Parallelization of Structured, Hierarchical Adaptive Mesh Refinement Algorithms[*]

Charles A. Rendleman
Vincent E. Beckner
Mike Lijewski
William Crutchfield
John B. Bell

Lawrence Berkeley National Laboratory
Berkeley, CA 94720

April 20, 1999

Abstract

We describe an approach to parallelization of structured adaptive mesh refinement algorithms. This type of adaptive methodology is based on the use of local grids superimposed on a coarse grid to achieve sufficient resolution in the solution. The key elements of the approach to parallelization are a dynamic load-balancing technique to distribute work to processors and a software methodology for managing data distribution and communications. The methodology is based on a message-passing model that exploits the coarse-grained parallelism inherent in the algorithms. The approach is illustrated for an adaptive algorithm for hyperbolic systems of conservation laws in three space dimensions. A numerical example computing the interaction of a shock with a helium bubble is presented. We give timings to illustrate the performance of the method.

Keywords: adaptive mesh refinement, parallel processing, conservation laws, load balancing, scientific computing

1 Introduction

Advanced, higher-order finite difference methods and local adaptive mesh refinement have proven to be an effective combination of tools for modeling problems in fluid dynamics. However, the dynamic nature of the adaptivity in time dependent simulations makes it considerably more difficult to implement this type of methodology on modern parallel computers, particularly, distributed memory architectures. In this paper we present a software framework that facilitates the development of adaptive algorithms for multiple-instruction, multiple-data (MIMD) architectures. The particular form of adaptivity we consider is a block-structured style of refinement, referred to as AMR, that was originally developed by Berger and Oliger [5]. The development of the methodology here uses the approach developed by Berger and Colella [4] for general systems of conservation laws and the extension of that methodology to three dimensions by Bell et

[*]Support for this work was provided by the Applied Mathematical Sciences Program of the DOE Office of Mathematics, Information, and Computational Sciences and by the Defense Threat Reduction Agency IACRO98-3017, through the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

al. [2]. This type of adaptive refinement has been extended to incompressible flows [1] and to low Mach number models for atmospheric flow [22] and combustion [18].

AMR is based on a sequence of nested grids with finer and finer mesh spacing in space, each level being advanced in time with time step intervals determined by the Courant-Friedrich-Level (CFL) condition. The fine grids are recursively embedded in coarser grids until the solution is sufficiently resolved. An error estimation procedure automatically determines the accuracy of the solution and grid management procedures dynamically create rectangular fine grids where required to maintain accuracy or remove rectangular fine grids that are no longer required for accuracy. Special difference equations are used at the interface between coarse and fine grids to insure conservation.

Several approaches to parallelization of AMR applied to hyperbolic conservation laws have been addressed in the literature. Crutchfield [9, 10] developed a prototype parallel implementation of a two-dimensional, adaptive gas dynamics algorithm using a coarse-grained MIMD model. Berger and Saltzman [3] used a data-parallel model to implement an adaptive gas dynamics algorithm on a Thinking Machines CM-2. Colella and Crutchfield [7] implemented a task-queue parallel model for a multifluid algorithm on a Cray C-90 which was used by Greenough et al. [13] to study a mixing layer. A major departure of the work presented here is the development of software infrastructure that provides a general framework for parallel, block-structured AMR algorithms. In the present approach, data distribution and communication are hidden in C++ class libraries that isolate the application developer from the details of the parallel implementation.

In the next section we will review the basic algorithmic structure of AMR and discuss the components of the algorithm in the context of hyperbolic conservation laws in more detail. The dynamic character of AMR leads to a dynamic and heterogeneous work load. In section 3 we discuss a load-balancing algorithm based on a dynamic-programming formulation that is used to control data distribution to processors. Section 4 provides a description of the parallel implementation focusing on the programming model used by the application developer and the support for that model provided by the underlying software. We illustrate the methodology for three-dimensional gas dynamics on computation of a shock-bubble laboratory experiment of Sturtevant and Haas [15]. We discuss overall performance and parallel efficiency in the context of this case study.

2 The Adaptive Mesh Refinement Algorithm

AMR solves partial differential equations using a hierarchy of grids of differing resolution. The grid hierarchy is composed of different levels of refinement ranging from coarsest ($l = 0$) to finest ($l = l_{max}$). Each level is represented as the union of rectangular grid patches of a given resolution. In this work, we assume the level 0 grid is a single rectangular parallelepiped, the *problem domain*, although it may be decomposed into several coarse grids. In this implementation, the refinement ratio is always even with the same factor of refinement in each coordinate direction, i.e., $\Delta x^{l+1} = \Delta x^l / r$, where r is the refinement ratio. (We note the basic concepts used in AMR require only a logically rectangular grid; neither isotropic refinement nor uniform base grids are requirements of the fundamental algorithm.) In the actual implementation, the refinement ratio can be a function of level; however, in the exposition we will assume that r is constant. The grids are *properly nested*, in the sense that the union of grids at level $l + 1$ is contained in the union of grids at level l for $0 \leq l < l_{max}$. Furthermore, the containment is strict in the sense that, except at physical boundaries, the level l grids are large enough to guarantee that there is a border at least one level l cell wide surrounding each level $l + 1$ grid. However, a fine grid can cross a coarser grid boundary and still be properly nested. In this case, the fine grid has more than one parent grid. This is illustrated in Figure 1 in two dimensions. (This set of grids was created for a problem with initial conditions specifying a circular discontinuity.) Grids at all levels are allowed to extend to the physical boundaries so the proper nesting is

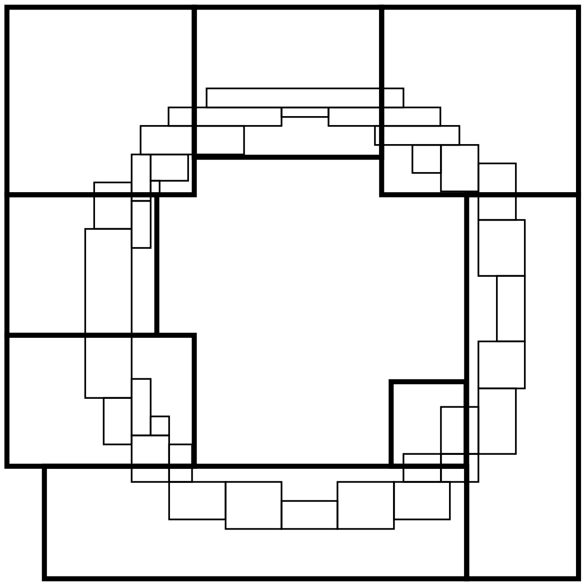

Figure 1: Two levels of refined grids. Grids are properly nested, but may have more that one parent grid. The thick lines represent grids at the coarse level; the thin lines, grids at the fine level.

not strict there.

