

Enhancing Scalability of Parallel Structured AMR Calculations

A. M. Wissink, D. Hysom, R. D. Hornung

This article was submitted to
ACM International Conference on Supercomputing (ICS'03)
San Francisco, CA, 06/23/2003 - 06/26/2003

May 08, 2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This report has been reproduced directly from the best available copy.

Available electronically at <http://www.doc.gov/bridge>

Available for a processing fee to U.S. Department of Energy

And its contractors in paper from

U.S. Department of Energy

Office of Scientific and Technical Information

P.O. Box 62

Oak Ridge, TN 37831-0062

Telephone: (865) 576-8401

Facsimile: (865) 576-5728

E-mail: reports@adonis.osti.gov

Available for the sale to the public from

U.S. Department of Commerce

National Technical Information Service

5285 Port Royal Road

Springfield, VA 22161

Telephone: (800) 553-6847

Facsimile: (703) 605-6900

E-mail: orders@ntis.fedworld.gov

Online ordering: <http://www.ntis.gov/ordering.htm>

OR

Lawrence Livermore National Laboratory

Technical Information Department's Digital Library

<http://www.llnl.gov/tid/Library.html>

Enhancing Scalability of Parallel Structured AMR Calculations*

Andrew M. Wissink
Center Applied Sci. Comp.
Lawrence Lvmr. Natl. Lab.
Livermore, CA
awissink@llnl.gov

David Hysom
Center Applied Sci. Comp.
Lawrence Lvmr. Natl. Lab.
Livermore, CA
hysom@llnl.gov

Richard D. Hornung
Center Applied Sci. Comp.
Lawrence Lvmr. Natl. Lab.
Livermore, CA
hornung@llnl.gov

ABSTRACT

We discuss parallel performance of structured adaptive mesh refinement calculations using the SAMRAI library. We focus on fundamental aspects of adaptive gridding and dynamic computation of changing data dependencies. Previous analysis of performance of large-scale parallel adaptive calculations revealed poor scaling in these operations. Specifically, we found that these operations are inexpensive for small problems, but that their costs can become unacceptable for problems run on large numbers of processors. This paper describes subsequent developments involving graph- and tree-based algorithms that reduce runtime complexity and substantially increase scalability. We characterize performance on realistic adaptive problems using up to 512 processors of an IBM SP system and up to 1024 processors of a Linux cluster.

Categories and Subject Descriptors

G.4 [Mathematical Software]: Algorithm design and analysis, Efficiency, Parallel and vector implementations; I.6 [Simulation and Modeling]: General; J.2 [Physical Sciences and Engineering]: Engineering, Mathematics and statistics

General Terms

Algorithms, Performance

Keywords

Adaptive Mesh Refinement, Parallel Computing, Combinatorial Algorithms

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. UCRL-JC-151791.

1. INTRODUCTION

In many important science and engineering simulation problems, key solution features occur only in localized regions of the computational domain. Adaptive mesh refinement (AMR) is an important tool for dynamically increasing spatial and temporal grid resolution where it is needed most to resolve local features. By focusing memory usage and computational effort, a highly resolved solution may be achieved more efficiently than if the grid is refined globally.

Structured adaptive mesh refinement (SAMR) is a particular adaptive gridding methodology in which a dynamic, locally-refined grid is implemented using structured grid concepts. Like other AMR approaches, SAMR presents complications for parallel computing that are absent in uniform grid calculations. Specifically, the need to move data on an irregular locally-refined grid configuration presents complex data communication patterns. Moreover, since the grid may be adapted frequently, the cost of computing grid-dependent data exchange information cannot be amortized over an entire calculation.

Our work uses the SAMRAI framework [19], an object-oriented software library we have developed at Lawrence Livermore National Laboratory. SAMRAI provides robust parallel adaptive gridding and data management capabilities as well as a flexible algorithm support framework that simplifies the development of sophisticated multi-physics SAMR applications. A full description SAMRAI and SAMR algorithms is beyond the scope of this paper. Our focus here is on the parallel performance of critical operations in large-scale adaptive computations. We describe operations that potentially become inefficient when problem size increases and execution requires a large number of processors. We also introduce algorithm enhancements that can substantially alleviate these performance problems.

We present results for relatively simple problems that allow us to control and analyze key adaptive aspects of the calculations. However, these problems use algorithms similar to those found in more complex multi-physics applications [19]. While our findings are the result of codes that use SAMRAI, the main performance issues and algorithm solutions we propose are applicable to other SAMR efforts.

The paper begins with a description of the salient features of SAMR grid structures, grid adaptivity, and manipulation of data during an adaptive computation. Then, we discuss benchmark problems that we use to compare performance of our earlier code implementation to new algorithm devel-

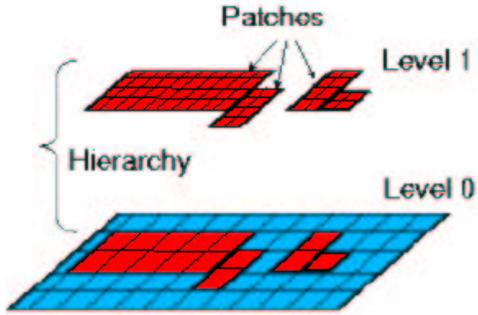


Figure 1: A simple two-level SAMR grid hierarchy, with a refinement ratio of (2,2).

opments. Next, we summarize parallel scaling problems observed in previous work and provide a detailed discussion of algorithmic changes to resolve these problems. Finally, we present parallel performance results to demonstrate performance improvements that the new algorithms provide.

2. SAMR OVERVIEW

Structured adaptive mesh refinement (SAMR) is a technique for increasing local spatial and temporal resolution in numerical simulations of scientific and engineering problems. SAMR was originally developed to achieve higher resolution shock calculations [5, 6]. Subsequent developments have expanded the SAMR algorithm space and the range of problems to which SAMR is applied, including incompressible flow [1, 21]; ALE hydrodynamics [2]; particle-continuum hybrids [17, 27]; flow in porous media [20]; solid mechanics [16, 26]; magnetohydrodynamics [4, 12, 15]; laser-plasma instabilities [13]; and astrophysics [8, 9].

2.1 The SAMR Grid Hierarchy

In the SAMR paradigm, a computational grid is implemented using structured grid components. The grid is a hierarchy of levels of spatial (and often temporal) resolution where all grid cells on a level have the same grid spacing. Also, levels are nested; that is, the coarsest level covers the entire computational domain and each successively finer level covers a portion of the interior of the next coarser level. Computational cells on each level are grouped into non-overlapping, logically-rectangular “box” regions called *patches* (see Fig. 1). Usually, simulation data are stored on patches in contiguous arrays that map directly to the grid cells without indirection. In SAMR parlance, the term *box*, which refers to a logically-rectangular region of grid index space, is ubiquitous. Each patch is defined by its bounding box and numerical routines and data communication are operations on patch data in box regions.

