CAIP TECHNICAL REPORT

Report Number: TR-275

A Simulation Framework for Evaluating the Runtime Characteristics of Structured Adaptive Mesh Refinement Applications

Sumir Chandra, Mausumi Shee, and Manish Parashar

The Applied Software Systems Laboratory

ECE/CAIP, Rutgers University

{sumir, mshee, parashar}@caip.rutgers.edu

Abstract

Structured adaptive mesh refinement (SAMR) methods for the numerical solution of partial differential equations yield highly advantageous ratios for cost/accuracy as compared to methods based on static uniform approximations. SAMR techniques seek to improve the accuracy of the solution by dynamically refining the computational grid at various levels in regions of high local solution error. Distributed implementations of these methods lead to significant challenges in dynamic data-distribution, load-balancing, and runtime management. This report presents a simulation framework that requires a post-partitioning trace as input and evaluates the runtime characteristics of SAMR applications in terms of the quality metric – data migration, intra- and inter-level communication, and load imbalance. The simulation framework serves as a useful tool for evaluating the runtime behavior and performance of different SAMR partitioners.

1. Introduction

Dynamically adaptive mesh refinement (AMR) [1] methods for the numerical solution to PDE's employ locally optimal approximations, and can yield highly advantageous ratios for cost/accuracy when compared to methods based upon static uniform approximations. These techniques seek to improve the accuracy of the solution by dynamically refining the computational grid in regions with large local solution error. Structured adaptive mesh refinement (SAMR) methods are based on uniform patch-based refinements overlaid on a structured coarse grid, and provide an alternative to the general, unstructured AMR approach. These methods are being effectively used for adaptive PDE solutions in many domains, including computational fluid dynamics [2, 3], numerical relativity [4, 5], astrophysics [6, 7], and subsurface modeling and oil reservoir simulation [8, 9]. Methods based on SAMR can lead to computationally efficient implementations as they require uniform operations on regular arrays and exhibit structured communication patterns. Furthermore, these methods tend to be easier to implement and manage due to their regular structure. Distributed implementations of these methods offer the potential for accurate solutions of physically realistic models of complex physical phenomena. These implementations lead to interesting challenges in dynamic resource allocation, data-distribution, load-balancing, and runtime management. Critical among these is the partitioning of the adaptive grid hierarchy to balance load, optimize communication and synchronization, minimize data migration costs, and maximize grid quality (e.g. aspect ratio) and available parallelism.

This report presents a simulation framework that requires a post-partitioning trace as input and evaluates the runtime characteristics of SAMR applications in terms of the quality metric – data migration, intra- and inter-level communication, and load imbalance. The simulation framework serves as a useful tool for evaluating the runtime behavior and performance of different SAMR partitioners.

2. Structured Adaptive Mesh Refinement

The numerical solution to a PDE is obtained by first discretizing the problem domain. One approach is to introduce a structured uniform Cartesian grid. The unknowns of the PDE are then approximated numerically at each discrete grid point. The resolution of the grid (or grid spacing) determines the local and global error of this approximation, and is typically dictated by the solution-features that need to be resolved. The resolution also determines computational costs and storage requirements. Dynamically adaptive solution techniques for PDE's provide a means for concentrating additional grid resolution and computational resources to regions in the application domain with large error. These techniques potentially lead to more efficient and cost-effective solutions to time-dependent problems exhibiting localized features, viz. shocks, discontinuities, or steep gradients.