Both the initial creation of the grid hierarchy and the subsequent regriding operations in which the grids are dynamically changed to reflect changing flow conditions use the same procedures to create new grids. Cells requiring additional refinement are identified and tagged using an error estimation criteria. The error estimation criteria may use Richardson extrapolation to estimate the error, as described in Berger and Colella [4], or it may use some other user-supplied criterion. The tagged cells are grouped into rectangular patches using the clustering algorithm given in Berger and Rigoutsos [6]. The generated patches will in general contain cells that were not tagged for refinement. The *grid efficiency* is the fraction of the cells in a new grid that are tagged by the error estimation process. A grid efficiency criterion (typically 70%) determines the minimum grid efficiency that is acceptable. These rectangular patches are refined to form the grids at the next level. The process is repeated until either the error tolerance criteria are satisfied or a specified maximum level of refinement is reached. The proper nesting requirement is imposed at this stage.

The initial, $t = 0$, data is used to create grids at level 0 through l_{max}. (Grids have a user-specified maximum size, therefore more than one grid may be needed to cover the physical domain.) As the solution advances in time, the regriding algorithm is called every k_l (also user-specified) level l steps to redefine grids at levels $l+1$ to l_{max}. Grids at level $l+1$ are only modified at the end of level l time steps, but because we sub-cycle in time, i.e., $\Delta t_{l+1} = \Delta t_l / r$, level $l+2$ grids can be created and/or modified in the middle of a level l time step if $k_{l+1} < r$.

When new grids are created at level $l+1$, the data on these new grids are copied from the previous grids at level $l+1$ where possible, otherwise the data is interpolated in space from the underlying level l grids.

The method we use for solving partial differential equations on a grid hierarchy is to solve on a given level using Dirichlet data obtained from coarser levels. This results in flux errors at the boundary with the coarse grid, which are then fixed in a synchronization step described below. The time-step algorithm advances grids at different levels using time steps appropriate to that level based on CFL considerations and the flux corrections are typically imposed in a time-averaged sense. The AMR algorithm, shown in Figure 2, is a recursive algorithm that advances levels l, $0 \leq l \leq l_{max}$. The recursive invocation of the **Advance** is repeated r times; r being the refinement ratio for level l. The repetition is required to comply with the CFL

```
**Recursive Procedure** Advance (level $l$)
    **if** time to regrid at level $l + 1$
        Estimate errors at level $l + 1$
        Generate new grids at level $l + 1$
        **if** $l + 1 < l_{max}$ then regrid $l + 2$
    **endif**
    **if** $l == 0$ obtain boundary data from physical
        boundary conditions.
    **else** obtain boundary data from coarser grids
        and from physical boundary conditions.
    Integrate level $l$ in time.
    **if** $l < l_{max}$
        **repeat** $r$ **times:** Advance (level $l + 1$)
    **endif**
    Synchronize the data between levels $l$ and $l + 1$.
**End Recursive Procedure** Advance
```

Figure 2: Basic AMR Algorithm

constraint: because the resolution of level $l + 1$ is r times that of level l, the time step must be cycled with steps r times smaller.

Before turning to the parallelization issues we first discuss, in the case of hyperbolic conservation laws, some of the details of the above algorithm, especially details related to communication of data between levels. Essentially all of the inter-level data communications occurs in two phases of the algorithm. The coarser grids supply boundary data in order to integrate finer grids, and the coarse and fine grids are synchronized at the end of fine grid time steps when the coarse and fine grid solution have reached the same time. For the case considered here, boundary data is provided by filling *ghost cells* in a band around the fine grid data whose width is determined by the stencil of the finite difference scheme. If this data is available from grids at the same level of refinement the data is provided by a simple copy. If the data is not available from grids at the same level, it is obtained by interpolation of the coarse grid data in time and space.

When the coarse and fine grids reach the same time and we synchronize, there are two corrections that we need to make. First, for all coarse cells covered by fine grid cells, we replace the coarse data by the volume weighted average of the fine grid data. Second, because coarse cells adjacent to the fine cells were advanced using different fluxes than were used for the fine cells on the other side of the interface, we must correct the coarse cell values by adding the difference between the coarse and fine grid fluxes. We note that for the case of explicit algorithms for conservation laws, the synchronization only involves corrections to the coarse grid. Almgren et al. [1] show that for other types of algorithms this synchronization can become considerably more complex.

The AMR software is organized into five relatively separate components. The error estimation and grid generation routines identify regions needing refinement and generate fine grids to cover the region. Grid management routines manage the grid hierarchy allowing access to the individual grids as needed. Interpolation routines initialize a solution on a newly created fine grid and also provide the boundary conditions for integrating the fine grids. Synchronization routines correct mismatches at coarse/fine boundaries arising because we integrate the levels independently, apart from boundary conditions. Finally, the integration routines discretize the physical processes on the grids.

3 Parallelization of AMR—Load Balance

We have adopted a coarse-grained, message-passing model in our approach to parallelization. This approach is generally associated with distributed memory MIMD architectures, but it can be used on shared memory architectures as well. We make this choice because it enhances portability on distributed memory architectures, and because we feel message-passing programs are more robust. In message-passing parallel programs, the only communication between processors is through the exchange of messages. Direct access to another processor's memory is not provided. In this approach, it is critical to choose carefully which processor has which piece of data. As is apparent from Figure 1, grids vary considerably in size and shape. In AMR the number of grids also changes and is seldom an integer multiple of the number of processors. It is therefore inefficient to assign the grids sequentially to the processors, since the result is unlikely to be load balanced. We will use a load-balancing strategy based on the approach developed by Crutchfield [9]. Because most of the data and computational effort is required by the finest level of grids, we need only be concerned with load-balancing the grids on the finest level. In general, the effort required by the coarser grids will be a minor perturbation.

In the serial version of the AMR algorithm, extra work is performed at the boundaries of grids (obtaining boundary data and synchronization). As a result, the overhead is reduced if the points tagged for refinement are covered with a small number of large grids, rather than a large number of small grids. However, it is generally true that decreasing the number of grids also decreases the grid efficiency. The regriding algorithm of Berger and Rigoutsos [6] used in the serial algorithm [2] seeks to optimize the computational efficiency of the grids by reducing the number of grids for a given minimum grid efficiency.