During initial construction and adaptive gridding of an SAMR grid hierarchy, levels are generated one at a time. The coarsest hierarchy level defines the physical extent of the computational domain. Each finer level is constructed by selecting cells on the next coarser level that require refinement based on some problem-based criteria. Then, the cells are clustered into boxes to form patches for the finer level as shown in Figure 2. The grid spacing on each finer level is given by a *refinement ratio* that indicates the num-

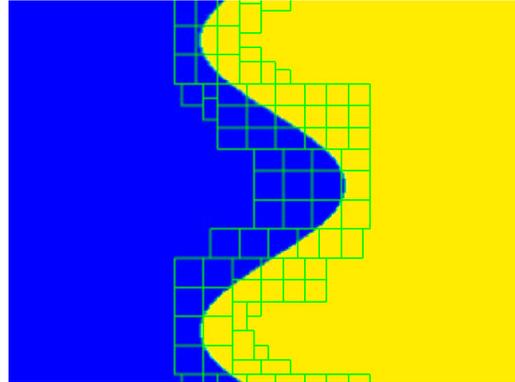


Figure 2: Cross-section of an adaptive grid hierarchy with finer levels covering a discontinuity in the solution. Boxes indicate patch boundaries on the fine level. Note that many fine patches may be needed to resolve features that are not aligned with grid coordinate directions.

ber of fine grid increments in each coordinate direction into which each coarse cell is divided. An important consequence of this organization is that fine *patch* boundaries align with coarse *cell* boundaries. This property facilitates data communication between levels and also the implementation of numerical methods on the grid hierarchy.

2.2 SAMR Algorithms

In this section, we describe common SAMR operations emphasizing those that are central to adaptive gridding; this will fix ideas for more detailed algorithm discussion that follows. SAMR applications can be decomposed into numerical integration operations and gridding operations. In particular, serial numerical routines operate on data living on patches and communication operations pass information between patches, for example, to fill *ghost cells*. SAMR computations are more complicated than those using static uniform grids since the SAMR grid changes dynamically. Also, solution procedures must account for varying levels of grid resolution by identifying cells to refine and properly treating boundaries between grid levels to generate a consistent solution.

We are interested in solving time-dependent partial differential equations (PDEs). In this case, SAMR time integration usually involves interleaving time steps on individual hierarchy levels [5, 19, 26]. When integrating a single level, serial numerical methods operate on patches to advance the solution of the PDE. Data is moved to patch ghost cells from nearby patch interiors before the numerical operations occur. Each ghost cell value comes from one of three sources: a value copied from a neighboring patch on the same grid level, interpolation of data from coarser or finer levels, or a physical boundary condition. Since patch configurations are dynamic, data source and destination information must be recomputed whenever the grid changes.

Levels change at certain points during the time integration sequence and each regrid phase may involve one or multiple levels. Level regridding involves four major steps: choosing cells on the next coarser level that require refinement (i.e., *cell-tagging*), constructing new patch regions, moving data from the old patch configuration to the new configuration,

and generating new data dependency information required to continue integration of the PDEs. Cell tagging is a serial operation and is entirely dependent on the problem being solved and so will not be discussed further here. Our focus in this paper is the construction of new grid patches and the computation of data transfer information.

In SAMRAI, the construction of patches from tagged cells involves several steps. First, tagged cells are covered by a set of boxes using the point clustering algorithm of Berger and Rigoutsos [7], which is common in the SAMR community. Second, these boxes are grown or shrunk to simplify setting physical boundary values in user code and to ensure properties like nesting of the level within the interior of the next coarser level. Third, the resulting boxes are load balanced and assigned to processors. This involves cutting boxes into smaller boxes until the estimated computational work on each resulting patch is equal to or less than an average per-processor workload. To increase the likelihood that neighboring patches will reside on the same processor, the boxes are then ordered according to their spatial location by placing a Morton space filling curve through the box centroids [18]. Finally, boxes are mapped to processors using a greedy bin-packing procedure.

Each processor constructs the patches and associated data for the boxes assigned to it. Note that we assign each patch and its data to one processor, rather than partitioning the data on a patch and distributing it to multiple processors. After making new patches, information describing the transfer of data to and from each patch in the hierarchy affected by the new level configuration must be regenerated. This requires the creation of a new *communication schedule* for each data communication phase that depends on the level.

A communication schedule maintains a list of data transactions between patches on the SAMR grid hierarchy. For example, setting ghost cell values for a patch level may require data interpolated from coarser levels or data copied from neighboring patches on the same level. Based on the hierarchy patch configuration and the data quantities and operations (e.g., interpolation) involved, the communication schedule creates and stores transactions describing the movement of data to and from each patch local to its processor. Each transaction contains the data destination region, the source patch that will supply this data, and any other necessary operator information. When a schedule is executed, each transaction performs either a local data copy or an interprocessor communication. For data movement between processors, a schedule assembles buffers that are exchanged via asynchronous MPI message passing calls. The buffers are formed to maintain a single buffer send and receive between each processor pair regardless of the number of patches or number of data quantities involved. The parallel data decomposition in SAMRAI is similar to that used in other SAMR support libraries. SAMRAI communication schedules are generalizations and extensions of ideas found in the KeLP library [3, 14]. SAMRAI communication schedules in are described in more detail elsewhere [19, 28].

This overview of primary adaptive and data communication operations performed in SAMR calculations serves as background to the results that follow. Next, we concentrate on performance of dynamic patch and schedule construction operations that are crucial to achieve scalable adaptive applications. We will provide additional details for the Berger-Rigoutsos and schedule construction algorithms as needed to

describe our recent algorithmic developments.

3. PARALLEL BENCHMARK PROBLEMS

In this section, we describe two problems that we use to evaluate parallel performance and compare changes to SAMRAI code implementation. Both problems employ a standard SAMR explicit time-stepping algorithm [5], involving spatial and temporal grid refinement. Adaptive gridding occurs every other time step on each hierarchy level. Each problem was chosen to emphasize a different notion of parallel scaling. The first problem uses the same problem size on all processor partitions, which allows us to investigate performance when more processors are applied to a fixed size problem. The size of the second problem is scaled proportionately with number of processors, thus revealing trends that appear as problem size is increased commensurate with the number of processors.

3.1 Non-scaled Problem

The first problem models a Sedov [25] spherical blast wave using the Euler equations of compressible gas dynamics. The solution is represented by five grid variables and is solved using a second-order Godunov shock capturing scheme [10, 11, 22]. The initial condition is a spherical pressure discontinuity of radius δr (we use $\delta r = 0.04375$) at the center of the domain in an otherwise homogeneous medium. The domain is a $1.0 \times 1.0 \times 1.0$ cube on which we apply a Cartesian grid domain with four grid levels with a refinement factor of 4 between successive levels; see Figure 3. Figure 4 shows the number of computational cells on each level as a function of simulation time. Note that the total problem size grows roughly linearly as the simulation advances, which is due to adaptivity of the finest grid to resolve the spherically-expanding shock.

For this problem, the same domain size and SAMR hierarchy level configuration was used on all processor partitions. In the performance results section, we examine the relative costs of time integration and adaptive gridding operations. Gridding operations include cell tagging, patch level construction (including Berger-Rigoutsos point clustering), data redistribution, and generation of communication schedules. The simulation is run to a time of 0.0035, which is about 13 coarse grid time steps. Each hierarchy level is regridded every two time advance steps.

3.2 Scaled Problem

The second problem models a sinusoidal-shaped advecting front using the scalar linear advection equation. The solution state is a single scalar grid variable and we solve the problem using the same integration methods as in the first problem. The problem domain is of size $2.0 \times 1.0 \times 1.0$ with three grid levels, where the mesh is refined by a factor of four between levels; see Figure 5.

