Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework[*]

Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, Noah Elliott
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
[awissink,hornung,skohn,smith84,elliott22]@llnl.gov

ABSTRACT
This paper discusses the design and performance of the parallel data communication infrastructure in SAMRAI, a software framework for structured adaptive mesh refinement (SAMR) multi-physics applications. We describe requirements of such applications and how SAMRAI abstractions manage complex data communication operations found in them. Parallel performance is characterized for two adaptive problems solving hyperbolic conservation laws on up to 512 processors of the IBM ASCI Blue Pacific system. Results reveal good scaling for numerical and data communication operations but poorer scaling in adaptive meshing and communication schedule construction phases of the calculations. We analyze the costs of these different operations, addressing key concerns for scaling SAMR computations to large numbers of processors, and discuss potential changes to improve our current implementation.

1. INTRODUCTION
Structured adaptive mesh refinement (SAMR) [4, 5] is an effective technique for focusing computational resources in numerical simulations of partial differential equations that span a range of disparate length and time scales [7, 8]. AMR is used to dynamically increase grid resolution locally to resolve important fine-scale features in the solution. The goal is to achieve a more efficient computation than one in which a globally-uniform fine grid is applied. SAMR is a particular brand of adaptive mesh refinement technology in which the locally-refined grid is defined with structured grid components. Like other dynamic mesh refinement approaches, SAMR presents complications for parallel computing that are absent in uniform grid calculations. The complexity of data communication arises from the need to transfer data between grid regions of differing resolution on irregular locally-refined grid configurations. Since grid generation may be performed frequently, the complexity of computing grid-dependent data exchange information cannot be amortized over an entire calculation. Also, substantial data transfers may occur as the grid is refined and coarsened.

The SAMRAI (Structured Adaptive Mesh Refinement Application Infrastructure) [16, 15] library was developed to support a wide range of parallel multi-physics SAMR applications. It provides general adaptive meshing and data management capabilities as well as a flexible algorithm development framework. SAMRAI is similar in spirit to other libraries that support SAMR computations [3, 18]. However, it provides novel tools to manage the complexity algorithms and data management in parallel multi-physics applications.

This paper begins by discussing characteristics of some applications built using SAMRAI and how the needs of such problems influenced the design of the framework. Then, we describe the parallel data communication infrastructure in the library. Lastly, we investigate the performance of adaptive calculations using this infrastructure. We provide a breakdown of computational costs in example calculations run on up to 512 processors of ASCI IBM Blue Pacific, and draw conclusions about overhead, load imbalances, and communication costs.

2. BACKGROUND
The basic features of the SAMR approach are rooted in the work of Berger, Oliger, and Colella [4, 5]. The computational grid consists of a collection of structured grid components, organized into a hierarchy of nested levels of spatial (and often temporal) grid resolution. Each level in the hierarchy represents a domain with uniform grid spacing. The domain on each level is expressed as a disjoint union of logically-rectangular “patch” regions. A level is defined by selecting cells for refinement on the next coarser level in the hierarchy and clustering these cells into a new set of patches. The selection of cells to refine depends on the needs of each computation.

SAMR offers potentially large savings in memory and computational effort when compared to globally-uniform static mesh calculations. However, difficulties associated with its implementation often make the application of SAMR prohibitive. Apart from the development of numerical methods for locally-refined grids, the complexity of data management is a fundamental hurdle. Data must be exchanged among

[*]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-144755.

©2001 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC2001 November 2001, Denver ©2001 ACM 1-58113-293-X/01/0011 $5.00

irregularly configured patch regions on a single level and between patches on different levels of resolution. These data communication patterns change whenever the grid changes. Data management becomes more complex in multi-physics applications. Such problems typically involve many data quantities with different centerings on the grid (cell-centered, node-centered, etc.), irregular data such as particles, and different solution procedures that share variables and use distinct data communication patterns.

The SAMRAI framework facilitates the development of parallel multi-physics SAMR applications by providing software tools to automatically manage the sort of application complexity described above. A primary design goal of SAMRAI is that the framework is extensible to problems outside the scope of traditional SAMR applications. As a result, the imprint of any particular SAMR algorithm on the SAMRAI communication infrastructure has been minimized. The object-oriented software design in SAMRAI captures the salient features of data communication in SAMR applications in a general framework. This not only enables extensible and specializable high-level algorithm components in SAMRAI [16], but allows application developers to specialize communication operations for their needs.

2.1 Application Characteristics

In this section, we briefly describe two multi-physics applications built using SAMRAI. Each is a research effort under development to explore the utility of SAMR technology in it respective problem domain. Parallel performance results involving these efforts will be presented in the future. Here, we introduce them to describe the characteristics of applications built with SAMRAI and to motivate the discussion of the SAMRAI communication infrastructure which appears later.

Multi-physics applications often couple different algorithmic components, each of which provides a distinct part of an overall solution scheme. An example of this is the ALPS (Adaptive Laser Plasma Simulator) [9] code under development in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. ALPS uses SAMR to simulate laser-plasma instabilities, using three numerical models to represent different physical processes. Integration of the full system of equations uses separate advance steps for the plasma fluid variables (density, pressure, velocity, and temperature), the light variables (amplitude and intensity), and a nonlinear potential equation that couples ions and electrons. ALPS developers specialized parts of SAMRAI (via class inheritance) and created application-specific routines [16]. The application takes advantage on the ability of the SAMRAI communication framework to coordinate complex data transfer, involving several solution variables and coarsen/refine operations during the different phases of the solution process.