In a parallel algorithm, the necessity of assigning each grid to a processor introduces another complication to the regriding algorithm. One possible parallelization strategy for AMR would be to modify the regriding algorithm to introduce a secondary goal of producing grids that can be assigned to processors in a fashion that balances the load. However, this approach introduces nuances of the particular machine characteristics into the grid generation algorithm. For this reason, we chose not to modify the regriding algorithm. We will accept the set of grids provided by the regriding algorithm and seek to find a well-balanced assignment of grids to processors. It turns out to be possible to find well-balanced assignments if we can make the following assumptions.

1. The computational cost of a grid can be estimated using some type of work estimate.
2. The total computational cost of the algorithm is well approximated as the sum of the costs of time-stepping the grids on the finest level. Other costs such as communications, time-stepping coarser grids, regriding, refluxing, etc., are treated as ignorable perturbations.
3. The grids can be approximated as having a broad random distribution in work estimate, i.e., that the standard deviation of the distribution is not small compared to the average.
4. The average number of grids per processor is large enough.

We can state the load-balancing problem in simple terms. Suppose we have K processors and N grids. The grids have random sizes and shapes taken from some distribution. The problem is to assign the grids to the processors so that the processors have nearly equal amounts of work. Let W_i^α be the work required for the i^{th} grid when the work is performed on processor α. The task for the load balance routine is to minimize the load balance inefficiency, which is defined as

$$\text{Inefficiency}_{LB} = 1 - \frac{\sum_{\alpha i} W_i^\alpha}{K \max_\alpha \sum_i W_i^\alpha}.$$

The minimization is over all possible distributions of grids onto processors. The term $\max_\alpha \sum_i W_i^\alpha$, in the denominator (the summation is over all grids assigned to processor α,) is the work load on the most loaded processor. The numerator, $\sum_{\alpha i} W_i^\alpha$, is the total work load on all processors. Therefore, the Inefficiency$_{LB}$, is simply related to the standard measure of parallel efficiency. This problem is an application of the knapsack dynamic programming algorithm, a description of which may be found in Sedgewick's book on algorithms [19]. In general, the knapsack problem is NP complete and finding the best assignment of grids to processors will require $O(N!)$ possible operations, which is not practical. Below is a heuristic algorithm for finding a good, if not optimal load balance.

```
procedure KNAPSACK
        Sort grids by size, those with the most work
            first.
        foreach grid, starting with the one with the
            most work, assign grid to the least loaded
            processor.
        endfor
L1:     Find the most loaded processor.
        foreach grid $i$ on the most loaded processor
            foreach grid $j$ not on the most loaded
                    processor
                if interchanging $i$ and $j$ improves LB,
                    perform interchange,
                    goto L1.
                endif
            endfor
        endfor
endproc KNAPSACK
```

Figure 3 is a plot of average inefficiency versus the average number of grids per processor. The data points in the figure were determined by applying the **KNAPSACK** algorithm to random test data and averaging the results of many trials. Several curves are presented, corresponding to different numbers of processors. Also presented, for purposes of comparison, is the load balance inefficiency for grids with fixed work estimate—the non-random case—where all grids are the same size. The distribution of the random test data was chosen to be similar in shape, mean, and variance to the distribution of grids produced by the regriding process of AMR. The results are not very sensitive to the choice of probability distribution. The inefficiency of the non-random case goes to zero when the number of grids is an integer multiple of the number of processors, as indicated by Figure 3. Note that the curves for randomly sized grids have general properties independent of the number of processors. When the ratio N/K is near one, the average inefficiency of the random work estimate cases is higher than the inefficiency of the non-random case. When the average number of grids per processor exceeds 2, the random distribution is always more efficient than the non-random distribution except when the average number of grids per processor is slightly less than an integer. The random variation in the work estimate of the grids allows extra freedom to improve the load balance by the interchange of grids. The inefficiency of the random case decreases faster than the inefficiency of the non-random case as the average number of grids per processor increases (excluding the points where N/K is integral.)

Figure 3 demonstrates that load balancing is easier when the work per grid is a random distribution than when the work per grid is uniform, provided that N/K is large enough. It also shows that the factor that

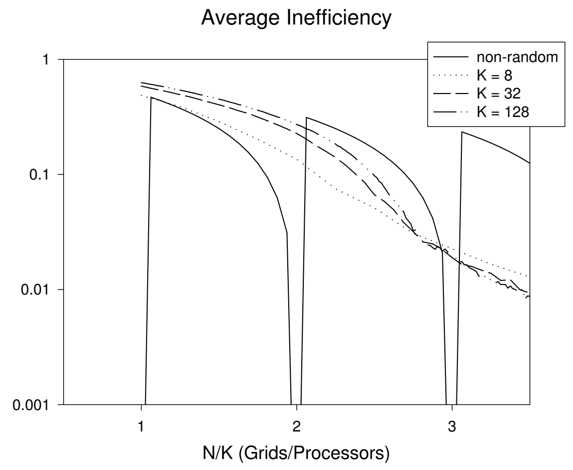

Figure 3: Average load balance inefficiency versus average number of grids per processor for differing numbers of processors. For comparison purposes, the load balance for non-randomly sized grids is also shown.

controls the quality of the load balance is N/K, the ratio of the number of grids to the number of processors. When the ratio is approximately three or greater, the balance is excellent. The inefficiency is only weakly dependent on the probability distribution or the number of processors.

The work estimate of a grid corresponds to the time estimate for computing a time-step on a grid. In the following, we estimate the integration time as being proportional to the number of grid points in the grid. The estimate is good but not exact since integration times are dependent on other variables as well. For example, for the operator split algorithm we use for gas dynamics, extra work is performed in boundary regions in the initial sweeps to provide updated boundary data for the later sweeps. This results in a mild dependence of the work on the shape of the grid. Also, points on the physical boundary of the domain contribute a different amount to the computational cost assigned to a grid. Such refinements have not been incorporated into the time estimate.