For this problem, we control the gridding process to increase the problem size proportionately with the number of processors. That is, we manually scale the grid system from a reference grid configuration to increase the number of grid cells by a specified amount. First, we run a problem on P processors and, after each gridding step, store the boxes that define the grid in a file. Then, when we run on more processors, we read the boxes from the file and refine each box by a factor proportional to the increase in number of processors. For example, to go from P to $2P$ processors, we

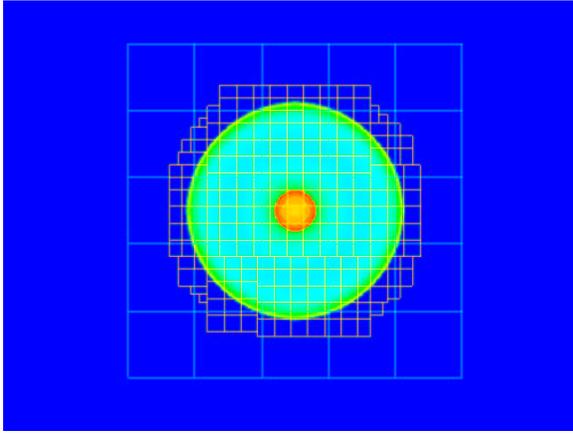


Figure 3: Pressure contours of Sedov spherical shock problem. The adaptive grid system uses four levels. Patch boxes on the finest two levels are shown.

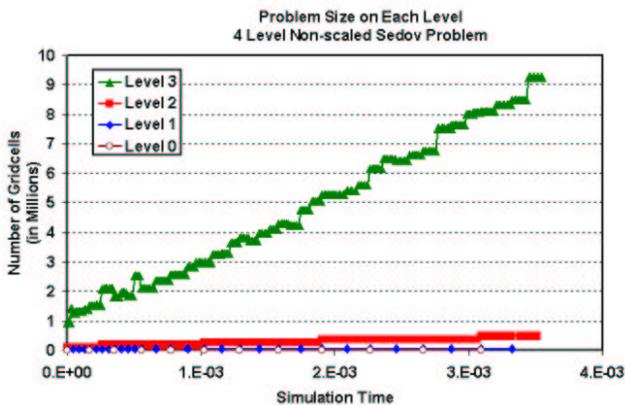


Figure 4: The number of grid cells grows during the simulation as the shock front expands. Note that most cells are on the finest level. Thus, the bulk of computational effort is used to resolve the shock.

double the number of cells in one coordinate direction.

The linear advection problem is very simple numerically, yet produces complex adaptive grid configurations due to the sinusoidal front. Thus, computing communication schedules is non-trivial. Since the problem is linear, we force the same time-stepping sequence on all grid configurations to insure that the same number of integration and communication procedures are performed in each case. Because the amount of numerical work is very small, data communication and adaptive gridding operations are relatively much more expensive in this case than in the Euler case. Therefore, this problem emphasizes adaptive gridding costs and is a good benchmark to assess scalability in the adaptive gridding operations. As a consequence of the way we construct the grids for this case, Berger-Rigoutsos point clustering is not needed. Thus, the adaptive gridding operations counted in the performance results section include only patch level construction (from pre-defined boxes) and generation of communication schedules. The simulation is run to a time of 0.6, which is about 25 coarse grid time steps. Like the non-scaled case, each hierarchy level is regridded

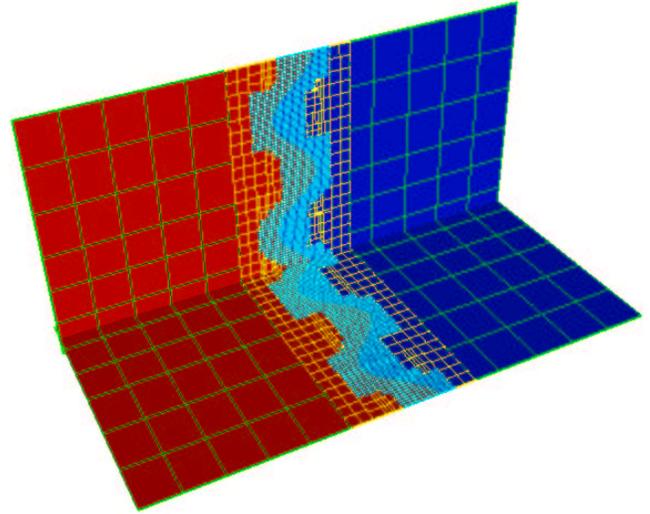


Figure 5: Scaled advecting sinusoidal front problem - density contours overlaid on adaptive grid.

every two time advance steps.

4. ADAPTIVE SCALING PROBLEMS

As discussed in Section 2, the two main phases of SAMR calculations that we consider are adaptive gridding and time integration. Previously, we reported [28] poor parallel scaling in the adaptive gridding parts of benchmark problems described in Section 3. The primary sources of inefficiency were our parallel implementation of the Berger-Rigoutsos point clustering algorithm and the generation of communication schedules. We observed that the total parallel processing time of these operations actually *increased* as the number of processors increased. Incidentally, one may think that data redistribution, whereby data is migrated from the old grid configuration to the new configuration, is a primary contributor to parallel inefficiency. However, the data communication support in SAMRAI is very fast, and we found that data redistribution required less than 1% of total execution time and scaled well on the problems we tested.

In our original analysis, we found that communication schedule construction was the first major scaling problem. Schedule construction cost grew roughly as the square of the number of patches in the problem. As a result, the time needed for these operations rapidly became a dominant cost when calculations involved many patches, which occurs when running larger problems on many processors. The second major scaling problem involved the Berger-Rigoutsos algorithm. In our initial parallel implementation, parallel array all-reduce operations were used over the irregular grid structure to build global tag histogram arrays on each processor at each step of the calculation. Each processor then performed identical operations to construct box regions that covered the tagged cells on the grid. The cost of the algorithm was negligible on small numbers of processors where the cost of global all-reduce operations was barely noticeable. However, as the number of processors was increased this implementation of Berger-Rigoutsos faced two scaling problems. First, as the problem size was increased, the number of global all-reduce operations increased proportionally.

Second, on a greater number of processors, the cost of each all-reduce operation also increased. More details on these issues as well as our solutions to these problems are discussed in later sections.

5. MODIFIED ALGORITHMS

In this section, we discuss changes to communication schedule construction and Berger-Rigoutsos point clustering operations in SAMRAI to alleviate performance problems described above. We consider these procedures computational overhead since they are needed for adaptive gridding but they are not directly part of the numerical solution process. Thus, our goal is to make them as efficient as we can, especially as we scale problem sizes up and run on many processors. In earlier analysis, we observed that our implementation became more costly as more patches were applied to the problem. Thus, we explored ways to generate fewer large patches rather than many smaller patches. However, we found that we can achieve better load balance when using many smaller patches since this allows finer grained control over the distribution of work across processors. Consequently, we chose to develop new approaches that scale well to large numbers of boxes to allow more load balancing flexibility without incurring unacceptable adaptive gridding overhead.