Multi-physics applications also require data quantities having different centerings on the grid and possibly new data structures that are not part of the underlying support framework. A hybrid continuum-particle application [14] code developed as a collaboration between the SAMRAI team and A. Garcia of San Jose State University uses SAMR to couple an Eulerian fluid model to a Direct Simulation Monte Carlo (DSMC) [1] particle model. The goal of this simulation effort is to explore hybrid methods for modeling complex fluid interface dynamics, such as the Richtmeyer-Meshkov instability. This code extends previous continuum-DSMC hybrid work [12] by allowing multiple DSMC regions, and supporting full adaptive mesh capabilities for particles in parallel. This application uses the ability of SAMRAI to support new data representations, including irregular structures like particles, that are not already provided by the framework. Because particle density changes throughout the calculation, the communication operations must also dynamically handle messages with varying size.

These applications provide a glimpse into the sort of application and algorithm space that SAMRAI is designed to explore. SAMRAI provides robust, flexible software support for managing complex algorithm and data coordination operations in these and other SAMR applications.

2.2 Design Goals

To support applications having the characteristics described in Section 2.1, the SAMRAI communication infrastructure possesses the following features:

1. Users can easily describe a data transfer phase of a computation by specifying communication operations that involve a collection of variables and operations to be performed on the associated data. Such operations include copying, temporal and spatial interpolation (including user-defined), and the application of user-defined physical boundary conditions. The descriptions of variable quantities and operations are independent of the grid configuration.

2. The communication framework automatically manages data transfers on a given SAMR patch hierarchy involving data with different mesh centerings (cell, node, etc.), types (integer, float, complex, etc.), and user-defined irregular structures such as particles.

3. New user-defined data types may be introduced without modifying or recompiling the SAMRAI framework.

3. COMMUNICATION INFRASTRUCTURE

SAMR solution algorithms may be represented as set of phases involving numerical computations on individual patches and inter-patch data transfers such as copying, coarsening, and refining. The SAMRAI data communication framework centers on four basic abstractions: a *communication algorithm*, a *communication schedule*, *patch data*, and a *message stream*. These concepts are used to partition SAMR data communication operations into procedures that are expressed at the application level using notions of variables and operators, procedures that manage transfers of variable data on a given SAMR patch hierarchy, and procedures that define data manipulation operations for specific data types. In the following sections, we discuss the role and implementation of each abstraction and how they work together.

3.1 Communication Algorithm

In SAMRAI, a *communication algorithm* is used to describe a data communication phase of a computation in terms of the variables and operations involved. There are two types of communication algorithms, a *coarsen algorithm* and a *refine algorithm*. The names are indicative of the sort of communication operations these objects describe. One moves data from a finer level to a coarser level. The other moves data from coarser levels to a finer level or between patches on the same level (a special case of refinement).

A developer constructs a communication algorithm by specifying the variables involved and the operators needed to perform the desired coarsening or refining operations. For example, in the ALPS application, a communication algorithm is used to represent the filling of ghost cell data before the advance of the plasma fluid variables (density, pressure, temperature), and another communication algorithm represents the inter-patch exchange of light variable during the solution of the light equations. In general, such operations involve different spatial refinement and time interpolation operations for each variable, some of which may be user-defined.

In SAMRAI, the notion of a variable is distinct from storage. Variables define the quantity involved, such as centering on the grid (for array-based types) and the type of the underlying data (e.g., int, float, complex). Variables are used to create data on the mesh via an "abstract factory" mechanism [11]. Usually, a variable object will persist throughout an entire computation, but storage associated with that variable will change as the grid changes. Similarly, a communication algorithm represents a communication phase of a computation independently of any particular mesh configuration. Thus, a communication algorithm is typically constructed during the initialization portion of a computation and is used throughout a calculation.

To summarize, a communication algorithm helps to manage the complexity of SAMR data transfer operations by separating grid-dependent details from the variables and operations involved. That is, a communication algorithm depends only on the variables and operations involved and is independent of a particular grid configuration. This allows an application developer to formulate the communication phases of a numerical solution process independently of the complex details of data movement on an adaptive mesh.

3.2 Communication Schedule

A *communication schedule* manages the data transfer operations required to perform the operations described by a communication algorithm on a particular mesh configuration. A communication schedule computes and stores transactions required to move data between patches. Thus, a schedule depends on a communication algorithm and a particular layout of patches on a SAMR hierarchy.

A communication schedule is constructed using a communication algorithm, which defines the variables and operations involved in moving data, and knowledge of the hierarchy configuration. A schedule is valid as long as the grid configuration used to create it remains unchanged. When the grid changes during adaptive meshing, the schedule is no longer valid and must be regenerated. For example, a communication algorithm can describe how to refine a set of variables between any pair of grid levels in a hierarchy. A different schedule is needed to move the data whenever the grid changes. While the patches in the grid hierarchy are distributed among processors, box information (i.e., the bounding-box region of the patch) and variable information are known to all processors. Thus, a different schedule object is composed on each processor and involves only those transactions involving data on that processor.

The notion of a schedule for managing data communication operations on parallel processors has been described in other work, such as multi-block PARTI [19] and KeLP [2, 10]. The goal of this approach is to amortize the cost of computing complex data dependencies when a set of communication operations may be used multiple times on a given mesh configuration. Although the mesh usually changes frequently in SAMR computations, there are many situations in which schedules are used multiple times before they must be regenerated. SAMRAI borrows ideas from these other efforts but adds extensions to handle multiple data types and variable-length data within the same schedule.

3.3 Patch Data

The communication algorithm and schedule implementations in SAMRAI work with arbitrary collections of variables and coarsen and refine operations, including user-defined variables and operations. To achieve this flexibility, all patch data types in SAMRAI inherit from the same interface. This includes data types provided by the library as well as new types linked to the library when an application is built. The features of this design are described elsewhere [16, 15]. For the purposes of this paper, we note that a *patch* is a container for all simulation data defined over a box region on the grid. During the execution of a communication schedule, local copy operations and operations that marshal/unmarshal data to message streams for parallel communication are invoked through the patch data interface. We describe these in more detail in the next section.

3.4 Abstract Message Streams

The parallel communication framework in SAMRAI uses asynchronous MPI message exchanges. We generate large messages by packing as much data as possible into a single message to minimize startup costs. The result is that a single message exchange occurs between each pair of processors that communicate during the execution of a schedule.

