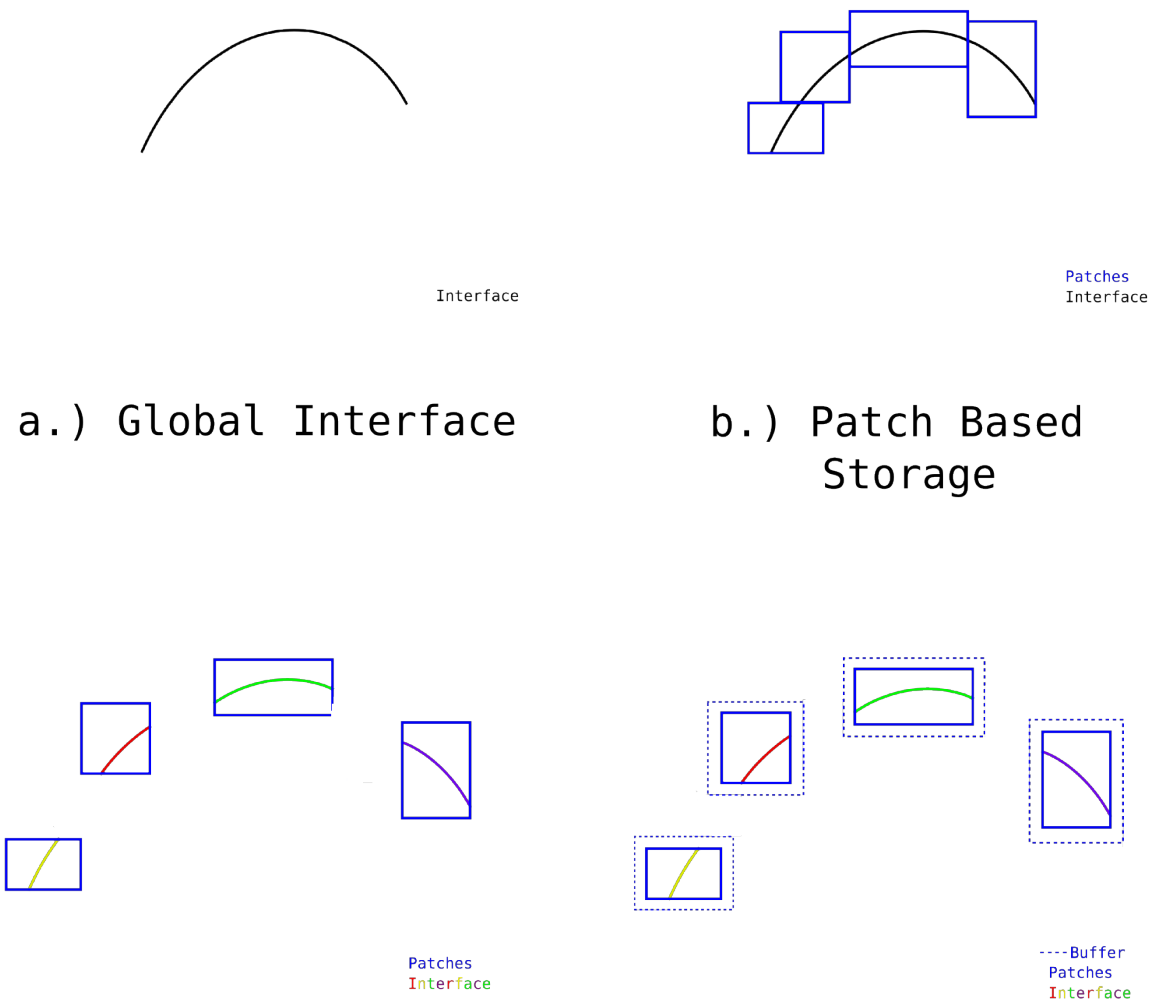
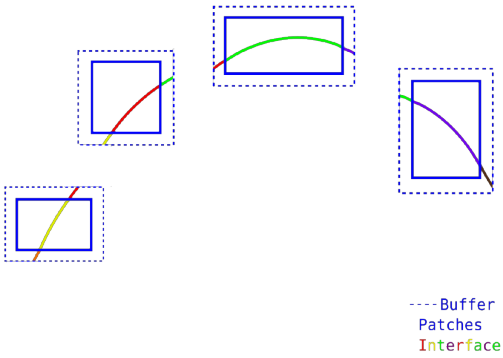
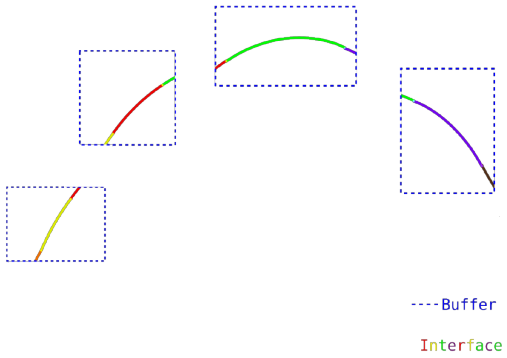