5.1 Communication Schedules

Our original communication schedule construction procedures involved excessive comparisons of patch box regions. The *box intersection* information needed to characterize the transfer of data between patches was generated using an $O(N^2)$ algorithm, where N is the number of patches involved. This approach compares each box on a destination level with every box on potential source levels. Our previous results showed that this does not require excessive execution time when N is sufficiently small. However, N tends to grow both with problem size and number of processors. When N becomes large enough, the schedule construction process begins to dominate total execution time due to the quadratic nature of the algorithm.

We explored three improved box intersection algorithms in our work. Each algorithm requires a setup phase, during which data structures based on spatial relationships between boxes are constructed. The setup is a one-time cost, (typically $O(N \log N)$), that can be amortized over many calls to compute box intersections depending on how the adaptive grid changes. For the sake of brevity, we discuss only our best performing algorithm here, which uses a data structure that we call a *Recursive Binary Box Tree* (RBBT).

The box intersection operations used in SAMR are similar to operations studied in other fields, e.g., computational geometry. We refer readers to Samet [23, 24], for general information and additional references. Although our RBBT structure is similar to other hierarchical data structures, such as octrees, k-d trees, etc, we believe it is novel in that it can be used to determine spatial relationships among boxes in an arbitrary number of spatial dimensions or other geometric shapes.

5.1.1 RBBT algorithms

A *Recursive Binary Box Tree* (RBBT) is a data structure that enables the efficient (typically $O(N \log N)$) computation of spatial relationships. The exact nature of an RBBT

is best explained by examining the **ConstructBoxTree** algorithm, which is invoked during the setup phase.

The root node of a binary tree is passed a set of boxes, for which a bounding box is computed. In 3D, the bounding box is then conceptually divided into two halves by constructing a bisecting plane that is parallel to the z-axis. The list of boxes is then partitioned into three subsets. The first subset contains the boxes that intersect the bisecting plane; this subset of boxes is assigned to the root node. The second subset contains the boxes that are entirely enclosed in one half of the bounding box; this subset of boxes is passed (in a recursive call) to the constructor for the left child node. The third subset contains boxes enclosed by the remaining half, and is passed to the constructor for the right child node. The tree that results from this process is called the *primary* tree.

Next, each node in the primary tree constructs a private, *secondary* binary tree. The constructor for the root node of the secondary tree is passed the set of boxes that was assigned to the node in the primary tree. The secondary tree's construction is identical to the primary tree, with the exception that the bounding box is divided by a plane that is parallel to the y-axis. In a similar manner, each node in the secondary tree constructs a *tertiary* tree, in which bounding boxes are divided by planes that are parallel to the x-axis.

ConstructBoxTree(boxlist B , dimension d)

1. **# Construct bounding box.**
2. Compute the bounding box for B .
3. Divide the bounding box into 2 quadrants; $j = 1$ or 2.
4. If $d = 1$, divide the box in a direction orthogonal to the x-axis
5. If $d = 2$, divide the box in a direction orthogonal to the y-axis
6. If $d = 3$, divide the box in a direction orthogonal to the z-axis
7. **# Partition boxlist B .**
8. For each box $b \in B$
9. If b is enclosed by quadrant j
10. Insert b in $boxlist_j$
11. Else
12. Insert b in this node's boxlist
13. **# Recursively construct child nodes.**
14. For $j = 1$ to 2
15. If $boxlist_j$ is empty
16. Set $child_j$ to null
17. Else
18. $child_j = \text{ConstructBoxTree}(boxlist_j, d)$
19. **# Construct private BoxTree.**
20. If this node's boxlist contains more than one box, and $d > 1$
21. $private_tree = \text{ConstructBoxTree}(boxlist, d - 1)$
22. Set $boxlist$ to null
23. Else
24. Set $private_tree$ to null

The RBBT can be "tuned" to reduce memory requirements and improve performance (i.e. reduce execution time) through the use of constraints that, for clarity, we have omitted from the above. One constraint is: if the boxlist that is passed to a node contains less than some specified number of boxes n , assign all the boxes to the node and return immediately (i.e., do not recurse to build child nodes and/or private trees). If n is relatively large, nodes will have larger

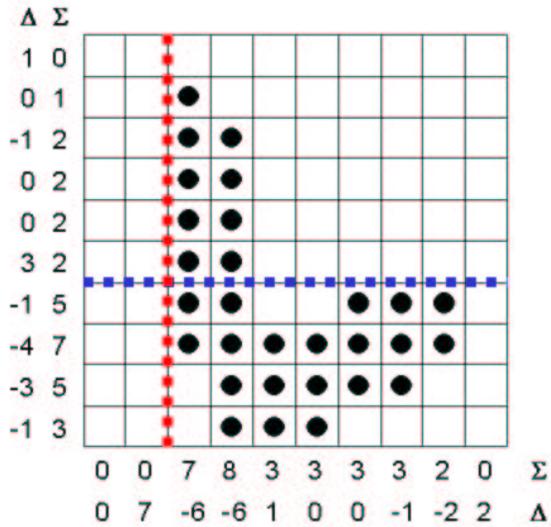


Figure 6: Potential box cut locations at inflection points in histogram data used in the BR algorithm. Dark circles indicate tagged cells. Σ is the histogram value indicating the number of tags in the perpendicular direction, and Δ is the discrete Laplacian. Potential inflection cut points occur along each direction at the largest sign change in Δ (dotted lines).

numbers of boxes in their boxlists (after partitioning), and the tree will have fewer nodes. If $n = 1$, a node’s boxlist will contain at most a single box, and the number of nodes in the tree will be maximized. In our experience, best performance was obtained with values around $n = 10$.

In general, the performance of binary tree based algorithms is known to degrade when the trees are unbalanced. With regard to RBBTs, performance can be shown to degrade if, at the conclusion of Step 12, the number of boxes in the three subsets is relatively unequal. In the worst case, one of the child subsets could be empty, which would result in an unbalanced tree. This issue can be addressed by incorporating a strategy whereby the “bisecting plane” is adjusted to achieve better balance during partitioning. Several strategies are possible here, but for the sake of brevity will not be discussed further.

The RBBT is used to perform a box intersection operation by “walking the tree” in a recursive manner. The **FindIntersectingBoxes** algorithm is based on the following necessary condition: a box \mathbf{b} can intersect a box contained in the subtree that is rooted at a node only if \mathbf{b} intersects the node’s bounding box.

The **BoxIntersection** algorithm operates as follows. Given an arbitrary box of interest \mathbf{b} , we want to find the subset of boxes from an RBBT that intersect with \mathbf{b} . Beginning at the RBBT’s root node, \mathbf{b} is compared with the node’s bounding box. If the two do not intersect, the call returns. If they do intersect, then \mathbf{b} is compared to each box in the node’s boxlist, and any boxes that intersect \mathbf{b} are added to the output. Next, \mathbf{b} is passed recursively to the left and right child nodes, and to the root node of the private RBBT.

5.1.2 Comparison and analysis

For simple configurations of boxes, e.g. a set of identically sized, regularly tiled boxes in an $N^{1/3} \times N^{1/3} \times N^{1/3}$ configuration, it is easy to show that RBBT construction is bounded by $O(N \log N)$, and that a single box intersection operation is bounded by $O(\log N)$. Analytic bounds for arbitrary collections of boxes have so far eluded us, however, we conjecture that, with some variation on the tree-balancing strategy discussed above, and the constraint (which is common in SAMR technologies) that the set of boxes from which the RBBT was constructed are non overlapping, these bounds can be shown to hold.