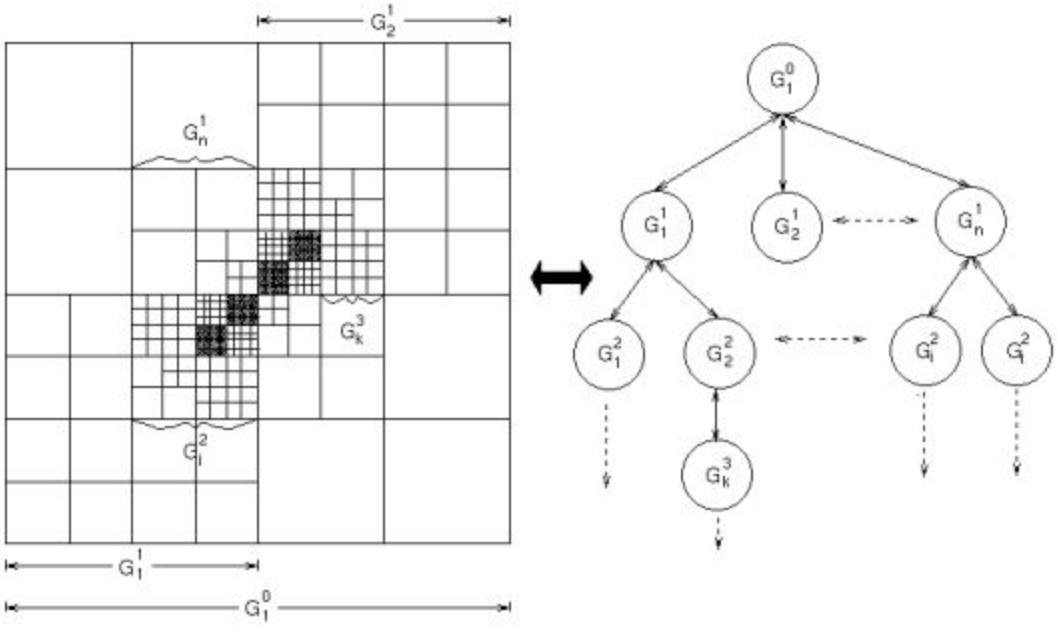

Figure 1: Berger-Oliger SAMR formulation

In the case of SAMR methods, dynamic adaptation is achieved by tracking regions in the domain that require higher resolution, and dynamically overlaying finer grids on these regions. These techniques start with a coarse base grid with minimum acceptable resolution that covers the entire

computational domain. As the solution progresses, regions in the domain with large solution error, requiring additional resolution, are identified and refined. Refinement proceeds recursively so that the refined regions requiring higher resolution are similarly tagged and even finer grids are overlaid on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The adaptive grid hierarchy corresponding to the SAMR formulation by Berger and Oliger [10] is shown in Figure 1. A selection of snap-shots of the adaptive grid hierarchy for the Richtmyer-Meshkov 3D SAMR application is shown in Figure 2.