To support variable-length data from multiple sources (i.e., variables and patches) in a single message, SAMRAI utilizes the notion of an *abstract message stream*. Each patch data object provides `packStream()` and `unpackStream()` operations to pack and unpack data to a message stream. When the size of an irregular data type cannot be determined solely from the box region in which it resides, a `getStreamSize()` operation must also be supplied. Thus, array-based continuum data and "grid-less" particle data are communicated using the same message stream; see Figure 1. The full generality of the SAMRAI message streams and patch data support is used is the Euler-DSMC application discussed in Section 2.1.

A key motivation for abstract message streams in SAMRAI is to make it easier to add new data types without modifying or recompiling existing SAMRAI framework code. It

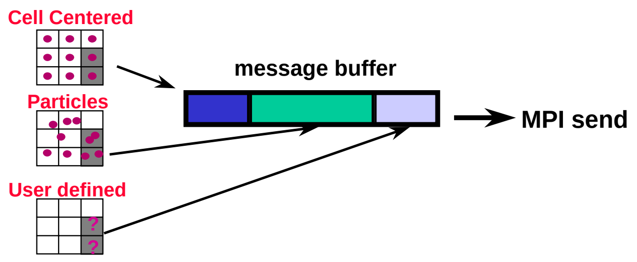

Figure 1: Abstract message streams in SAMRAI support variable-length data from different sources. Packing and unpacking operations are provided by each patch data type.

also eliminates the need for users who may wish introduce new types to work with MPI calls directly. The use of large messages in SAMRAI amortizes message start-up costs over all data that are passed from one processor to another. This may yield significant savings over alternative approaches that use many messages. We remark that our approach uses more copy operations to pack and unpack data from message streams than other approaches that use MPI buffers more directly or use MPI derived datatypes. However, derived datatypes are difficult to use for dynamically changing and variably structured data. In Section 5, we show that data communication is a small portion of overall simulation cost. As a result, we focus our efforts to improve parallel performance on other aspects of SAMR computations.

4. HYDRODYNAMICS APPLICATION

We evaluate SAMRAI performance for standard SAMR applications that utilizes the well-known explicit hydrodynamics algorithm of Berger and Colella [4]. The algorithm uses both spatial and temporal mesh refinement during time integration and applies numerical flux correction operations to maintain global conservation across multiple refinement levels. Object-oriented features of the SAMRAI implementation of this algorithm and its use in other applications are described elsewhere [16]. Here, we focus on aspects relevant to our analysis of parallel performance.

During time advancement on an SAMR patch hierarchy, re-meshing operations are interleaved with integration steps. During re-meshing, computational cells are selected to identify regions where refinement is needed. Cell-tagging is performed on each patch separately and is therefore quite scalable. We use the signature pattern recognition algorithm of Berger and Rigoutsos [6] to cluster tagged cells into logically-rectangular patch regions. Note that re-meshing and integration operations are performed on level at-a-time. Thus, each level is load balanced separately from the others.

The Berger-Rigoutsos algorithm implementation in SAMRAI [17] performs parallel array reductions over the irregular grid structure to build tag signature arrays on each processor. The accumulation of the tagged cells into signature arrays uses global all-reduce operations. These collective communication operations synchronize the procedure on each processor so that the processors construct identical box regions from which to build new patches. This algorithm was designed for small numbers of processors where the cost of global all-reduce operations are negligible.

The boxes constructed by the Berger-Rigoutsos algorithm are further massaged for load balancing. For example, a large box may be chopped into a set of smaller boxes to make it easier to assign the boxes to processors. Each box is chopped until its size is less than the per-processor average size, computed by dividing the total number of computational cells by number of processors. Boxes are then ordered according to their spatial location using a Morton space filling curve algorithm [13] which places a curve through the box centers and partitions the curve. The goal of this last step is to maximize the assignment of adjacent patches to the same processor. The boxes assigned to each processor are used to generate patches on that processor.

Once a new patch level is constructed and load balanced, the integration algorithm constructs new communication schedules using its communication algorithms and the new patch configuration. The time integration routines for the hyperbolic problems discussed in Section 5, including the construction of the communication algorithms and schedules, are provided by SAMRAI. Numerical kernels to integrate the variables on each patch are supplied by users of the library. While SAMRAI uses the object-oriented capabilities of C++ to manage application complexity, object-orientation is not introduced into computationally-intensive operations. Users of SAMRAI write patch-based numerical kernels in FORTRAN or C for simplicity and efficiency.

5. PERFORMANCE RESULTS

We investigate parallel performance of two adaptive problems implemented using SAMRAI. The first models a 3D propagating spherical shock with the Euler equations of gas dynamics. This problem is not scaled; that is, the same sized problem is run on all processor partitions. The second problem models a 3D sinusoidal advecting front with the scalar linear advection equation. This problem is scaled, in that the problem size increases proportionately with number of processors. We use the non-scaled problem to investigate how SAMR calculation perform when processors are added to a fixed problem. We use the scaled problem study trends as problem size is increased as more processors are applied.

Both problems employ the hyperbolic time integration algorithm supplied by SAMRAI. They differ only in the number of variables involved and the operations performed in the numerical kernels. One solution variable appears in the linear advection application while a system of five variables represents the solution in the Euler case. Because the computational effort to update the numerical solution in the linear advection case is less, the linear advection problem has higher data communication and re-meshing costs relative to total computation time than the Euler case.