5.2 Berger-Rigoutsos Algorithm

One of the key procedures that we use to generate a new patch level from tagged cells is the point clustering algorithm of Berger and Rigoutsos (BR) [7]. This algorithm groups tagged cells into an initial collection of logically-rectangular box regions from which patches will be created. This algorithm is commonly used in the SAMR community. Here, we provide a simplified description of our implementation written as a recursive procedure:

BR_Cluster(box box_{in} , boxlist $boxes_{out}$)

1. Compute tag histogram for each dimension of box_{in} .
2. Set n_{tags} = number of tagged cells in box_{in} .
3. If $n_{tags} > 0$
4. Set $bbox_{tag}$ = smallest bounding box for box_{in} tags
5. If $n_{tags} / (\text{num cells in } bbox_{tag}) < tol_{eff}$
6. Loop over sides of $bbox_{tag}$, long to short
7. If \exists zero histogram value near middle of side
8. Set x_{cut} = assoc. cell index, exit loop
9. Elseif \exists inflection point in histogram
10. Set x_{cut} = assoc. cell index, exit loop
11. If x_{cut} set
12. Cut $bbox_{tag}$ into box_{left}, box_{right} at x_{cut}
13. **BR_Cluster**($box_{left}, boxes_{left}$)
14. **BR_Cluster**($box_{right}, boxes_{right}$)
15. If $boxes_{left}, boxes_{right}$ obey $tol_{combine}$
16. Append $boxes_{left}, boxes_{right}$ to $boxes_{out}$
17. If $boxes_{out} = \emptyset$, add $bbox_{tag}$ to $boxes_{out}$
18. Else set $boxes_{out} = \emptyset$

A bounding box for all tagged cells is passed in to the initial call to the routine; i.e., box_{in} . The routine outputs a list of non-overlapping boxes that covers the tags, $boxes_{out}$. The routine is called recursively until the fraction of tagged cells in the given box is greater than some user-prescribed tolerance (line 5) and the total number of cells in the set of boxes into which a box is cut is less than some fraction of cells in the original box (line 16). Figure 6 illustrates how tag histogram data is used to choose cut points for the box.

The main operation that we are concerned with regarding parallel performance is the tag data histogram computation step (line 1). Recall that patches in SAMRAI, and hence tag data, are distributed across processors. The main parallel inefficiency in our original BR implementation arose from the fact that we accumulated tag information into identical signature arrays on each processor using global MPI all-reduce operations. Then, identical operations on the histogram data were performed on each processor. On small numbers of processors, the cost of this implementation is acceptable. However, as problems are scaled up to run on large

Table 1: Breakdown of processor participation for a 5-step Euler sphere computation on 128 processors. There were a total of 588 calls to the BR algorithm.

Recursion Level	Participating processors	Percent of total calls
11	2	1.4
10	4	7.1
9	8	17
8	8	30.6
7	8	46.3
6	8	61.9
5	18	74.5
4	32	84.7
3	80	91.5
2	80	95.6
1	100	98
0	125	100

numbers of processors, algorithm performance degrades considerably for three reasons. First, the amount of histogram data in each reduction step increases as the length of the arrays grow with the number of cells in the problem. Second, the number of global reductions increases as we consider more box regions. Third, the cost of each global reduction is larger on more processors (theoretically $O(P \log P)$ but is in practice implementation-dependent).

The BR algorithm starts by requiring knowledge of all tagged cells which may cover a large portion of the domain and involve many patches. However, our analysis revealed that as the recursion level increases, the box sizes and number of tags evaluated by the algorithm quickly decrease. Thus, the number of processors *needed* to construct histogram information quickly diminishes. Table 1 shows how the number of participating processors changes in the BR algorithm with each recursion level in a sample problem using 128 processors. The relevant tag data for most calls resides on a small subset of processors. The routine was called a total of 588 times and recursion level 5 data reveals that 74.5% of all calls required 18 or fewer processors.

These findings indicate global reductions are unnecessary for most histogram construction operation in the BR algorithm. To achieve a more scalable approach, we restructured the parallel implementation by replacing global reductions with collective communication operations involving only processors needed to participate. The reformulated BR implementation uses a *binary tree reduction* approach to determine participating processors. A designated *root* processor manages the processors that participate and construction of histogram information.

The new implementation is similar to the **BR_Cluster** routine above with a few changes. The primary difference occurs at the tag histogram computation in line 1. In the new implementation, if this processor is not the root processor and owns no patch that intersects box_{in} , the routine returns. Otherwise, only the participating processors are used to compute tag histograms for box_{in} . These histograms are assembled on the root processor using an all-to-one reduction operation. Also, only the root processor performs the operations involving the location of box cut points and

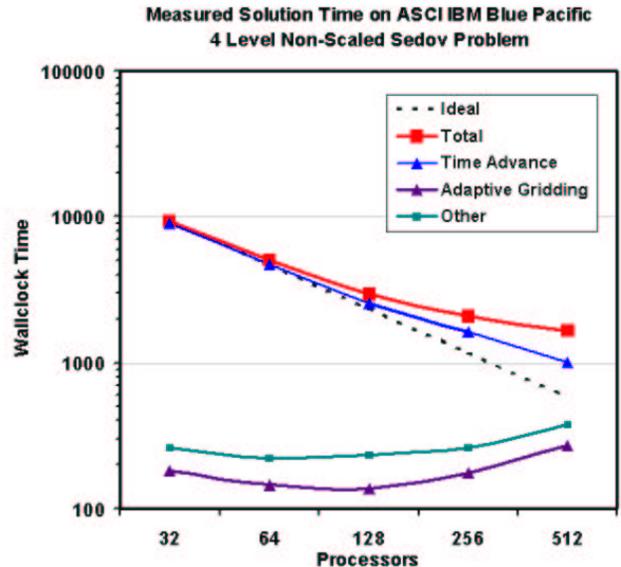


Figure 7: Wallclock time measurements for the non-scaled four level spherical shock problem run on IBM Blue Pacific (data from Table 3).

cutting boxes. In the box cutting step at line 13, the root processor broadcasts box_{left}, box_{right} to all processors participating in the current recursion level. Then, recursive calls (lines 14, 15) are made on the participating processors only. Finally, at the end of the recursion, the root processor broadcasts $boxes_{out}$ to *all* processors.

We note that we first used MPI communicators for the all-to-one reductions but later found that hand-coded MPI send-recvs operations performed slightly better. We postulate that this may arise from the need to synchronize all processors each time an MPI communicator is formed.

6. PERFORMANCE RESULTS

In this section, we we assess performance of the new algorithms on the benchmark problems described in Section 3 on two parallel computer systems. The first is the ASCI IBM Blue Pacific system constructed of 256 four processor SMP nodes (244 of which are available for typical batch runs), each with 1.5GB memory and 332 MHz PowerPC 604e processors. The interconnect network is switch-based, with an omega topology supporting up to 150Mbytes/s bi-directional bandwidth between nodes. The second system is LLNL’s M&IC MCR Linux cluster system with 1152 two processor nodes, each with 4 GB memory and 2.4 Ghz Intel processors. Its interconnect is Quadrics QsNet Elan3 with 300 Mbytes/s MPI bandwidth and $< 5\mu s$ latency.