As previously stated, this load-balancing technique requires several assumptions about the computational scheme that is being balanced. We have previously commented about the estimation of computational cost of the grid, and the random distribution of grid sizes. The last assumption is that computational costs other than the cost of time-stepping the finest grids are ignorable. Communication costs are generally important in message-passing parallel programs, and effort is devoted to its reduction. Communication costs are ignored in this load balance scheme. No effort is made to reduce communication costs by placing adjacent grids on the same processor, or on adjacent processors. This is justified in the case of the AMR algorithm for gas dynamics considered here since the Godunov time-step integration is floating-point intensive requiring approximately a thousand operations per zone in three dimensions. Since the ratio of communication cost to calculation cost for modern multiprocessors is not overly large, it is reasonable to ignore communication costs in the load balance. We also ignore the computational costs of time-stepping coarse grids because, consisting of fewer grid points, they are much less expensive to time-step. Similarly we expect the costs of regriding and refluxing to be ignorable.

The utility of the load-balancing technique presented in this section is not limited to specific case of AMR for hyperbolic conservation laws presented here. The only requirements that must be met are that it be possible to estimate the execution time for the irregularly sized units, that the random distribution of data unit sizes be reasonably broad, and that the total computational cost is well approximated as the sum of the computational costs of the irregularly sized data units.

4 Implementation

This section describes the implementation of the software described in the previous sections from a perspective of the application programmer. Following some initial remarks, we describe the software infrastructure, with special emphasis given to the two constructs that were added to the serial version to achieve parallelism.

Prior to the development of the parallel version of the hyperbolic system software, a serial version was written, consisting of more that 50,000 lines of C++ and FORTRAN code. Several other software systems for solving partial differential equations were also written using the same software framework. Therefore, a significant design point for the parallel implementation of the software was that the work of the application programmer be minimized, both in the conversion of existing software and the writing of new AMR applications. The serial versions of the software were written using a C++ foundation library which, among other tasks, managed data-structures that are passed to FORTRAN routines for numeric processing. We chose to implement parallelism at this level, using a Single Program, Multiple Data (SPMD) approach. In this approach FORTRAN data is distributed and each processor independently processes its FORTRAN compatible data.

4.1 Software Infrastructure

The methodology described in this paper has been embodied in a software system that allows for a broad range of physics applications. It is implemented in a hybrid C++/FORTRAN programming environment where memory management and control flow are expressed in the C++ portions of the program and the numerically intensive portions of the computation are handled in FORTRAN. The software is written using a layered approach, with a foundation library, BoxLib, that is responsible for the basic algorithm domain abstractions at the bottom, and a framework library, AMRLib, that marshals the components of the AMR algorithm, above it. Other support libraries, built with BoxLib, are used as necessary to implement application components such as the interpolation of data between levels and the coarse-fine interface synchronization routines.

The foundation library, BoxLib, and a previous framework library, called libamr, has been described by Crutchfield and Welcome [11]. BoxLib has been modified and improved from the version described in that paper, but has kept its core functionality and data structures intact. The improvements were made incrementally, so that existing application programs would be minimally impacted. The framework library, libamr, however was abandoned when it proved inflexible for applications other than simulating hyperbolic systems of conservation laws.

The foundation library BoxLib provides support for programs solving finite difference equations on domains that are unions of non-intersecting rectangles. The library presents to the application programmer several abstractions: a global index space, labeled by tuples of integers; rectangular regions of that index space; non-intersecting unions of rectangular regions; FORTRAN compatible data defined over rectangular regions; and FORTRAN data defined over unions of non-intersecting rectangular regions. Each of the abstractions provides a rich set of operations. For example, the Box class is used to represent a rectangular region of index space. Among many other operations, Boxs can be resized, shifted, or intersected with other Boxs. The FORTRAN compatible data objects, usually floating point numbers, are designed to be used with FORTRAN-77 style subroutines. They are allocated as a contiguous block of memory and the FORTRAN routines interpret them as multi-dimensional arrays. The number of dimensions in the FORTRAN compatible data is equal to the problem's spatial dimensionality plus one. This allows the definition of multi-component objects defined at each point in the index space. It is useful to note that we are able to implement a large portion of the code in a space-dimension independent manner by defining the dimension as a compile time parameter.

BoxLib owes many of its concepts to Hilfinger and Colella's FIDIL [16] language for defining algorithms for solving partial differential equations. However, FIDIL is much more general and expressive, allowing the user to express directly mathematical operations on data defined on non-rectangular domains of index space, including domains not defined as a union of non-intersecting rectangles. Unfortunately, FIDIL has not become widely available. LPARX, described by Kohn et al. [17], and KeLP, described by Fink [12], are other libraries that owe much of their conceptual foundation to FIDIL. These libraries have similar purposes to the parallel implementation of BoxLib described in this paper.

The framework library, AMRLib, supports the flow of control and data management of a time-dependent AMR application through its decomposition into problem dependent and problem independent components. The problem dependent parts include the particular hyperbolic systems to be solved (and a suitable integration scheme), the initial and boundary conditions, the problem domain, and the error estimation routines. As a consequence of the component-wise decomposition, adapting an existing integration module for use with the AMR algorithm is usually straightforward. When a new problem is being set up the changes required to the code are localized to overriding certain virtual functions of a support class, `AmrLevel`, in the framework. In addition, the programmer designates some physical quantities as *state data*; for example, the usual conserved quantities of gas dynamics together with any additional scalar conserved variables. The framework then provides for the efficient allocation in memory of the `FORTRAN` compatible data associated with the state data, the storage of restart/checkpoint and plot files. The state data also define a quantity's coarse-to-fine grid interpolation method used in regriding and boundary patch filling. The remainder of the AMR framework treats the data in a problem independent fashion, usually as a list of state data. Thus, the data structures, memory management, grid generation algorithms, time-step control, the sub-cycling on sub-grids, interior boundary conditions, and the interfacing between grids to insure conservation are nearly completely divorced from the particular system being solved.

4.2 Parallel Implementation

As stated above, because of the considerable body of AMR code that uses the AMRLib framework and the BoxLib foundation libraries, we implemented parallel support in such a way as to minimize the additional work for application programmers. The fundamental parallel abstraction in BoxLib is the `MultiFab`, which is the class that encapsulates the `FORTRAN` compatible data defined on unions of non-intersecting `Box`s. The grids that make up the `MultiFab` are distributed among the processors, with AMRLib assigning grids to processors using the distribution given by the load balance scheme described in section 3. The processor assignment of grids is managed by a `DistributionMapping` object, which also caches information necessary for efficient message passing of ghost cell data. Non-`MultiFab` operations and data structures are replicated on each processor. For example, the index space manipulations that determine sub-grid intersections are repeated and the results sometimes cached on each processor. This non-parallel work is usually measured to be small. One positive consequence of the replicated index space manipulation is that most exchanges of ghost cell data can be done with only one message: because each processor possesses the global data layout, processors can post send and receive requests with other processors without an initial query phase to determine data size and location.