The remainder of this section is comprised of four parts. The first two describe our analysis of the performance of the non-scaled and scaled problems. We report wallclock time in different phases of each calculation across a range of processors to determine trends in parallel performance of each phase. We find that two operations, load balance and communication schedule construction, tend to have compet-

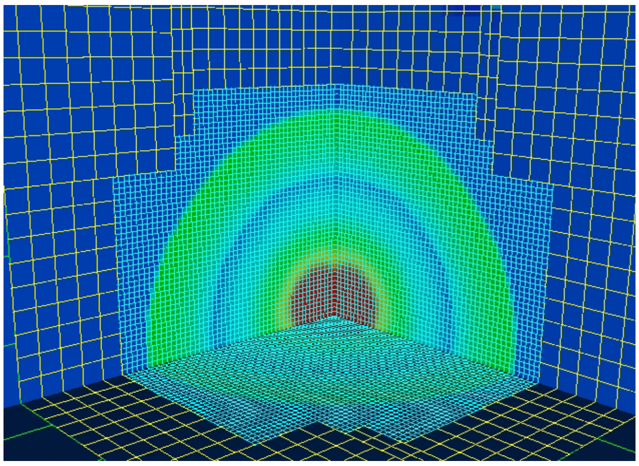

Figure 2: Density contours overlaid on adaptive grid system for spherical shock calculation.

ing interests with respect to parallel scaling. The last two sub-sections provide details about these operations and describe our observations regarding our attempts to minimize their costs and how cost reduction in one affects the performance of the other.

All calculations are performed on the IBM ASCI Blue Pacific system. This system is an IBM machine constructed of 256 four processor SMP nodes, 244 of which are available for typical batch runs. Each processor is a 332 MHz PowerPC 604e. Each node has 1.5 GB memory. An omega topology interconnect network supports up to 150Mbytes/s bi-directional bandwidth between nodes.

5.1 Non-scaled Problem

In this section, we discuss the performance of the non-scaled spherical shock Euler problem. The adaptive problem uses three levels of mesh resolution where the mesh is refined by a factor of four between successive levels; see Figure 2. Figure 3 shows how the number of computational cells on each level changes with simulation time. The number of cells on the finest level constitutes 94% – 96% of the total cells in the calculation. Of the total time spent in the time integration portion of the solution process, 97% – 98% is spent on the finest level for all processor partitions. Problem size grows roughly linearly as the simulation advances during the course of the 15 coarsest grid timesteps over which we ran the computation. This growth is due from the fine mesh adapting to resolve the spherically-expanding shock. The same problem size is used on all processor partitions.

The two primary phases of the calculation are grid generation and time integration. Grid generation involves three major steps: construction of new patch regions from tagged cells (Berger-Rigoutsos procedure plus load balance), construction of communication schedules, and data movement from the old mesh configuration to the new configuration. Time integration uses numerical routines outside of SAMRAI. Data movement to fill ghost cell regions during time integration and to redistribute data during re-meshing is performed by SAMRAI.

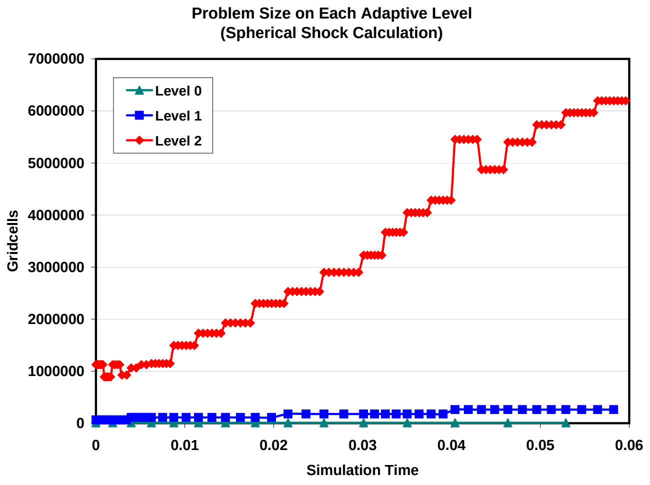

Figure 3: The number of cells on the finest level grows during the coarse of the simulation. The number of cells on the two coarser levels remains roughly the same.

Table 1 shows wallclock time measurements for time integration and grid generation. In the table, total integration time includes time in numerical kernels, wait time due to load imbalance, and communication overhead. Grid generation time includes time spent in Berger-Rigoutsos, schedule generation, and data redistribution. The entry labeled “other” includes parts of the calculation that fell outside these six phases, such as cell-tagging, level data initialization, and load balancing. The times reported are averages across processors; That is, times are summed over all processors and then divided by the number of processors. They also represent an average of several runs on the fully-loaded system performed at different times over the course of two weeks.

Figure 4 shows a plot of the data in Table 1. In the time integration phase, the two main parallel implementation overheads are load imbalance and communication. Of these, we find load imbalance to be most significant. Further analysis reveals that the imbalance is not caused by inefficient distribution of cells to processors. Rather, it results from numerical operations with different costs performed in different regions of the flow. This is discussed in more detail in Section 5.3.

Grid generation shows poorer scaling than the time integration. Of the operations used in the grid generation, communication schedule generation is the most dominant cost and Berger-Rigoutsos operations have the most detrimental effect on scaling performance. Recall that the Berger-Rigoutsos implementation, which was designed for smaller numbers of processors, performs all-reduce operations to construct signature arrays. While it is efficient on 64 processors, requiring only 1% of the total time, its cost grows with the number of processors due to use of the all-reduce operations. It requires up to 22% of the total time on 512 processors. Global all-reduce operations on ASCI Blue Pacific are particularly slow and is an acknowledged problem. Hence, the observed growth in the cost of the Berger-Rigoutsos al-

Processors	64		128		256		512	
Total	**1874.3**		**1087.3**		**729.1**		**652.1**	
Time Integration	**1570.1**	**84%**	**868.9**	**80%**	**510.1**	**70%**	**335.1**	**51%**
Computation - num kernels	1031.4	55%	516.6	48%	259.8	36%	131.0	20%
Wait time - load imb	431.1	23%	284.8	26%	202.1	28%	165.8	25%
Communication overhead	107.8	6%	67.5	6%	48.2	7%	38.3	5%
Grid Generation	**269.8**	**14%**	**181.4**	**17%**	**172.5**	**24%**	**241.7**	**37%**
Berger-Rigoutsos	13.6	1%	19.0	2%	42.1	6%	145.1	22%
Schedule construction	245.4	13%	157.0	14%	126.2	17%	93.0	14%
Data re-distribution	10.8	1%	6.9	1%	7.8	1%	17.7	3%
Other	**35.3**	**2%**	**36.7**	**3%**	**46.5**	**6%**	**71.6**	**11%**

Table 1: Timing results for non-scaled spherical shock calculation. Results show wallclock time for each operation and percentage of total wallclock time for the different phases of the calculation.