Both benchmark problems are timed on a range of processors, from 32 to 512 on ASCI Blue Pacific and from 32 to 1024 on the MCR Linux system. To eliminate possible inconsistencies from machine load conditions, each of the cases was run three times and the times reported are an average of the three. The overall times are decomposed into two general phases; *time advance* and *adaptive gridding*. Time advance includes numerical kernel computations on patches, including costs resulting from load imbalances, as

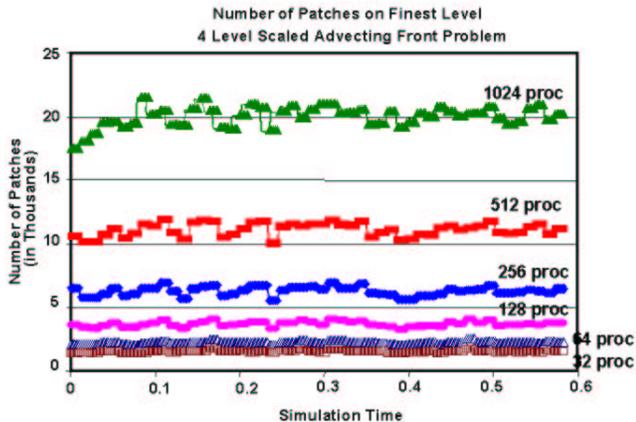


Figure 8: Number of patches on the finest level on various processors for the scaled advecting front calculation.

well as communication required to exchange data between patches. Adaptive gridding includes the cost of performing the Berger-Rigoutsos clustering to build a new level, constructing communication schedules, and distributing data from the old grid configuration to the new. All other operations are classified as “other”, which includes such operations as level initialization, load balancing, etc. We show timings with our original implementation and the new implementation, which uses the RBBT algorithm in the construction of communication schedules and the parallel binary tree implementation of Berger-Rigoutsos clustering.

6.1 Non-scaled Problem Performance

Timing results for the non-scaled four level adaptive Sedov spherical shock benchmark, discussed in Section 3.1, are shown in Tables 2 and 3 for the MCR Linux and ASCI IBM systems, respectively.

Adaptive gridding operations are clearly much more efficient using the new algorithms. Significant cost savings is seen in both the communication schedule construction and the Berger-Rigoutsos clustering phases of adaptive gridding operations. Communication schedule construction cost is reduced by a factor of 2 to 5. On the ASCI IBM system, the new Berger-Rigoutsos clustering operation is significantly faster on larger processor partitions. In our original implementation, this operation constitutes 34% of the total execution time but the new binary tree implementation requires only about 3%. On the Linux MCR system, the Berger-Rigoutsos operation is very fast (4%) in the original implementation so we see only a slight improvement with the new implementation. This system has a very fast and efficient implementation of global reduction operations, so the efficiency enhancements in the new algorithm had less of an effect on reducing the overall time. Nevertheless, the results from the IBM system indicate the modified implementation should be effective on parallel systems where global reductions are relatively expensive; for example, on cluster systems with a slow network.

The overall scaling trends for time advance and adaptive gridding phases using the new algorithms on IBM Blue Pacific are shown in Figure 7. A plot of the data from MCR

shows similar trends. We expect some reduction in parallel efficiency because the problem size was held constant, hence the problem size per processor decreases as the processor count increases. In spite of this, overall parallel scaling is reasonable for this adaptive problem. The costs we designate as “other” start to become more significant on larger numbers of processors. This is predicted by Amdahl’s law, since non-parallel overhead operations begin to constitute a larger proportion of the total computational work.

6.2 Scaled Problem Performance

Timing results for the scaled three level adaptive linear advection benchmark, discussed in Section ??, are shown in Tables 4 and 5 on the MCR Linux and ASCI IBM systems, respectively. Unlike the spherical shock calculation discussed above, for which the same global problem size is run across all processors, the problem size in this case is scaled with the number of processors so the number of grid cells per processor remains roughly constant.

The cost of the time advance portion of the calculation scales well. Also, the communication costs associated with refine and coarsen operations and in passing data from an old to a new hierarchy level both scale reasonably well. Recall that the Berger-Rigoutsos operation is not performed in this case; see Section 3.2. The primary operation that scales poorly in the original implementation is construction of communication schedules. Recall that the cost of constructing communication schedules in our original implementation grows as $O(N^2)$ with the number of patches; see Section 5.1. The number of patches used in the calculation is roughly proportional to the problem size and hence grows with the number of processors. Figure 8 shows the number of patches on the finest level for the various processor partitions over the course of the adaptive simulation. As the number of processors is doubled, the number of patches on the level also roughly doubles. Thus, there is a significant increase in schedule construction cost in our original implementation as the problem is scaled to run on larger numbers of processors. Use of the RBBT algorithm in the new implementation significantly reduces this cost.

Figure 9 compares the scaling characteristics using our original implementation to the new implementation. The new implementation clearly has much better scaling characteristics.

7. CONCLUDING REMARKS

This paper investigates algorithms to enhance the scaling efficiency of adaptive gridding operations in structured adaptive mesh refinement (SAMR) applications. Although our evaluation focuses on their implementation in the SAMRAI library, the algorithms are applicable to SAMR applications in general. Earlier tests revealed that adaptive gridding operations can become very expensive on large numbers of processors. We introduce a new Recursive Binary Box Tree (RBBT) algorithm and an alternative parallel implementation of the commonly used Berger-Rigoutsos point clustering algorithm to improve adaptive gridding efficiency. We then evaluate the performance of two adaptive benchmarks, scaled and non-scaled, on two parallel systems, reporting a breakdown of the scaling characteristics of the various parts of the adaptive calculation.

The RBBT is a data structure that enables efficient computation of spatial relationships. We apply it in the con-

Table 2: Timing results on the Linux MCR system of the non-scaled spherical shock problem. The wallclock time, and percentage of total wallclock time, are shown for the different phases of the calculation.

Processors	32		64		128		256		512		1024	
<i>Original Implementation</i>												
Time Advance												
Computation - num kernels	1647.0	87%	845.3	84%	451.0	80%	260.7	73%	160.1	59%	111.4	49%
Communication overhead	101.0	5%	57.8	6%	36.7	7%	25.0	7%	18.2	7%	12.8	6%
Adaptive Gridding												
Schedule construction	72.3	4%	49.1	5%	37.7	7%	34.3	10%	37.0	14%	40.9	18%
Berger Rigoutsos	4.1	1%	4.4	1%	4.8	1%	6.1	2%	9.5	4%	12.7	6%
Data re-distribution	6.4	1%	3.4	1%	1.9	0%	1.1	0%	0.9	0%	0.8	0%
Other	43.1	2%	32.4	3%	31.5	5%	31.7	8%	45.5	16%	50.7	21%
Total	1873.9		992.5		563.6		358.9		271.2		229.3	
<i>New Implementation</i>												
Time Advance												
Computation - num kernels	1603.6	90%	848.1	89%	455.3	84%	260.8	78%	159.0	66%	111.0	56%
Communication overhead	98.8	5%	58.5	6%	36.7	7%	25.3	8%	18.6	8%	13.2	7%
Adaptive Gridding												
Schedule construction	13.2	1%	10.5	1%	10.3	2%	10.5	3%	13.7	6%	17.1	8%
Berger Rigoutsos	3.0	1%	3.2	1%	3.5	1%	5.6	2%	9.0	4%	11.8	6%
Data re-distribution	6.3	1%	2.4	1%	1.9	1%	1.1	0%	0.8	0%	0.8	0%
Other	42.3	2%	32.9	2%	31.6	5%	32.3	9%	44.6	16%	49.7	23%
Total	1767.0		955.6		539.3		335.7		245.5		203.5	

Table 3: Timing results on the IBM Blue Pacific system of the non-scaled spherical shock problem.