Operations involving `MultiFab`s are performed in one of two different ways depending on the implicit communications pattern. In the simplest case, there is no interprocessor communication; the calculation is parallelized trivially with each processor operating independently on its local data. The simple calculation may involve several `MultiFab`s if their processor distributions are the same. These operations are

```
void LevelAdvance{MultiFab& mf,
            const MultiFab& other_mf)
{
  fillBoundary( mf ); // Boundary fill.
  for ( int i = 0; i < mf.length(); ++i )
  {
    advance(mf[i], other_mf[i]); // step.
  }
}
```

Figure 4: A prototypical, serial singly-nested loop

```
void LevelAdvance(MultiFab& mf,
            const MultiFab& other_mf)
{
  fillBoundary( mf ); // Boundary fill.
  FabIterator fi(mf);
  DependentFabIterator data(other_mf, fi);

  while ( fi.good() )
  {
    advance(*fi, *data); // Time step.
    fi.next(); // Next iteration.
  }
}
```

Figure 5: A prototypical, parallel singly-nested loop

implemented as singly nested loops over the grids in the `MultiFab`. The more complicated case will be described later.

Fortunately, most of the code using the serial BoxLib library was written in a straightforward way with simple, singly nested loops over grids at a fixed level of refinement. In order to demonstrate the style of coding for such loops we present a simple example. The procedure `LevelAdvance`, shown in Figure 4, advances a level of grids through one time-step. In this example, the target `MultiFab`, `mf`, has its ghost cells filled using the `fillBoundary` procedure. Then, the `FORTRAN` compatible data is accessed, grid by grid, and further processed using an external procedure, `advance`. The `FORTRAN` compatible data is accessed using an overloaded C++ array reference operator, i.e., `mf[i]`. In this example, The external procedure takes a second `FArrayBox` argument, provided by indexing into *other_mf*.

Figure 5 shows the same loop in parallel BoxLib. Again, the ghost cells in *mf* are filled using the external procedure `fillBoundary`. The `fillBoundary` procedure would, in general, not be implemented as a simple loop over the constituent `FArrayBox`s in the `MultiFab`; its style of implementation is discussed later. The parallel looping construct follows the SPMD model: all processors execute the loop, but different grids are processed on each processor. The `FabIterator` and `DependentFabIterator` are abstractions of loop indices that permit referring to the `FORTRAN` compatible data of a `MultiFab`. On a single processor computer, the `FabIterator` provides the same functionality as an integer looping index. On a multi-processor, the `FabIterator` only iterates over the grid data which is local to the processor. Since all grid data is uniquely assigned to some processor, all grid data is processed in the loop. An attempt to access grid data with a different processor distribution will cause a run-time error. The `FabIterator`

```
void NaiveCopy(MultiFab& mf_to,
         const MultiFab& mf_from)
{
  FabIterator fi(mf_to);
  while ( fi.good() )          // Outer
  {
    FabIterator fo(mf_from);
    while ( fo.good() )        // Inner
    {
      if ( fi->intersects(*fo) )
        fi->copy(*fo);
      fo.next();
    }
    fi.next();
  }
}
```

Figure 6: Incorrect implementation of a `MultiFab` to `MultiFab` copy.

is said to *control* the execution of the loop. The `DependentFabIterator` provides access to data in a second `MultiFab` with the same processor distribution. It is said to be *dependent* because it provides access to data in a `MultiFab` in the same order as the data accessed by the control iterator. The final `fab.next()` statement advances the `FabIterator`. The loop is executed while the iterator is "good," i.e., when there are still locally unprocessed grids. There is no implied synchronization of processors at the end of this loop: the processor does not stall waiting for other processors to complete their loop bodies.

Because the `FabIterator` is an abstraction, it can be used to hide implementation details which may improve parallel performance. For example, the order in which data is delivered to the loop body is not defined. In addition, the `FORTRAN` compatible data which is delivered to the loop body need not be defined on the same `Box`s that define the `MultiFab`. All that is required is that iterators deliver all of the data in the `MultiFab` before the loop is completed. These properties permit the use of asynchronous message passing and provide the potential for the overlap of computation and message passing (currently not implemented.) It is important to note that the same loop structure can be used on purely shared-memory architectures. In that case separate threads of execution would execute the loop body using a task-queue approach.

A different parallel construction is necessary when communication is required between `MultiFab`s. In AMR applications this occurs in the context of multiply nested parallel loops. The simplest case arises in copying data from one `MultiFab` onto another `MultiFab` with a different processor distribution. Other examples are in the *fill patch* operation, which interpolates from coarse cell data on to overlying fine grid patches, and the previously mentioned`fillBoundary`, used to fill ghost cells. These loops cannot be handled using the same mechanism as for singly nested loops. To see why, consider the incorrectly coded example in Figure 6. The *Outer* loop body is executed only for the sub-grids within *mf_to* that are local to the processor. This implies that the *Inner* loop body is executed only if the *mf_from* and the *mf_to* grid data are local. It is easy to see that a *mf_from* sub-grid can only update an intersecting *mf_to* sub-grid if they both reside on the same processor. That is, if the distribution of sub-grids in the two `MultiFab`s is not identical, some grids that should be involved in the copy operation will not be copied.

To perform such calculations correctly, the loops are processed in two stages: data is exchanged between processors and then the local targets are updated. It is important to emphasize that only one message need be sent from a processor to another, since each processor can amalgamate individual data transfers into a single message using its knowledge of the parallel data layout. We illustrate, in Figure 7, the parallel

```
Procedure ParallelCopy (MultiFab& mf_to,
      const MultiFab& mf_from)
  MultiFabCopyDescriptor mfcd;
  foreach grid $i$ in mf_to
    foreach grid $j$ in mf_from
      if $i$ intersects $j$
        Register copy request in mfcd
      endif
    endfor
  endfor
  Gather remote data into mfcd.
  foreach grid $i$ on processor
    copy from mfcd to grid $i$
  endfor
End Procedure ParallelCopy
```

Figure 7: Parallel `MultiFab` to `MultiFab` copy.

implementation of a multiply nested loop by presenting the pseudo-code for the simple case where grid data is copied from one grid to another. The procedure `ParallelCopy` first creates a helper object *mfcd* of class `MultiFabCopyDescriptor` which is used to build the message-passing requests formed in the first, nested loop. This loop determines the intersecting portions of the *mf_to* and *mf_from* `MultiFab`s and builds a list of the intersecting regions and the grids that provide the data for the intersecting regions. After this initial loop is completed, the data needed by remote processors is sent, and the data needed from the remote processors is received. The messages associated with the sends and receives proceed concurrently using asynchronous message passing. When the data has been gathered, each local patch in the *mf_to* `MultiFab` that intersects a patch in the *mf_from* `MultiFab` is updated through a copy operation.