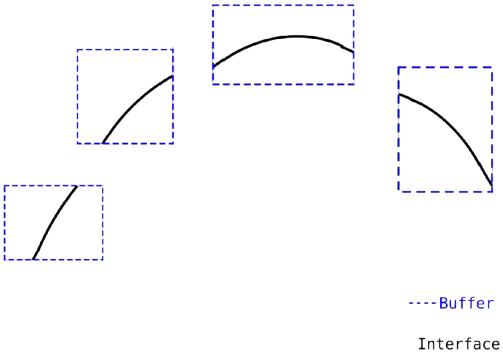
Figure 4.3: Patch Front update



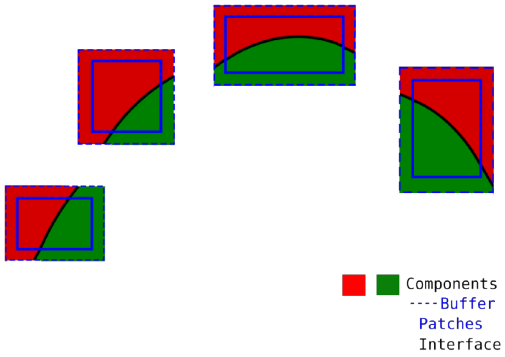
e.) clip exchange



f.) adding clip data



g.) clip merging



h.) component query

Figure 4.4: Patch Front update 2

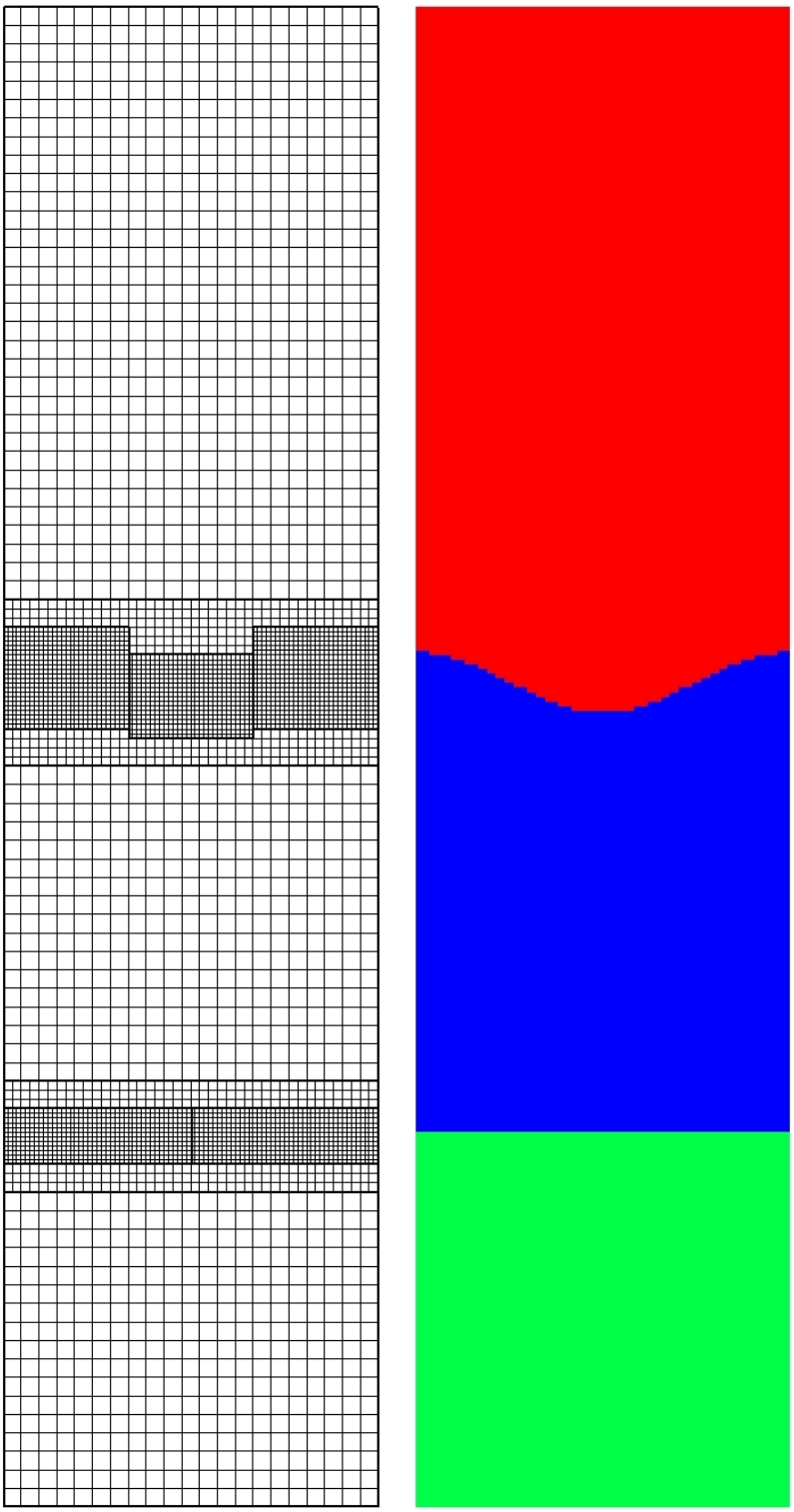


Figure 4.5: A patch based mesh (left) and the accompanying density profile (right)

Chapter 5

Results

We present several benchmark test problems to illustrate the effectiveness of the new combined cFluid-AMR algorithm. Using AMR to calculate problems involving a shock wave is very common because for such problems, changes are concentrated around the shock and contact interaction region. Shock represents a sharp and discontinuous change in a physical variable such as density, velocity or pressure. Initially the adaptive refinement area only covers a small region in the neighborhood of the discontinuity. As the simulation progresses, the shock moves through the entire domain and interacts with other discontinuities such as a contact surface. As a shock moves through the domain, the AMR patches in the simulation can track and refine it along with its reflections and refractions. FronTier can deal with multi-material or multi-component computation. The tracked moving interface is used as a boundary between two materials. In a combined cFluid-AMR simulation, the discontinuities are detected and tracked by both AMR patches and the interface. In our benchmark problems, we choose to simulate problems which need to

combine these two features. The Richtmyer-Meshkov [27] instability serves as a good example.