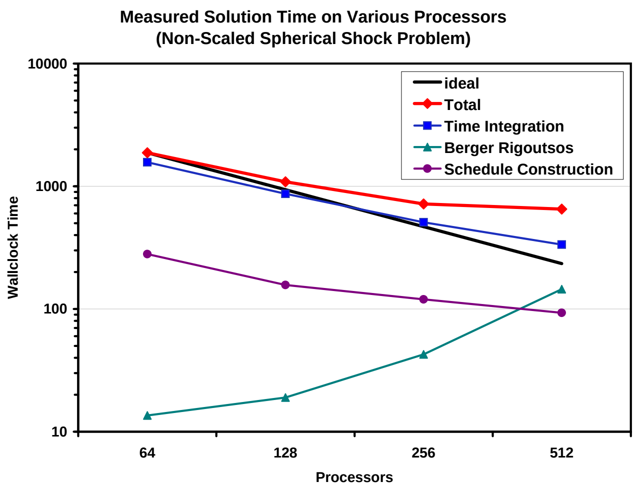

Figure 4: Wallclock time measurements in Table 1 for the non-scaled spherical shock calculation.

gorithm may be less on other architectures. We have yet to perform detailed tests on other systems.

The results reveal interesting characteristics about the communication vs. computation costs for this calculation. Including data re-distribution after re-meshing and inter-level data transfers performed during time integration, the total cost of data movement remains roughly constant at about 7% of total on all processor counts tested. Thus, the application is not communication bound and our object-oriented design (Section 3) does not impeded performance. The most inefficient parts of the computation are instead related to adaptive gridding. In particular, the cost of generating communication schedules and the Berger-Rigoutsos algorithm are the largest parallel performance bottlenecks. While communication schedule generation is performed by the SAMRAI infrastructure, the dominant part of their cost are due to algorithmic inefficiencies in their implementation, not from a misuse of object-orientation. We will address these algorithmic concerns in future work. This point is discussed further in Section 5.4.

5.2 Scaled Problem

In this section, we discuss the performance of the scaled advecting sinusoidal front problem. In this experiment, we artificially control the mesh generation process by manually scaling the mesh so that the number of gridcells per processor remains constant for the duration of each computation. To do this, we first run a problem on P processors and store the mesh after each re-meshing step. This mesh is then refined as we increase the number of processors. To go from P to $2P$ processors, we double the number of cells in one direction. To go from $2P$ to $4P$, we double the cells in an additional direction, and so on. The use of the linear advection equation allows us to force the same time-stepping sequence on all refined grids. The use of a sinusoidal front helps us make grid configurations more representative of typical SAMR problems.

The adaptive problem uses three levels of mesh resolution where the mesh is refined by a factor of four between levels, as shown in Figure 5. Figure 6 shows how the number of computational cells on each level changes with simulation time. As was the case with the non-scaled spherical shock problem, the vast majority of cells are on the finest refinement level. However, note that the overall problem size remains roughly constant over the course of the computation. The calculation is run over a total of 25 coarsest grid timesteps.

The scaled problem is run from 32 to 512 processors. Table 2 shows wallclock time measured for time integration and grid generation, and the associated operations within each. The entry labeled “other” includes parts of the calculation that fell outside the time integration and grid generation phases, as cell-tagging, level data initialization, and load balancing. This information is plotted in Figure 7.

As we observed for the non-scaled problem, communication costs (i.e., MPI communication) including both inter-level data updates during the time integration and data-redistribution during re-meshing, scale well and constitute a relatively small percentage of the overall execution time. Although the communication to computation ratio of this problem is greater than in the Euler problem, we still experience low communication overheads with respect to overall computation time. This is further evidence that the object-oriented implementation is not detrimental to performance.

Processors	32		64		128		256		512	
Total	**1458.2**		**1451.5**		**1687.9**		**1772.6**		**2123.7**	
Time Integration	**1206.8**	**83%**	**1209.1**	**83%**	**1485.8**	**88%**	**1452.2**	**82%**	**1516.5**	**71%**
Computation - num kernels	982.1	67%	950.8	66%	981.9	58%	945.6	53%	954.8	45%
Wait time - load imb	91.2	6%	123.7	9%	369.1	22%	381.2	22%	428.6	20%
Communication overhead	133.5	9%	134.6	9%	134.8	8%	125.4	7%	133.1	6%
Grid Generation	**213.9**	**15%**	**193.3**	**13%**	**121.1**	**7%**	**188.7**	**11%**	**369.4**	**17%**
Schedule construction	197.4	13%	174.0	12%	104.4	6%	175.5	10%	363.1	17%
Data re-distribution	12.9	1%	15.0	1%	18.1	1%	18.7	1%	19.8	1%
Other	**37.5**	**3%**	**49.0**	**3%**	**80.7**	**5%**	**131.6**	**7%**	**237.7**	**11%**

Table 2: Timing results for scaled advecting front calculation. Results show wallclock time for each operation and percentage of total wallclock time for the two main phases of the calculation, time integration and grid generation.