4.3 Miscellaneous Remarks

The examples shown in section 4.2 do not give an accurate impression as to the amount of work the application programmer needs to do to convert a serial AMR program to a parallel one. In these examples, it seems that nearly every line of the serial procedure in Figure 4 needs to be changed in converting it to the parallel version shown in Figure 5. However, of the more that 50,000 lines of code in the serial application, fewer that 500 needed to be changed to parallelize the simple, singly nested loops. The multiply nested loops involved a considerable number of changes. These are nearly completely hidden from the application programmer because they are implemented either within BoxLib or AMRLib or one of their auxiliary libraries.

Finally we note that we used the MPI [21, 14] message-passing library. We used only core functionality within the MPI library to ensure portability. Furthermore, we encapsulated the MPI specific library calls within a class, `ParallelDescriptor`, which presents to users of BoxLib an abstraction of a message-passing environment, The use of the `ParallelDescriptor` abstraction facilitates porting of our codes to other message passing environments.

5 Numerical Example

To test the parallel performance of the adaptive algorithm, we have modeled the interaction of a Mach 1.25 shock in air hitting a spherical bubble of helium. The case being modeled is analogous to one of the experiments described by Haas and Sturtevant [15]. The helium is a factor of 0.139 less dense than the surrounding air which leads to acceleration of the shock as it enters the bubble and a subsequent generation of vorticity that dramatically deforms the bubble.

The computational domain is a rectangular region with length x of 22.5 cm and width y and height z of 8.9 cm. The radius of the bubble is 2.25 cm. The bubble is centered at the point ($x = 16$ cm, $y = 4.45$ cm, $z = 4.45$ cm) and the shock is initialized at 13.5 cm in the x direction moving in the direction of increasing x. We use the operator-split second-order Godunov method of [8], with Strang [23] splitting. Reflecting boundary conditions are set on the constant z and constant y planes. To minimize the x extent of the problem, the inflow and outflow velocities on the constant x planes, as well as the interior fluid velocities, are set so the frame of reference is shifted to one in which the post-shock velocity is zero. We include a density perturbation of random phase and amplitude over a range of wave numbers to break the four-fold symmetry of the problem.

We use a γ-law equation of state for each gas with $\gamma_a = 1.4$ for air and $\gamma_f = 1.667$ for the helium. Mixtures of the two gases are modeled with an equation of state defined using effective γs,

$$\Gamma_c = \frac{1}{\frac{f}{\gamma_f} + \frac{(1-f)}{\gamma_a}}$$

for sound speeds and

$$\Gamma_e = 1 + \frac{1}{\frac{f}{\gamma_f - 1} + \frac{(1-f)}{\gamma_a - 1}}$$

for energy. The harmonic average used to compute Γ_c expresses the net volume change of a mixture of the gases in terms of their individual compressibilities. The sound speed defined by

$$c = \sqrt{\Gamma_c p/\rho},$$

is used in the integration routine for defining characteristic speeds and for approximate solution of the Riemann problem used to define fluxes. We assume that the two components of a mixed fluid cell all are at the same pressure. Pressure is computed from density and internal energy using Γ_e, namely,

$$p = (\Gamma_e - 1)\rho e$$

The formula used to compute Γ_e insures that mixing of the two fluids at the same pressure does not result in a pressure and internal energy change of the composite fluid.

Two sets of experiments are presented. We used the SGI/Cray T3E-900 (T3E, below) [20] at the NERSC facility at Lawrence Berkeley National Laboratory to perform these experiments. The T3E compute nodes are 450 MHz Digital Alpha 21164 processors, capable of up to 900 MFlops. The communication back plane is capable of high bandwidth bi-directional message traffic with low latency (Bandwidth, up to 600 MBytes per second, processor to processor; latency, approximately 1 microsecond per message.) Benchmarks performed on other machines will yield different results. The integration used six conserved quantities (mass, momentum, energy, and mass of helium.)

In the first set of experiments only a single level of refinement is used, i.e. no adaptivity. Three experiments were run, where the number of CPUs and the problem size increases by a factor of eight over the next coarsest experiment. The results of the experiment are summarized in Table 1. Each processor has one grid which is 20x20x20 not including boundary cells. The figure of merit we use to measure performance is the total wall-clock time per each cell advanced in the grid. The number of cells advanced gives an indication of the increase in work load with problem size. On the T3E, we used the intrinsic function `_rtc()`, normalized using the result of `IRTC_RATE()`, to measure wall-clock time. As noted above, the operator split

CPUS	Base Grid	Cells advanced	μ-sec/cell
4	80x20x20	3104000	70.7
32	160x40x40	49664000	75.2
256	320x80x80	794624000	115.5

Table 1: Single Level performance

CPUS	Effective Grid	Cells advanced	μ-sec/cell
4	320x80x80	142042944	95.7
32	640x160x160	1440509824	105.5
256	1280x320x320	16705155232	154.4

Table 2: Adaptive performance

algorithm performs additional work in boundary cells in early sweeps to provide accurate boundary data for later sweeps. This additional time is included in the time/cell reported in the table. The performance degrades with increasing processor number because inter-processor communication of interior ghost cell grid data increases: no communications are required for ghost cell regions that abut the physical boundary.

The second sets of experiments were fully adaptive. For these computations we allowed the program to refine the grid where the density gradient was large or in mixed regions where a non-negligible amount of both air and helium are present. For these computations we integrate to a fixed time after the shock has completely passed through the bubble. Again we perform three runs with each succeeding run having eight times more processors and the base grid having eight times more points. In these runs there are two levels of refinement, the first by a factor of 2 and the second by a factor of 4. We define the *effective resolution* to be the resolution the finest grid would have if it covered the entire problem domain. For example, the coarsest adaptive calculation has an effective resolution of 320x80x80, which is the same as the resolution of the finest single level example.