These benchmarks comparisons are carried out on the same computer with identical memory and CPU load. The test case RM2D-N has Neumann boundaries at the top and bottom of the computational domain. The simulations were performed on a single core of the Intel Xeon X5650 @ 2.67GH CPU. RM2D-D uses Dirichlet boundaries at the top and bottom, and the simulations were carried out on a single core of a Intel Xeon CPU L5430 @ 2.66GHz CPU. Care was taken to ensure that no computing resources were used by other programs, and I/O was turned off to eliminate errors in CPU counting due to system delay. The CFL factor in all cases was set to 0.75 for both single grid and AMR calculations. We also make sure we use the same software versions for all comparison simulations.

5.1 Input Choices

For both RM2D-N and RM2D-D, we use the density ratio 1:2 across the contact surface and the polytropic constant $\gamma = 1.667$. The pre-shock pressure is set to 1 for a Mach 2 shock. The computational domain is a 1×4 rectangular region.

5.1.1 Refinement Criteria

The SAMRAI input file contains two general types of AMR specific parameters: refinement criteria parameters and gridding algorithm parameters. The choice of these parameters significantly affect the efficiency of AMR, and there-

fore the comparison between an AMR simulation with a simulation using a single fully refined grid.

The refinement criteria section of the SAMRAI's input file allows the choice of user defined refinement algorithms. For each scalar variable managed by SAMRAI the user is given the option to choose different refinement criterion functions. In the case of Euler equations, these scalar variables include pressure and density. The built-in refinement choices for scalar variables are deviation, gradient and shock condition.

After a hyperbolic sweep, the selected functions are used to compute on the updated variables, and those regions with values exceeding the criteria will be refined. Among these functions, deviation is the difference between neighboring cells. The region is refined if it exceeds the cut-off value from the user input. The gradient function calculates the first order gradient across the cells and compares it to the the user input value. The shock detector tags cells where

$$|Value_{n+2} - Value_{n-2}| \times SHOCK\_ONSET < |Value_{n+1} - Value_{n-1}|$$

or

$$|Value_{n+2} - Value_{n-2}| \times |Value_{n+1} - Value_{n-1}|$$

and

$$Max(|Value_n - Value_{n-1}|, |Value_n - Value_{n+1}|) > SHOCK\_TOL$$

where SHOCK_ONSET and SHOCK_TOL are user supplied parameters.

In addition to these generic refinement options, a specific function is implemented to enforce refinement within a certain distance from the tracked

`front`. Since SAMRAI also maintains the grid component at all levels, we inserted a function to search for component changes in N cells around each cell, where N is a user supplied integer value. A discussion of the results of different AMR gridding and refinement parameters can be found in section 5.2.2.

5.1.2 Gridding Algorithm Parameters

While the refinement criteria function calculates and decides where to refine the mesh, the selection of gridding algorithm and its associated parameters determine how to refine. The SAMRAI-AMR library provides many user defined parameters for the size and placement of AMR patches over the tagged cells. These parameters have large effect on the performance gains when compared with a single fully refined grid simulations.

- max_levels - the number of patch levels.
- ratio_to_coarser - ratio of refinement to next coarser level.
- largest_patch_size - the largest size a patch can be.
- smallest_patch_size - the smallest size a patch can be.
- efficiency_tolerance - the minimum percentage of tagged cells a patch can contain.
- combine_efficiency - the efficiency parameter which determines when large patches can be split into smaller ones.

Unless otherwise noted, each simulation has a max_level of 3, largest_patch_size of 50, 100, or 200 per level, and a smallest_patch size of 12, 24, or 48 per level. Each simulation has an efficiency_tolerance of .65 and a combine_efficiency of .85.

5.2 Objectives of Test Cases

Simulations were carried out on Dirichlet and Neumman boundary conditions. The Dirichlet boundary condition case compares cFluid-AMR interoperation with other methods. The Neumman boundary condition case compares computational efficiency under different AMR parameters.

5.2.1 RM2D with Dirichlet Boundary

This set of simulations show the differences in solution and calculation time of different methods. We compared cFluid-AMR tracked, cFluid tracked with fully refined mesh, and cFluid untracked at regular and high resolutions. The CPU times are listed in Table 5.1. For the tracked AMR simulation the effective resolution, which is defined as the resolution at the finest level when applied to the entire domain is 200×800. cFluid tracked and cFluid untracked at regular resolutions are also 200×800. The high resolution untracked simulation was set by doubling the number of grid points in each dimension so that the numerical diffusion is not visually significant.

The numerical solutions are compared side-by-side at $t = 3$, 6, and 9, respectively (Figures 5.1, 5.2, 5.3). The untracked simulation with regular resolution shows a large difference from the others in that the interface is

very diffused. A much refined mesh is needed for the untracked simulation to show comparably sharp material interface, see the case with 800×3200 mesh. However, we can see from Table 5.1 that the computational time required for this level of refinement is very large which will make three dimensional simulation impossible.