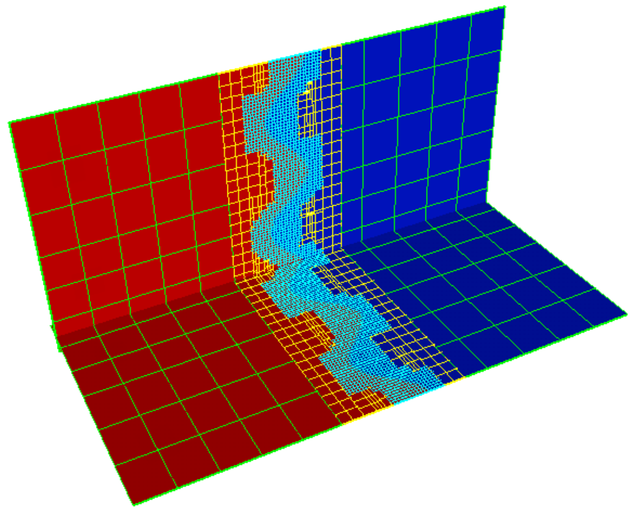

Figure 5: Scaled advecting sinusoidal front problem - density contours overlaid on adaptive grid.

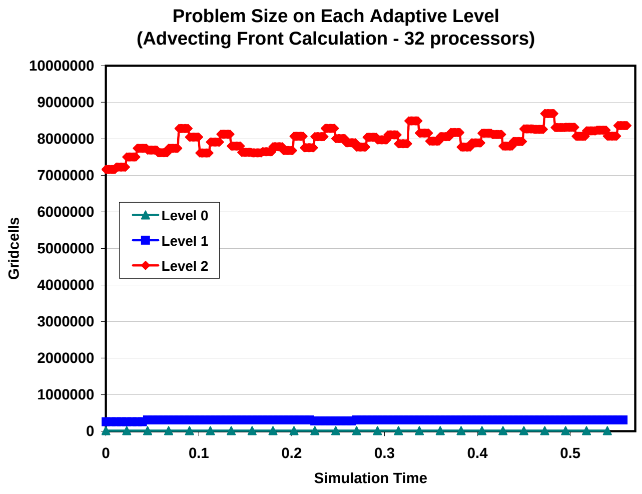

Figure 6: The number of computational cells on each level for the 32 processor case of the scaled advecting front calculation. For the 64, 128, 256, and 512 processor cases, the pattern is identical but the number of cells increases proportional to the processor count increase.

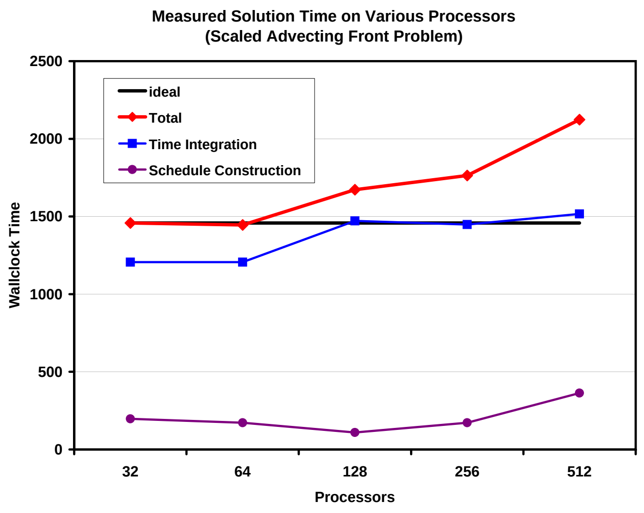

Figure 7: Wallclock time measurements in Table 2 for the scaled advecting front calculation.

Note that the cost of the Berger-Rigoutsos algorithm is not included because we generate the grid hierarchy manually.

The two primary sources of inefficiency in these computations are load imbalance and the cost of generating communication schedules. In analyzing this problem, we found that these two operations have competing efficiency requirements. Attempts to improve the efficiency in one cause a decrease in the other. The next two sub-sections describe our observations in more detail.

5.3 Load Balance

Achieving good load balance is challenging for adaptive calculations due to the dynamic and non-uniform (in space) nature of the workload. Load imbalance is a dominant cost in both the non-scaled and scaled problems discussed in previous sections. In this section, we analyze the observed imbalance and propose techniques for its remedy.

The load balance strategy outlined in Section 4 attempts to distribute patches to processors to balance the number of computational cells on each processor. We refer to this as *predicted load balance.* The assumption is that the amount

Processors	64	128	256	512
predicted	94%	87%	77%	64%
measured	71%	66%	55%	48%

Table 3: Predicted vs. Measured overall load balance efficiency for non-scaled spherical shock calculation. The disparity arises from different numerical operations being performed in the numerical kernels in different computational cells.

of computational work is proportional to the number of computational cells. However, we find this is not always the case. Table 3 shows predicted and measured load balance for the non-scaled spherical shock calculation. Since load balance changes after each re-meshing step, the overall load balance reported is is computed by averaging over steps. That is, predicted load balance σ at a particular step is computed as:

$$\sigma = \frac{\sum_{p=1}^{P} n(p)/P}{n(p)_{max}} \tag{1}$$

where $n(p)$ is the number of cells on processor p and P is the total number of processors, and $n(p)_{max}$ is the maximum number of cells on all processors. Predicted load balance (the values shown in Table 3) is computed as:

$$\sigma_{overall} = \frac{\sum_{i=1}^{steps}(n(i) \cdot \sigma(i))}{\sum_{i=1}^{steps} n(i)} \tag{2}$$

where $n(i)$ is the total number of computational cells at step i and $\sigma(i)$ is the load balance efficiency at the step. This formula weights the load balance on each level and at each step by the number of computational cells on the level. Thus, we factor the amount of computational work into determination of the overall average load balance. Overall measured load balance is computed in exactly the same way, except that $n(p)$ represents the computed time in the numerical kernels on processor p, and $n(i)$ is the computed time across all processors at step i.

The disparity in predicted and measured load balance results from different operations in the numerical kernels performed in different regions of the flowfield. For example, in regions near shocks and rarefactions, flux computations are more expensive. The non-uniform operation distribution breaks the uniform workload assumption (i.e. that computational work on each cell is uniform). To reinforce this point, Table 4 shows the overall predicted and measured load balance for the scaled advecting front calculation. This calculation performs identical operations in each cell and the disparity between predicted and measured efficiency is much less.