In Figure 8, we show volume renderings of the helium mass fraction at six times during the evolution of the bubble. The earliest frame shows the initial data. The next frame shows the compression of the bubble as the shock (moving left to right) passes through the bubble. Subsequent frames show the deformation of the bubble into a torus as the vortex ring generated as the shock traverses the bubble begins to control the dynamics. The bubble evolution qualitatively agrees with the experimental results of Haas and Sturtevant. In Figure 9 we show the helium mass fraction at the end of the simulation with the boxes used by the adaptive algorithm superimposed over the rendering. The plane of boxes to the right are refinements around the shock.

Performance data for these computations are summarized in Table 2. The number of cells advanced, as in the single level case, does not include boundary work required for the Strang splitting algorithm. The timings also include all of the additional work such as error estimation and grid generation associated with the adaptive algorithm. The changes in the μ-sec/zone provides some indication of the overhead associated with the adaptive algorithm. As we noted above, the coarsest adaptive run has equivalent resolution to the finest single level run. For this case, the use of adaptive mesh refinement reduced the total wall-clock time by a factor of 7 compared to the single level calculation. The effectiveness of AMR is also illustrated by the total wall-clock time for the adaptive runs. The ratio of the wall-clock time for the finest to coarsest adaptive calculations is 190 which is less than the theoretical best factor of 256 for perfectly scaled finite difference methods on a uniform grid. In the finer calculation, the refined grids are fitted more closely to the active region which decreases the amount of refined grid.

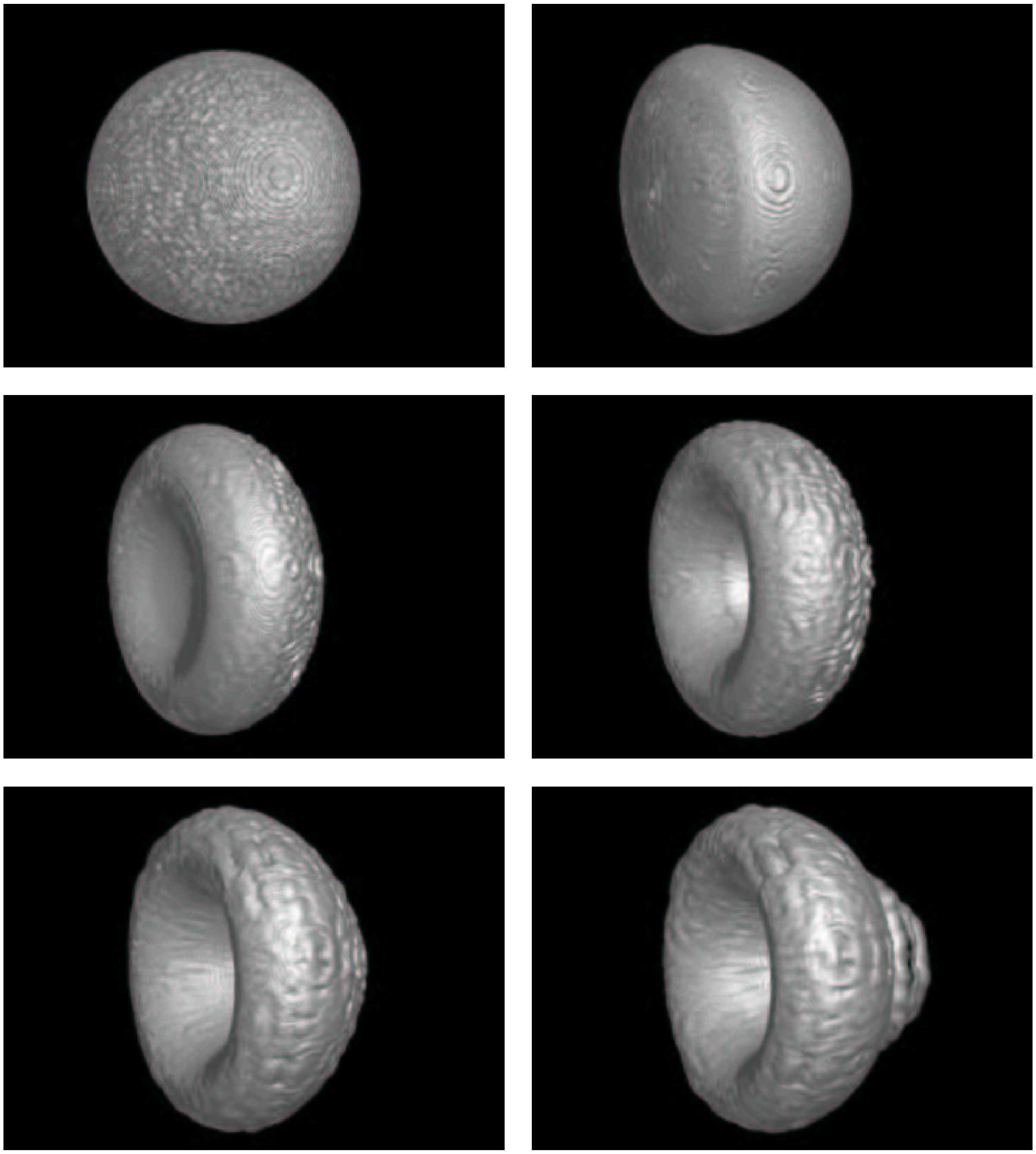

Figure 8: Temporal evolution of helium bubble. Rendering shows mass fraction of helium.