The growth rate, resolution of the interface and vortex rolling of the bubble and spike in the tracked AMR and the tracked fully refined calculation are comparable. This shows that the quality of the solution in the AMR simulation is the same as the fully refined non-AMR solution, see 4.1. However, the AMR simulation takes 40% less CPU time compared to the non-AMR simulation.

5.2.2 RM2D with Neumann Boundary

This set of simulations tests AMR capabilities in a more demanding situation, where we are interested in refining the pressure gradient as well as the region near the contact surface. We make a comparison between AMR-tracking and fully refined tracking. The shock wave in this set of simulations is reflected by both top and bottom boundaries each time it passes through the material interface. Such reflection is repeated three times during the simulations. This set of simulations also generate complicated reflected rarefaction waves. The computational domain was changed slightly to 1.024 x 4.096 to make comparisons easier with different ratios of refinement. In all of these AMR simulations, we use three levels of refinement and the same mesh size at the finest level. However coarse grid size and refinement ratios per level vary.

Figure 5.4 presents the end time solution and Table 5.2 shows the corre-

sponding CPU times and other statistical data of the simulations.

The pressure gradient refinement criteria used in the simulations shown in Figure 5.4 C and D are shock_tolerance = 460 and onset = .95. These parameters were chosen to resolve the primary shock through most of the simulation but do not resolve reflected waves when the shock passes through the interface. The pressure gradient refinement criteria for simulation B is shock_tolerance = 30 and onset = .85. As we can see from Figure 5.4, simulation B captures almost the same degree of detail for the secondary waves as the fully refined calculation. However, this also makes their CPU time almost comparable to each other.

Simulations C and D differ only in their coarse mesh size, and the ratio of mesh refinement at the last level. Simulation C has 2,2 refinement ratio in two levels, and simulation D has 2,4 refinement ratio in two levels. There is a noticeable difference in performance due to the change of the last refinement ratio from 2 to 4. The major difference is in the number of fine level steps taken. This is shown in Table 5.2. The timeRefinementIntegrator's function makes this decision. The maximum velocity values on different grid levels can be slightly different, and the maximum velocity values on the finest level are slightly larger then on coarser levels. For a doubling of refinement the CFL dictates:

$U_{medium} \times \frac{\Delta t_{medium}}{\Delta x} = 2U_{fine} \times \frac{\Delta t_{fine}}{\Delta x}$

If $U_{fine} > U_{medium}$ then the finest level can not take 2 steps of size $1/2dt_{medium}$ to reach the sync time with the next coarser level. Instead the finest level must take 3 steps of size $1/3dt_{medium} + epsilon$ where $epsilon << dt_{medium}$. Simulation D has a finest cell size $1/4 \times dx_{medium}$ as opposed to $1/2 \times dx_{medium}$ in sim-

ulation C. This allows simulation D to take 5 steps of size $1/5dt_{medium}+epsilon$. Due to this phenomenon, to advance the the same amount of time D takes 5 steps and one sync when optimally it should take 4, as opposed to C which takes 6 steps and 2 syncs when it optimally should take 4. This leads to the biggest calculation time improvement over fine grid, of 43 percent between simulation D and the fully refined simulation A.

The number of cell operations per coarse time step is a a measure of computational intensity at that step in the calculation. The plot of this value during the course of a calculation is shown in Figure 5.5 for simulation D in Figure 5.4. The graph has rapid jumps that correspond to the gridding algorithm's decisions of where and how to place refined patches. A sample of this is illustrated by the jump indicated by the first two arrows in Figure 5.5 and the corresponding patch configuration change. An overall growth trend can be seen as the instability spreads, with more mixing toward the end of the simulation.

Simulation	Resolution	Time (s)
AMR Tracked	effective 200x800	6174
Tracked	200x800	10015
Un-Tracked	200x800	5885
Un-Tracked	800x3200	480416

Table 5.1: Timing results for RM2D Dirichlet

Image	A	B	C	D
Simulation Type	Fully Refined	AMR	AMR	AMR
Levels	1	3	3	3
Ratio	NA	2,4	2,2	2,4
Coarse Grid	256x1024	32x128	64x256	32x128
Fine Grid	256x1024	256x1024	256x1024	256x1024
Shock Tolerance	NA	30	460	460
Shock Onset	NA	.85	.95	.95
Cell Updates	3.51e9	1.27e9	9.84e8	7.73e8
Time per Cell Update	3.8e-6	9.6e-6	1.2e-5	9.8e-6
Fine Steps	9850	14164	17493	13033
Time (s)	13380	12106	12061	7589