Table 4 also shows the average number of patches per processor per step on the finest level in the scaled advecting front calculation. The drop in predicted load balance efficiency corresponds to the decreasing number of patches on each processor. We try to restrict the growth of the total number of patches in the problem as we scale up the number of processors because increasing the total number of patches adversely affects the efficiency of communication schedule construction. This is discussed in more detail in Section 5.4. As the number of patches per processor decreases, the predicted efficiency decreases. That is, it becomes more difficult to distribute patches to processors using a bin-packing algorithm (discussed in Section 4) to balance the workload as the number of bins increases and the relative number of patches per bin decreases. We also remark that the use of a space-filling curve to enhance locality places further restrictions on the potential bins to which each patch may be assigned. However, we have not analyzed the impact of this constraint on load balance efficiency.

Processors	32	64	128	256	512
predicted	95%	92%	80%	73%	70%
measured	93%	91%	75%	*–	*–
avg patches/proc	26.6	15.7	6.6	5.2	4.5

Table 4: Predicted vs. Measured overall load balance efficiency for scaled advecting front calculation. * **Instrumentation in the code to record computed load balance efficiency performs an extra global communication operation at each step, which polluted timing results on large numbers of processors.**

Processors	32	64	128	256	512
predicted	95%	95%	94%	93%	92%
avg patches/proc	26.6	25.0	23.7	21.9	19.5

Table 5: Predicted load balance efficiency for scaled advecting front calculation for case where total number of patches is scaled proportionately with the number of processors.

We found that we can achieve more efficient load balance when the number of patches is scaled proportionally with the number of processors. Table 5 shows predicted load balance for the scaled advecting front calculation when the number of patches is increased in a manner roughly proportional to the number of processors. The number of patches per processor per step for the finest level is also shown. The table shows that load balance remains good across the range of processors as long the number of patches per processor is held roughly constant.

In summary, we find two primary causes of poor load balance. First, the assumption that computational work is uniform in all computational cells does not always hold, leading to imbalance in the distribution of workload. We will address this in the future by developing a non-uniform load balancing approach that accounts for the spatially non-uniform distribution of work when chopping boxes and distributing patches to processors. For example, the workload can be computed by estimating the time to update each cell. Since the large-scale features of the flowfield change slowly relative to the frequency of re-meshing, this should be a good prediction of workload for the next integration step. The second source of inefficiency occurs from restricting the total number of patches in the problem. We verified that we can maintain good load balance across a wide range of processors by scaling the total number of patches proportionately with the number of processors. Although increasing the total number of patches can improve load balance, it has a detrimental effect on communication schedule generation costs, which is the subject of Section 5.4.

5.4 Schedule Generation costs

Communication schedule generation costs can have a large performance impact on the SAMR problems we investigated on large numbers of processors. This section discusses our observations. In particular, we look at how effectively their construction costs were amortized through re-use and evaluate problem characteristics that affect their cost.

Communication schedules compute the data dependencies between patches in the hierarchy. They are generated whenever the grid changes after a re-meshing operation. In the results reported earlier, schedule generation cost is significant (up to 17%) in both the scaled and non-scaled calculations. A goal of maintaining schedules as separate objects is to amortize the cost of their generation. Table 6 shows how effective this amortization is for the non-scaled spherical shock calculation. This calculation was run for 15 coarse-grid timesteps. The table shows how many times schedules are generated for a particular communication phase of the computation, how many times schedules are used to communicate data, and their generation cost as a percentage of total computation time. The range in percentage of time reported reflects the different percentages measured across the range of processors, from 64 to 512 processors.

Communication operation	**# times sched created**	**# times sched used**	**cost % total time**
time adv-refine	33	259	7-8%
time adv-coarsen	61	61	1-3%
regrid-data redist	32	32	1-3%

Table 6: Communication schedule re-use statistics for the non-scaled spherical shock calculation. The columns indicate the number of times schedules are constructed, number of times it is used to invoke communication, and cost of construction as a percentage of the total cost.

The most expensive schedules to generate are the ones used to refine data between levels and set boundary conditions to fill ghost cells before each integration step on each level. Each of these schedules is used an average 8 times before it had to be re-generated. However, not all schedules are re-used. Each data re-distribution schedule, which moves data to the new grid configuration after re-meshing, is used only once. It cannot be reused since the grid is different each time this communication phase is invoked. The schedules that coarsen data at synchronization steps during time integration are also generated each time they are used. We could reuse these coarsen schedules. However, since these schedules are relatively inexpensive to build and their reuse frequency is much less than the first ghost cell filling schedules (every step on a level vs. every step on the next coarser level), we chose to simplify our implementation and re-generate these schedules each time they are used.

We find the cost of constructing communication schedules goes up with the number of patches in the problem. As discussed in Section 3.2, a schedule is constructed separately by each processor to form a list of data transactions involving patches on that processor. Construction of the transaction list is a local operation on each processor. However, the need to fill ghost regions from coarser levels requires intersection of each patch against all others in the problem. The cost of schedule construction grows therefore roughly as $O(N^2)$, where N is the total number of patches used in the problem. This causes a tradeoff in the optimization strategies for load balance and communication schedule generation cost. We show in Section 5.3 that it is possible to achieve nearly ideal load balance by scaling N proportionately with P, but this benefit is outweighed by an increase in the cost of performing schedule construction with increasing N.

In summary, we can amortize the construction cost of some schedules by using them multiple times and see a definite cost advantage in doing so. The most important issue, however, is that with the current implementation, schedule generation costs grow as $O(N^2)$ with the total number of patches in the problem. In future work, we will explore techniques to reduce the cost of this process by exploiting spatial relationships between patches in the schedule construction process. For example, octree data structures used to efficiently traverse spatial data structures could reduce the computational complexity of schedule construction to $O(N \log N)$.