Processors	32		64		128		256		512	
<i>Original Implementation</i>										
Time Advance										
Computation - num kernels	8391.5	81%	4369.3	77%	2448.1	68%	1345.8	57%	861.4	29%
Communication overhead	635.4	6%	364.2	6%	236.5	7%	153.2	6%	117.8	4%
Adaptive Gridding										
Schedule construction	945.3	9%	607.3	11%	454.3	13%	375.0	16%	408.3	13%
Berger Rigoutsos	54.7	1%	83.0	2%	104.2	3%	254.3	11%	1025.2	34%
Data re-distribution	47.5	1%	26.8	1%	16.8	1%	11.5	1%	9.5	1%
Other	265.5	2%	216.6	3%	341.0	8%	236.0	9%	592.8	19%
Total	10339.9		5667.2		3600.9		2375.8		3015.1	
<i>New Implementation</i>										
Time Advance										
Computation - num kernels	8363.9	89%	4360.2	86%	2343.0	80%	1473.4	71%	895.1	54%
Communication overhead	622.0	6%	360.1	7%	226.8	8%	153.5	7%	117.5	7%
Adaptive Gridding										
Schedule construction	119.1	1%	92.4	2%	100.7	3%	130.8	6%	213.5	13%
Berger Rigoutsos	18.7	1%	28.2	1%	23.4	1%	39.6	2%	53.8	3%
Data re-distribution	43.8	1%	24.7	1%	13.6	1%	7.7	1%	5.5	1%
Other	262.4	2%	222.5	3%	233.2	7%	262.9	13%	374.9	22%
Total	9429.9		5088.1		2940.8		2067.9		1660.4	

Table 4: Timing results on the Linux MCR system of the scaled advecting front problem. The wallclock time, and percentage of total wallclock time, are shown for the different phases of the calculation.

Processors	32		64		128		256		512		1024	
<i>Original Implementation</i>												
Time Advance												
Computation - num kernels	315.9	79%	314.2	76%	317.6	63%	362.9	45%	357.3	21%	379.0	8%
Communication overhead	35.4	9%	34.8	9%	35.2	7%	40.4	5%	44.6	3%	100.9	2%
Adaptive Gridding												
Schedule construction	30.5	8%	43.3	11%	130.0	26%	380.6	47%	1277.3	75%	4054.3	87%
Data re-distribution	2.7	1%	2.8	1%	3.2	1%	3.6	1%	4.1	1%	5.2	1%
Other	16.0	3%	16.0	3%	20.0	3%	22.8	2%	32.1	1	107.5	2%
Total	400.5		411.2		506.0		810.2		1715.4		4647.0	
<i>New Implementation</i>												
Time Advance												
Computation - num kernels	323.1	84%	329.8	83%	321.9	78%	330.1	77%	334.7	69%	350.1	51%
Communication overhead	35.6	9%	35.3	9%	36.4	9%	39.3	9%	44.5	9%	75.1	11%
Adaptive Gridding												
Schedule construction	6.6	2%	10.9	3%	17.6	4%	31.2	7%	68.1	14%	159.2	23%
Data re-distribution	2.7	1%	2.8	1%	3.1	1%	3.4	1%	3.7	1%	4.0	1%
Other	17.0	4%	19.0	4%	32.3	7%	22.4	6%	33.3	7%	92.5	14%
Total	385.0		397.7		411.4		426.3		484.3		680.9	

Table 5: Timing results on the IBM Blue Pacific system of the scaled advecting front problem.

Processors	32		64		128		256		512	
<i>Original Implementation</i>										
Time Advance										
Computation - num kernels	1182.7	64%	1309.7	56%	1323.5	42%	2144.8	33%	1739.3	12%
Communication overhead	186.8	10%	242.1	10%	227.9	7%	311.8	5%	382.8	3%
Adaptive Gridding										
Schedule construction	384.8	21%	715.0	31%	1467.2	47%	3918.2	60%	11969.6	83%
Data re-distribution	18.1	1%	21.7	1%	25.9	1%	41.7	1%	50.4	1%
Other	66.8	4%	47.6	2%	92.8	3%	115.6	1%	178.8	1%
Total	1839.1		2336.1		3137.4		6532.1		14321.0	
<i>New Implementation</i>										
Time Advance										
Computation - num kernels	1267.3	80%	1287.7	78%	1324.8	74%	2119.3	75%	1572.6	58%
Communication overhead	181.8	12%	192.9	12%	211.7	12%	272.4	10%	333.2	12%
Adaptive Gridding										
Schedule construction	45.3	3%	81.4	5%	149.7	8%	149.7	5%	621.6	23%
Data re-distribution	15.7	1%	17.0	1%	19.5	1%	23.9	1%	17.4	1%
Other	69.3	4%	79.0	4%	93.6	5%	270.5	9%	180.6	6%
Total	1579.3		1658.0		1799.4		2835.7		2725.4	