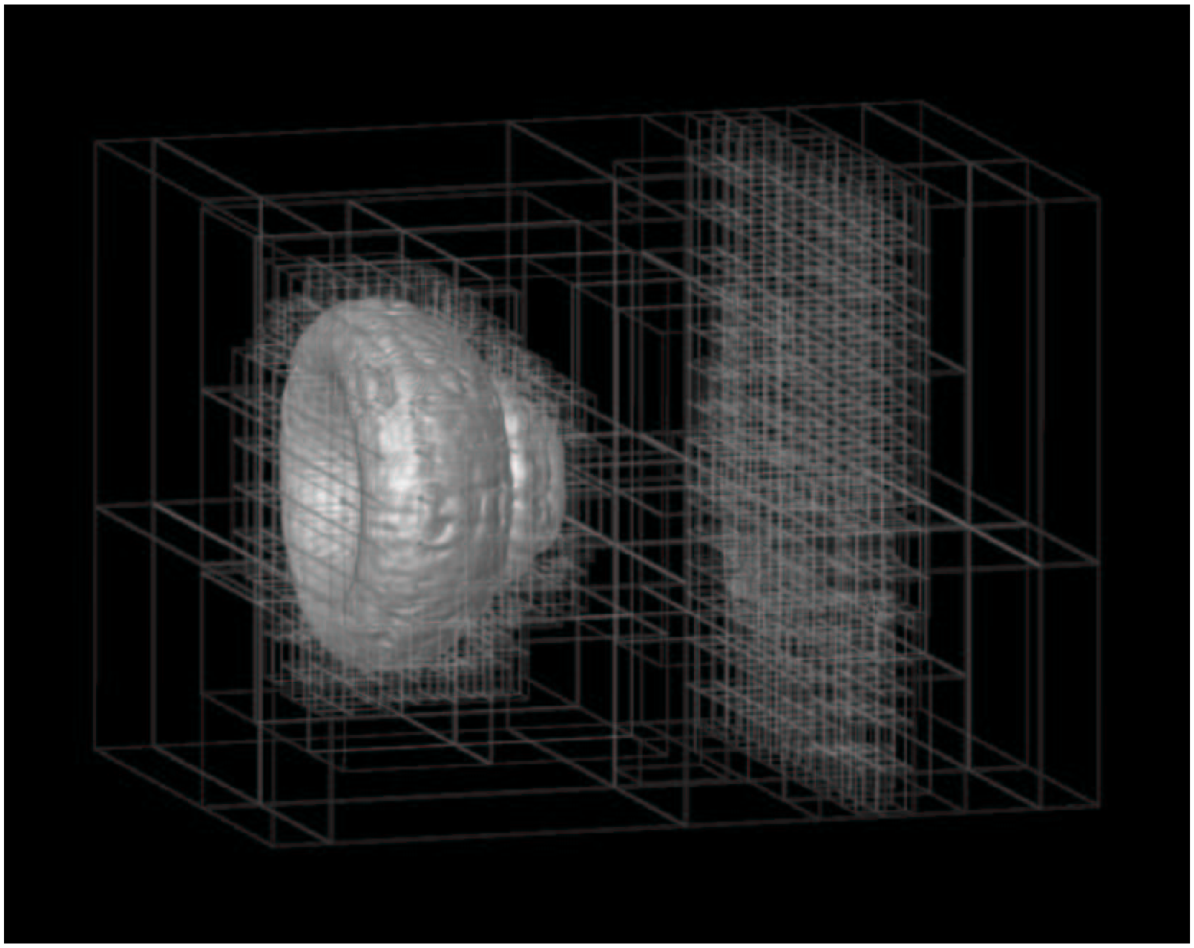

Figure 9: Volume rendering of the helium mass fraction at the end of the simulation with boxes used by the refinement scheme superimposed on the image.

6 Conclusions

The methodology presented here provides a strategy for parallelization of block-structured adaptive refinement algorithms. We described an efficient and effective dynamic-programming approach that achieves acceptable load balance for these algorithms. We also described the software infrastructure used in our application programs, and the changes we needed to make to produce parallel versions of our applications. For the case of hyperbolic conservation laws we have demonstrated that methods we have used provide are effective and have the further benefit of relieving the applications developer of much of the burden of parallelization.

We note that additional performance analysis of applications built in this framework offer the possibility of improving the parallel scaling. We are currently pursuing improvements to the methodology as well as using the software framework for other applications including a variety of low Mach number algorithms.

References

[1] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. *J. Comput. Phys.*, 142:1–46, May 1998.

[2] J. Bell, M. Berger, J. Saltzman, and M. Welcome. A three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM J. Sci. Statist. Comput.*, 15(1):127–138, 1994.

[3] M. Berger and J. Saltzman. AMR on the CM-2. *Applied Numerical Mathematics*, 14:239–253, 1994.

[4] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

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

[6] M. J. Berger and J. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:1278–1286, 1991.

[7] P. Colella and W. Y. Crutchfield. A parallel adaptive mesh refinement algorithm on the C-90. In *Proceedings of the Energy Research Power Users Symposium*, July 11–12, 1994. `http://www.nersc.gov/aboutnersc/ERSUG/meeting_info/ERPUS/colella.ps`.

[8] P. Colella and H. M. Glaz. Efficient solution algorithms for the Reimann problem for real gases. *J. Comput. Phys.*, 59(2):264–289, June 1985.

[9] W. Y. Crutchfield. Load balancing irregular algorithms. Technical Report UCRL-JC-107679, Lawrence Livermore National Laboratory, July 1991.

[10] W. Y. Crutchfield. Parallel adaptive mesh refinement: An example of parallel data encapsulation. Technical Report UCRL-JC-107680, Lawrence Livermore National Laboratory, July 1991.

[11] W. Y. Crutchfield and M. L. Welcome. Object-oriented implementation of adaptive mesh refinement algorithms. *Journal of Scientific Programming*, 2(4):145–156, 1993.

[12] Steven J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California, San Diego, November 1998.

[13] J. A. Greenough, W. Y. Crutchfield, and C. A. Rendleman. Numerical simulation of a wave guide mixing layer on a Cray C-90. In *Proceedings of the Twenty-sixth AIAA Fluid Dynamics Conference*. AIAA-95-2174, June 1995.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation. The MIT Press, Cambridge, Mass, 1994.

[15] J.-F. Haas and B. Sturtevant. Interaction of weak shock waves with cylindrical and spherical gas inhomogeneities. *J. Fluid Mech.*, 181:41–76, 1987.

[16] P. N. Hilfinger and P. Colella. FIDIL: A language for scientific programming. In Robert Grossman, editor, *Symbolic Computing: Applications to Scientific Computing*, Frontiers in Applied Mathematics, chapter 5, pages 97–138. SIAM, 1989.

[17] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under LPARX. *Journal of Scientific Programming*, 5(3):185–202, 1996.

[18] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee. An adaptive projection method for unsteady low-Mach number combustion. *Comb. Sci. Tech.*, 140:123–168, 1998.

[19] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[20] Silicon Graphics, Inc. The Cray T3E-900 scalable parallel processing system, 1999. `http://www.sgi.com/t3e/t3e_900.html`.

[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation. The Mit Press, Cambridge, Mass, 1996.

[22] D. E. Stevens, A. S. Almgren, and J. B. Bell. Adaptive simulations of trade cumulus convection. submitted for publication, 1998.

[23] G. Strang. On the construction and comparison of difference schemes. *SIAM J. Numer. Anal.*, 5:506–517, 1968.