6. CONCLUDING REMARKS

In this paper, we described the parallel communication infrastructure in the SAMRAI library and explored its performance for structured AMR applications. We observed that, in general, complex algorithms and data communication operations found in AMR computations do not preclude scaling application codes to large number of processors. However, there are fundamental AMR operations that require special attention to achieve scalable performance. We draw several conclusions from the parallel performance analysis carried out in this work. First, the use of object-oriented abstractions in SAMRAI does not impede large-scale parallel performance. Second, the parallel implementation of the Berger-Rigoutsos cell clustering algorithm we use exhibits unacceptably-large costs on large processor counts. Third, AMR applications must treat spatially non-uniform computational cost distributions to achieve good parallel load balance. Fourth, the implementation of communication schedule generation in SAMRAI currently prohibits scaling applications to large numbers of processors due to $O(N^2)$ computational complexity in certain operations.

The object-oriented design of the parallel communication framework in SAMRAI (Section 3) is based on a survey of the data management needs of a broad range of multi-physics applications. To assess the parallel performance of this infrastructure, we studied scaling efficiency of a standard hyperbolic SAMR algorithm over a range of processors on the distributed memory IBM ASCI Blue Pacific parallel platform. Timing measurements for both non-scaled and scaled adaptive problems reveal that total data communication cost (i.e., MPI message exchanges) is 8% or less when run on up to 512 processors. Thus, the test applications are not bound by communication costs. Although the test applications do not exploit the full flexibility of the framework, we find that use of object-oriented abstractions in the implementation of these applications do not detrimentally impact performance.

Most of the scaling inefficiencies that we observed are associated with adaptive gridding operations. For example, the Berger-Rigoutsos cell clustering algorithm is efficient on small numbers of processors; it constitutes only about 1% of the total execution time on 64 processors. However, the cost of this algorithm grows to about 22% on 512 processors. The primary source of the performance degradation is the use of global reduction operations in our implementation of the algorithm. This algorithm was originally developed for smaller-scale parallel systems with fewer processors and, under these circumstances, it incurs acceptable costs. However, the collective communication operations are a bottleneck on large numbers of processors. We plan to investigate alternative algorithms and other approaches that rely less on global reductions.

The two other major sources of inefficiency that we observed are poor load balance and costly communication schedule construction. Interestingly, these operations represent competing interests. We verified that good load balance may be achieved if we grow the total number of patches N in the problem proportionately with the number of processors. However, in our current implementation schedule creation costs uses operations with $O(N^2)$ computational complexity. Thus, the benefit of improved load balance efficiency can be overwhelmed by the increase in communication schedule generation cost. In future work, we plan to explore ways to exploit information about the spatial relationships among patches into the schedule construction process. For example, octree data structures, which efficiently traverse spatial data structures, may prove useful.

Finally, we observed significant disparities between predicted and measured load balance efficiency in our computations. Judicious code instrumentation revealed that this was due to different operations being used in different regions of the flow. For example, the computation of numerical flux terms near rarefactions and shocks is more computationally complex, and hence more expensive, than in regions of nearly constant flow. Our load balance procedure that assumed that the computational expense to update each cell is inappropriate under these circumstances. This is an important issue to consider in any computation in which numerical operations are not uniformly distributed. We are currently investigating more robust non-uniform load balance techniques that weight individual cells according to computational work.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the assistance of Dr. Jeff Vetter and Dr. John May from the parallel tools group in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory in assisting with tools to better understand the parallel performance of our codes.

8. REFERENCES

[1] F. Alexander and A. Garcia. Direct simulation Monte Carlo. *Computers in Physics*, 11:588, 1997.

[2] S. B. Baden, P. Colella, D. Shalit, and B. V. Straalen. Abstract KeLP. In *10th SIAM Conference on Parallel Processing for Scientific Computing*, Portsmouth, Virginia, March 2001.

[3] J. Bell and P. Colella. AMR software packages. See `http://seesar.lbl.gov/AMR/index.html`.

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

[5] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

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

[7] G. Bryan and M. L. Norman. Simulation x-ray clusters with adaptive mesh refinement. In *Proceedings 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.

[8] 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 *Proceedings of SC00*, 2000.

[9] M. R. Dorr, F. X. Garaizar, and J. A. F. Hittinger. Simulation of laser plasma filamentation using adaptive mesh refinement. Technical Report UCRL-JC-138330, LLNL, 2001. Submitted to J. Comput. Phys.

[10] S. J. Fink, S. B. Baden, and S. R. Kohn. Flexible communication schedules for block structured applications. In *Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, California, August 1996.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software*. Addison-Wesley Publishing Co., Menlo Park, CA, 1995.

[12] A. Garcia, J. Bell, W. Crutchfield, and B. Alder. Adaptive mesh and algorithm refinement using direct simulation Monte Carlo. *Journal of Computational Physics*, 154:134–155, 1999.

[13] Gutman. Use of morton space-filling curve for load balance. *Dr. Dobb's Journal*, pages 115–121, July 1999.

[14] R. Hornung. A hybrid model for gas dynamics: Continuum-DSMC with AMR. In *First SIAM Conference on Computational Science and Engineering, Washington D.C.*, Sept 21–23 2000. Also available as Lawrence Livermore National Laboratory technical report UCRL-VG-139774.

[15] R. D. Hornung and S. R. Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/SAMRAI`.

[16] R. D. Hornung and S. R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency: Theory and Practice*, 2001. (to appear).

[17] S. R. Kohn. A parallel software infrastructure for dynamic block-irregular scientific calculations, Ph.D thesis. Technical Report CS95-429, Dept. of Computer Science and Engineering, University of California San Diego, 1995.

[18] M. Parashar and J. C. Browne. A common data management infrastructure for parallel adaptive algorithms for PDE solutions. In *Proceedings of Supercomputing 97*, 1997.

[19] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.