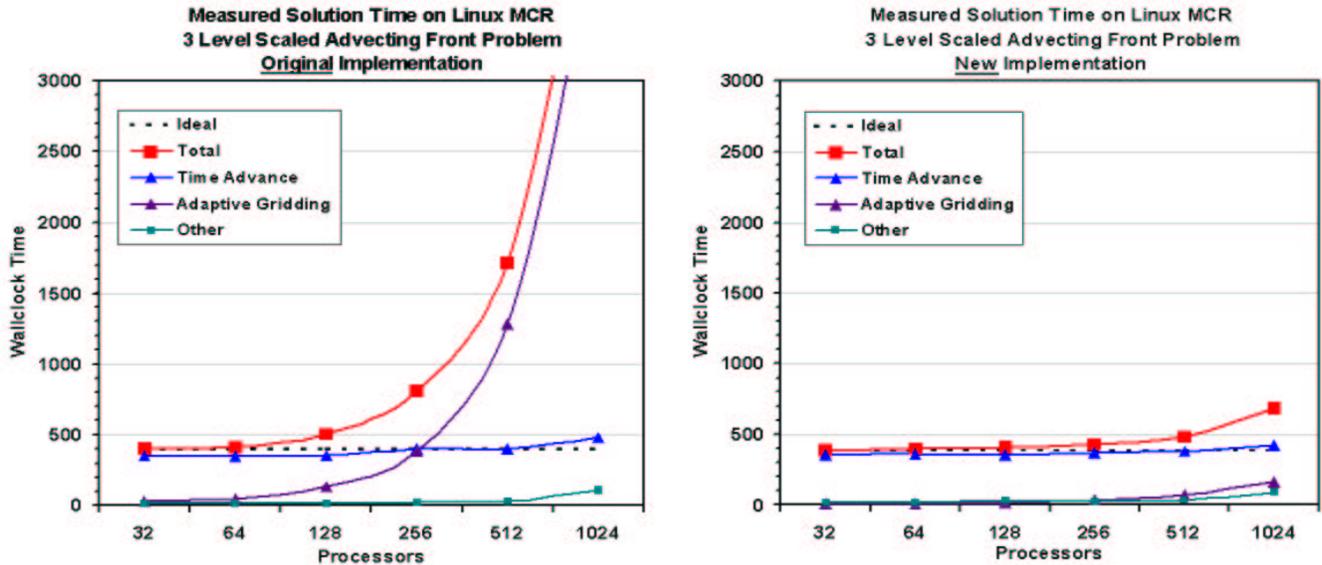


Figure 9: Comparison of scaling qualities of the scaled advecting front problem of our original to our current implementation using the RBBT algorithm (data from Table 4).

struction of communication schedules, which describe the data dependencies between patches. The schedules must be reconstructed each time the adaptive grid changes. The algorithm we originally used in constructing communication schedules was $O(N^2)$ in complexity, where N is the number of patches on the level. Using the RBBT algorithm, we are able to reduce the complexity to $O(N \log N)$. This leads to significant reductions in cost for calculations scaled to 1024 processors.

On one of the systems tested, ASCI IBM Blue Pacific, our original implementation of the Berger-Rigoutsos point clustering algorithm became very costly on large processor configurations. This was due to the use of global reductions in the implementation, which become more expensive as the number of processors increases. We developed an alternative parallel implementation that uses localized sends and receives in place of global reductions and saw significant savings on this system. Improvements were less significant on a high-performance Linux cluster system but we attribute this to the fact that the system has very efficient global reduction operations so our modified approach has less of an effect. We speculate this new implementation will primarily be useful on systems where global reductions are expensive, relative to other communication operations (e.g. commodity cluster systems with slow interconnect).

The scaling studies we perform in this paper with adaptive problems run on up to 1024 processors reveal that the core numerical operations and communication scale well to larger numbers of processors. Although we were able to make significant progress in reducing adaptive gridding costs through the use of more efficient algorithms, further enhancements will be necessary as we push to systems that use even larger numbers of processors.

8. REFERENCES

- [1] A. J. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. A conservative adaptive projection method for variable density incompressible Navier-Stokes equations. *J. Comp. Phys.*, 142:1–46, 1998.
- [2] R. W. Anderson, R. B. Pember, and N. S. Elliott. An arbitrary Lagrangian-Eulerian method with local structured adaptive mesh refinement for modeling shock hydrodynamics. In *40th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, Jan 14–17 2001.
- [3] S. B. Baden, P. Colella, D. Shalit, and B. V. Straalen. Abstract KeLP. In *Tenth SIAM Conf. on Parallel Processing for Scientific Computing*, Portsmouth, VA, March 2001.
- [4] D. S. Balsara. Divergence-free adaptive mesh refinement for magnetohydrodynamics. *J. Comp. Phys.*, 174:614–648, 2001.
- [5] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, 1989.
- [6] M. J. Berger and J. Olinger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–512, 1984.
- [7] M. J. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Trans. on Systems, Man., and Cybernetics*, 21:1278–1286, September 1991.
- [8] G. Bryan and M. L. Norman. Simulation x-ray clusters with adaptive mesh refinement. In *Proc. of the 12th Kingston Meeting on Theoretical Astrophysics: Computational Astrophysics (ASP Conference Series, 123)*, 1997. eds. D. A. Clarke and M. J. West, p. 363.
- [9] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, G. Henry, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Truran, and M. Zingale. High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *Proc. of SC00*, 2000.

- [10] P. Colella. A direct eulerian MUSCL scheme for gas dynamics. *SIAM J. Sci. Stat. Comput.*, 6:104–117, 1985.
- [11] P. Colella and H. M. Glaz. Efficient solution algorithms for the riemann problem for real gases. *J. Comp. Phys.*, 59:264–289, 1985.
- [12] D. L. de Zeeuw, T. L. Gombosi, C. P. T. Groth, K. G. Powell, and Q. F. Stout. An adaptive mhd method for global space weather simulations. *IEEE Trans. Plasma Sci.*, 28:1956–1965, 2000.
- [13] M. R. Dorr, F. X. Garaizar, and J. A. F. Hittinger. Simulation of laser plasma filamentation using adaptive mesh refinement. *J. Comp. Phys.*, 177:233–263, 2002.
- [14] S. J. Fink, S. B. Baden, and S. R. Kohn. Flexible communication schedules for block structured applications. In *Third Int. Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, CA, August 1996.
- [15] R. Friedel, R. Grauer, and C. Marliani. Adaptive mesh refinement for singular current sheets in incompressible magnetohydrodynamics. *J. Comp. Phys.*, 134:190–198, 1997.
- [16] F. X. Garaizar and J. A. Trangenstein. Adaptive mesh refinement and front-tracking for shear bands in an antiplane shear model. *SIAM J. Sci. Comput.*, 20:750–779, 1998.
- [17] A. Garcia, J. Bell, W. Crutchfield, and B. Alder. Adaptive mesh and algorithm refinement using direct simulation Monte Carlo. *J. Comp. Phys.*, 154:134–155, 1999.
- [18] R. Gutman. Use of Morton space-filling curve for load balance. *Dr. Dobb's Journal*, pages 115–121, July 1999.
- [19] R. Hornung and S. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002. See also <http://www.llnl.gov/CASC/SAMRAI>.
- [20] R. D. Hornung and J. A. Trangenstein. Adaptive mesh refinement and multilevel iteration for flow in porous media. *J. Comp. Phys.*, 136:522–545, 1997.
- [21] D. F. Martin and P. Colella. A cell-centered adaptive projection method for the incompressible Euler equations. *J. Comp. Phys.*, 163:271–312, 2000.
- [22] J. Saltzman. An unsplit 3d upwind method for hyperbolic conservation laws. *J. Comp. Phys.*, 115:153, 1994.
- [23] H. Samet. *Applications of spatial data structures*. Addison-Wesley, Menlo Park, CA.
- [24] H. Samet. *Design and analysis of spatial data structures*. Addison-Wesley, Menlo Park, CA.
- [25] L. I. Sedov. *Similarity and Dimensional Methods in Mechanics*. Academic Press, New York, NY, 1959.
- [26] J. A. Trangenstein. Adaptive mesh refinement for wave propagation in nonlinear solids. *SIAM J. Sci. Stat. Comput.*, 16(4):819–839, 1995.
- [27] H. S. Wijesinghe, R. D. Hornung, A. L. Garcia, and N. G. Hadjiconstantinou. 3-dimensional hybrid continuum-atomistic simulations for multiscale hydrodynamics. In *Proc. of ASME International Mechanical Engineering Congress and R and D Expo*, Washington, D. C., November 15–21 2003.
- [28] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott. Large scale parallel structured amr calculations using the samrai framework. In *Proc. of SC01*, Denver, CO, November 10–16 2001.