Table 5.2: Timing results for RM2D NEUMANN effective 256x1024

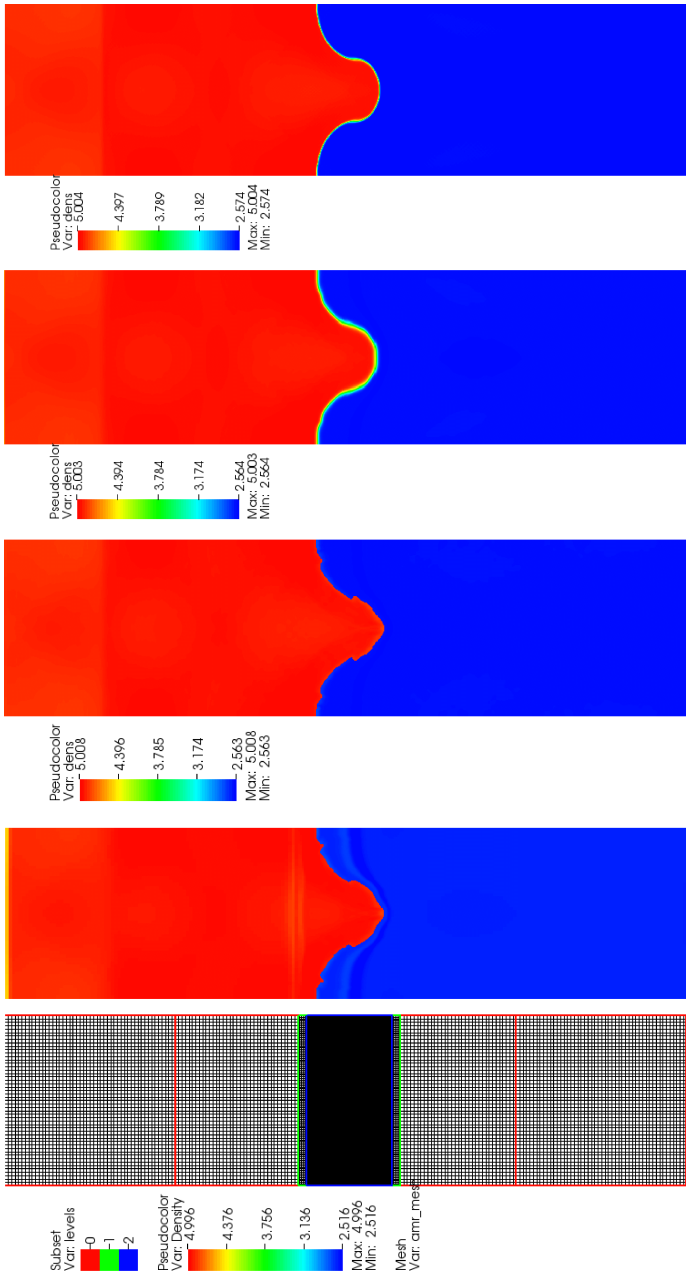


Figure 5.1: Richtmyer-Meshkov comparison at t = 3

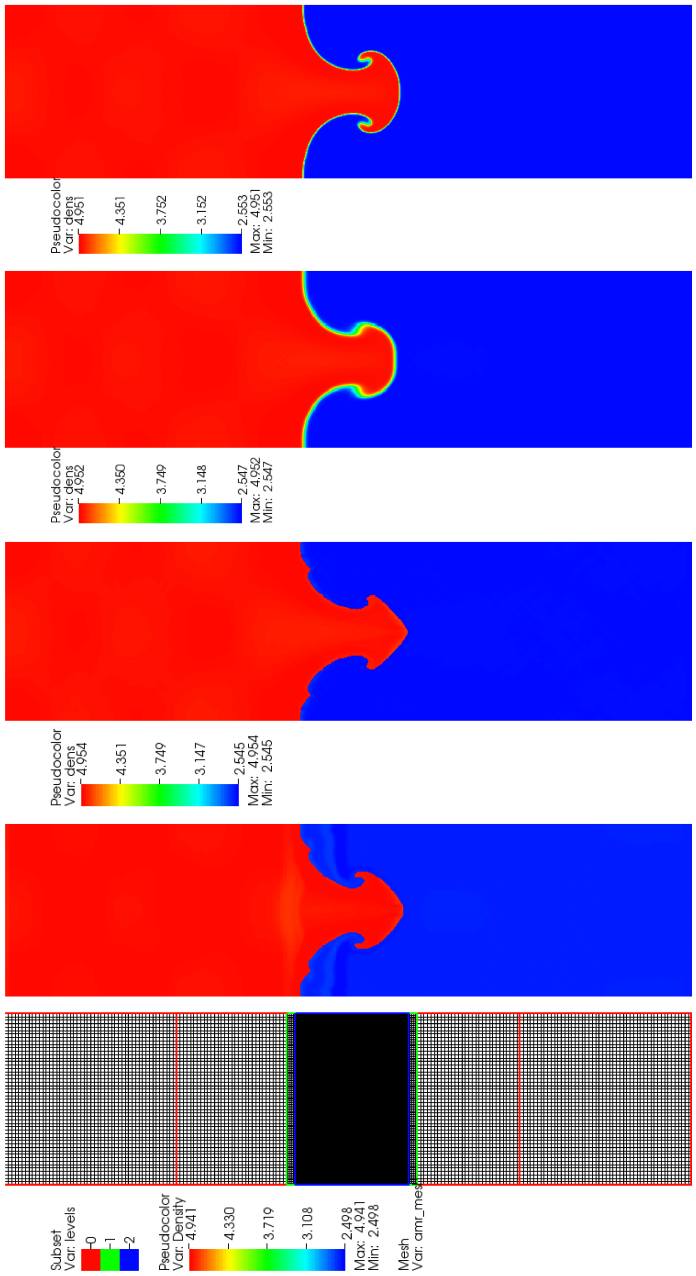


Figure 5.2: Richtmyer-Meshkov comparison at t = 6

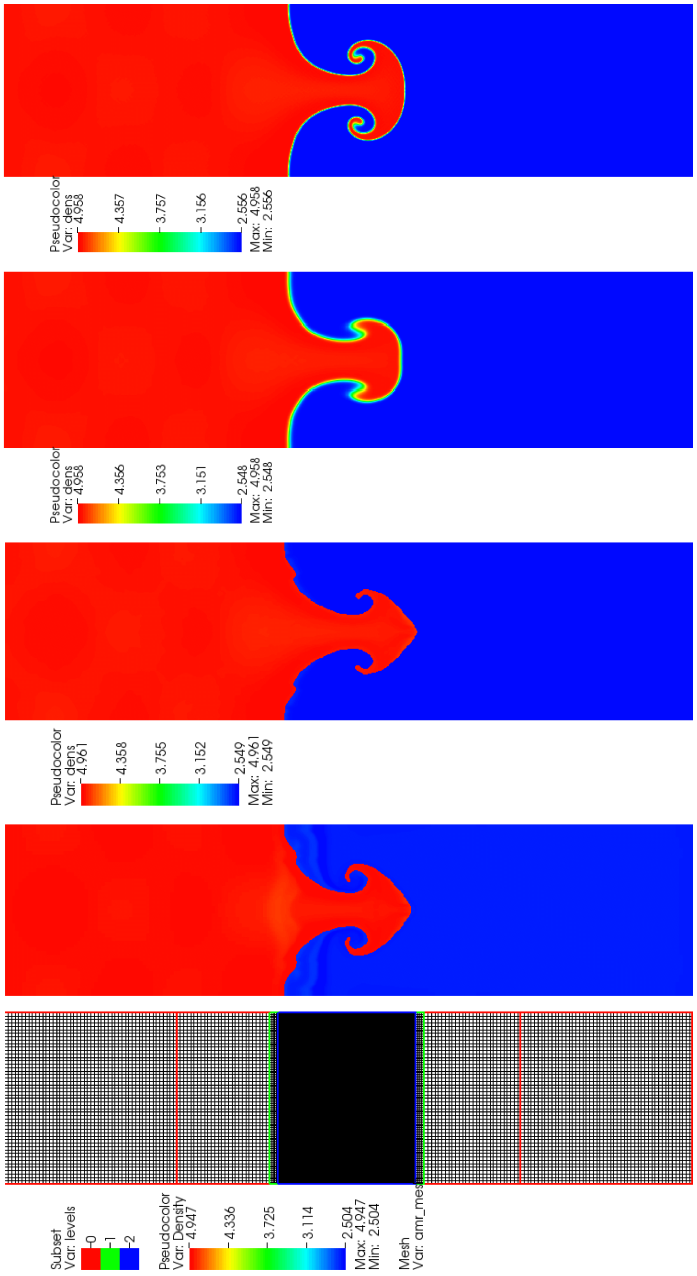


Figure 5.3: Richtmyer-Meshkov comparison at t = 9

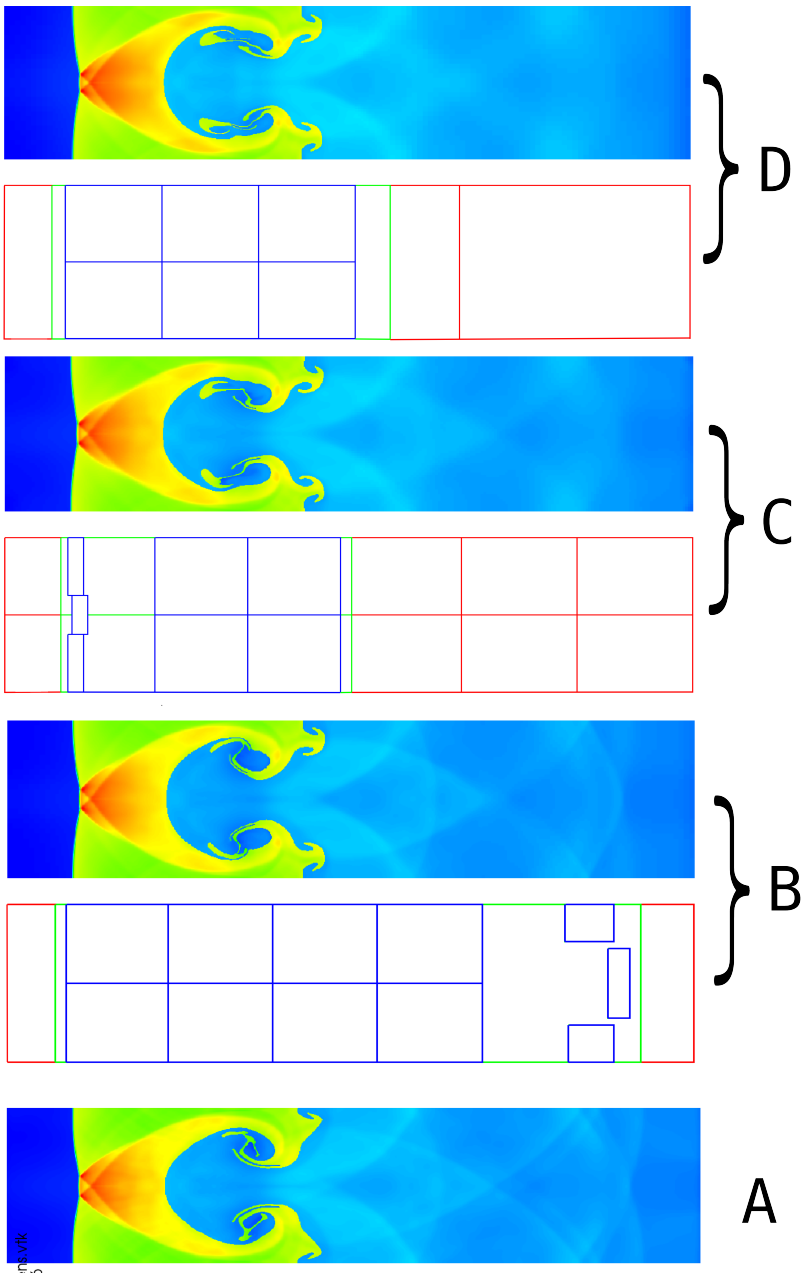


Figure 5.4: RM2D NEUMANN

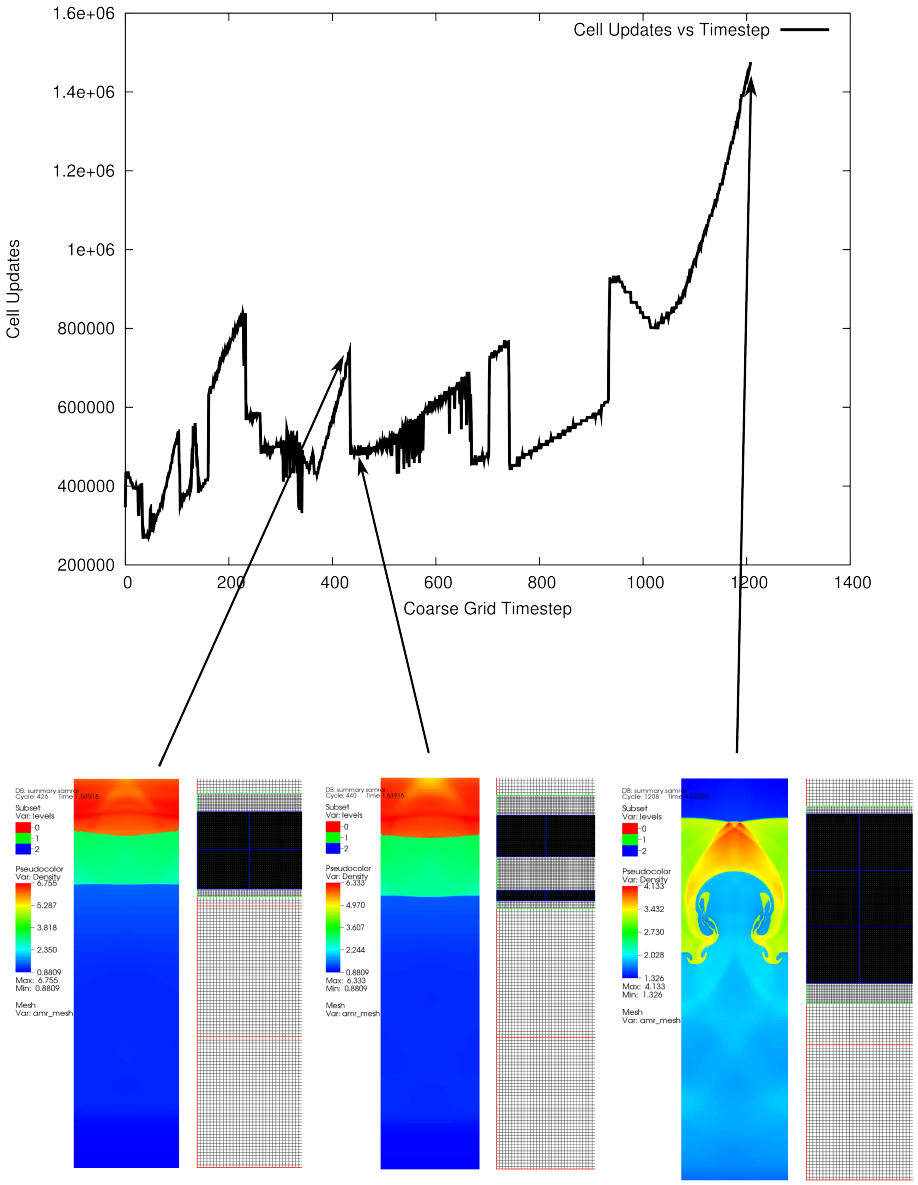


Figure 5.5: Cell operations vs coarse grid step

Chapter 6

Conclusion

We reviewed previous work on the interoperation between front tracking (FronTier) and adaptive mesh refinement (AMR), and the implementation algorithms by Xu, et al. The old algorithms were based on the unmodularized front tracking code which requires complex insertion and revision of the front tracking functions. The implementation added significant volume of specialized functionalities to the FronTier code and was error-prone.

We presented a new algorithm for the FronTier-AMR interoperation. This new algorithm is based on a modularized cFluid package and the FronTier-Lite library. The new algorithm significantly improves the code clarity and simplifies the coupling of FronTier and AMR. We replaced the Overture AMR package with the high quality SAMRAI library. We made one important change to the FronTier-AMR coupling by requiring that front propagation and ghost-fluid cell update only be carried out in the finest level of AMR patches. This change greatly reduced the need for the inter-level data communication of front geometry. It also allows the interface optimization and topological

reconstruction to be performed in a uniform grid, that is, the finest level of grid. This new algorithm is simpler with a significant reduction in number of functions in the code. We estimate that the new code for the interoperation between FronTier and AMR is ten times smaller than the previous one. The new code is easy to understand and debug. It is also more robust and more efficient for parallel communication and load balancing, and should provide significant speed ups.

The new cFluid based FronTier-AMR algorithm has been tested on several cases for the simulation of Richtmyer Meshkov instability. The results are comparable to solutions with fully refined mesh without AMR, while the computation is much faster. As with all AMR calculations, the parallel scaling and speed-up is dependent on the specific problem and parameters chosen, but the benefit of FronTier-AMR coupling is evident.

Our work shows that a truly adaptive method with both AMR and front tracking is possible.

Bibliography

[1] I-L. Chern, J. Glimm, O. McBryan, B. Plohr, and S. Yaniv. Front tracking for gas dynamics. *J. Comp. Phys.*, 62:83–110, 1986.

[2] C. W. Hirt and B. D. Nichols. Volume of fluid (vof) method for the dynamics of free boundaries. *J. Comp. Phys.*, 39(1):201–225, 1981.

[3] W. Mulder, S. Osher, and J. Sethian. Computing interface motion in compressible gas dynamics. *J. Comp. Phy.*, 100:209–228, 1992.

[4] M.J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, 1989.

[5] D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant, and J. E. Bussoletti. A locally refined rectangular grid finite element method: Application to computational fluid dynamics and computational physics. *J. Comp. Phys.*, 92(1):1–66, 1991.

[6] M. Gittings, R. Weaver M. Clover, T. Betlach, N. Byrne, R. Coker. E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, Dale Ranta, and R. Stefan. The rage radiation-hydrodynamic code. Technical Report LA-UR-06-0027, Los Alamos National Laboratory, 2004.

[7] D. Brown, K. Chand, W. Henshaw, B. Miller, J. Painter, R. Pember, B. Philip, D. Quinlan, and T. Rutaganira. Overture amr. http://acts.nersc.gov/overture/.

[8] J. Glimm, X.-L. Li, R. Menikoff, D. H. Sharp, and Q. Zhang. A numerical study of bubble interactions in Rayleigh-Taylor instability for compressible fluids. *Phys. Fluids A*, 2(11):2046–2054, 1990.

[9] E. George, J. Glimm, X. L. Li, A. Marchese, and Z. L. Xu. A comparison of experimental, theoretical, and numerical simulation Rayleigh-Taylor mixing rates. *Proc. National Academy of Sci.*, 99:2587–2592, 2002.

[10] J. W. Grove, R. Holmes, D. H. Sharp, Y. Yang, and Q. Zhang. Quantitative theory of Richtmyer-Meshkov instability. *Phys. Rev. Lett.*, 71(21): 3473–3476, 1993.

[11] R. L. Holmes, J. W. Grove, and D. H. Sharp. Numerical investigation of Richtmyer-Meshkov instability using front tracking. *J. Fluid Mech.*, 301: 51–64, 1995.

[12] Z. L. Xu, M. Kim, T. Lu, W. Oh, J. Glimm, R. Samulyak, X. L. Li, and C. Tzanos. Discrete bubble modeling of unsteady cavitating flow. *International Journal for Multiscale Computational Engineering*, 4:601–616, 2006.

[13] J. Glimm, E. Isaacson, D. Marchesin, and O. McBryan. Front tracking for hyperbolic systems. *Adv. Appl. Math.*, 2:91–119, 1981.

[14] J. Glimm and O. McBryan. A computational model for interfaces. *Adv. Appl. Math.*, 6:422–435, 1985.

[15] J. Glimm, J. W. Grove, X.-L. Li, K.-M. Shyue, Q. Zhang, and Y. Zeng. Three dimensional front tracking. *SIAM J. Sci. Comp.*, 19:703–727, 1998.

[16] J. Glimm, J. W. Grove, X.-L. Li, and D. C. Tan. Robust computational algorithms for dynamic interface tracking in three dimensions. *SIAM J. Sci. Comp.*, 21:2240–2256, 2000.

[17] X.-L. Li, B. X. Jin, and J. Glimm. Numerical study for the three dimensional Rayleigh-Taylor instability using the TVD/AC scheme and parallel computation. *J. Comp. Phys.*, 126:343–355, 1996.

[18] R. P. Fedkiw, T. Aslam, B. Merriman, and S. Osher. A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *J. Comp. Phys.*, 152:457–492, 1999.

[19] M.J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–512, 1984.

[20] S. Osher and R. Sanders. Numerical approximations to nonlinear conservation laws with locally varying time and space grids. *Math. Comput.*, 41:321–336, 1983.

[21] M. J. Berger. On conservation at grid interfaces. *SIAM J. Num. Anal.*, 24:967, 1987.

[22] Z. L. Xu, J. Glimm, Y. Zhang, and X. Liu. A multiscale front tracking method for compressible free surface flows. *Chemical Engineering Science Journal*, 62:3538–3548, 2007. SB Preprint Number: SUNYSB-AMS-06-09.

[23] Z. Xu, R. Samulyak, X. Li, and C. Tzanos. Atomization of a High Speed Jet. *APS Meeting Abstracts*, pages G2+, November 2005.

[24] Selenic Consulting. Mercurial is a free, distributed source control management tool. http://mercurial.selenic.com.

[25] David A. Wheeler. Sloccount: a set of tools for counting physical source lines of code. http://www.dwheeler.com/sloccount.

[26] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[27] R. D. Richtmyer. Taylor instability in shock acceleration of compressible fluids. *Comm. Pure Appl. Math.*, 13:297–319, 1960.