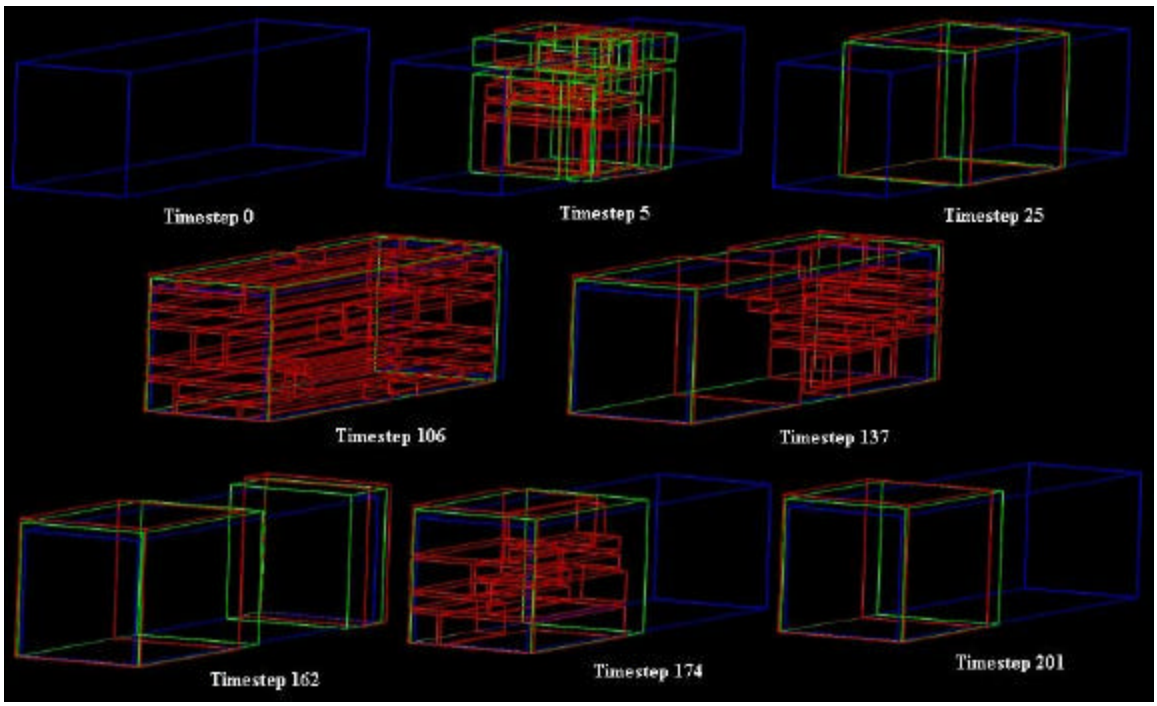

Figure 2: Dynamic adaptation in 3-D Richtmyer-Meshkov application

3. Computation and Communication Behavior in SAMR

In the targeted SAMR formulation, the grid hierarchy is refined both in space and in time. Refinements in space create finer level grids which have more grid points/cells than their parents. Refinements in time mean that finer grids take smaller time steps and hence have to be advanced more often. As a result, finer grids not only have greater computational loads, but also have to be integrated

and synchronized more often. This results in a space-time heterogeneity in the SAMR adaptive grid hierarchy. Furthermore, regridding occurs at regular intervals at each level of the SAMR hierarchy and results in refined regions being created, moved, and deleted.

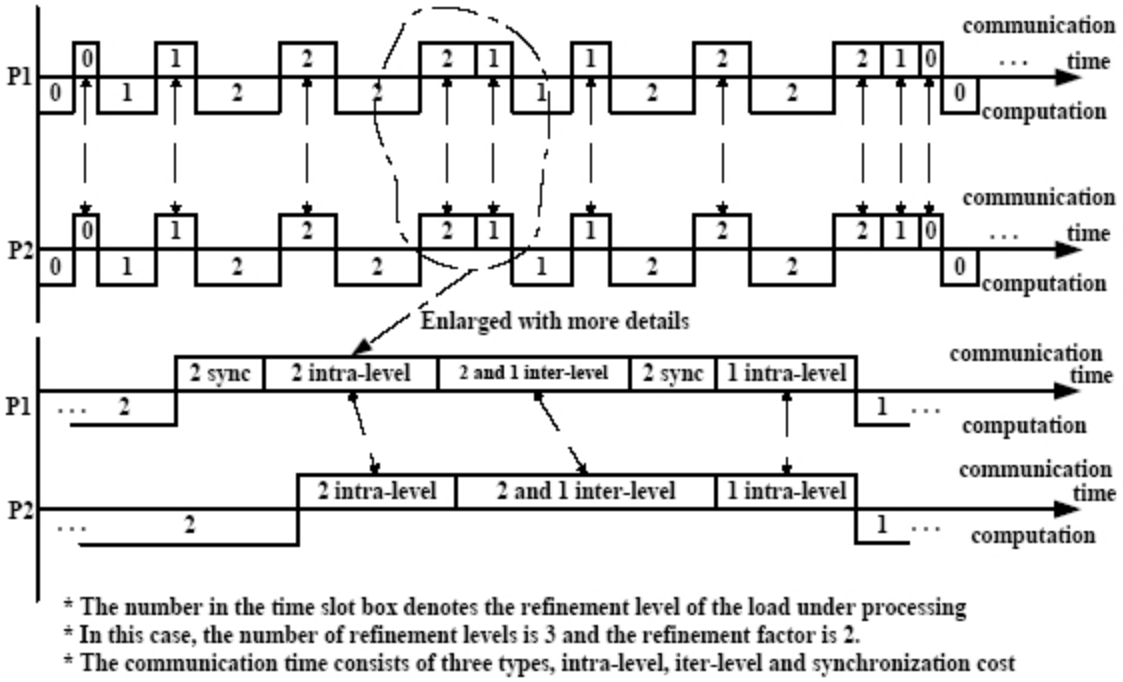

Figure 3: Timing diagram for parallel SAMR algorithm

Parallel implementations of SAMR applications typically partition the dynamic grid hierarchy across available processors and the processors operate on their local portions of the computational domain in parallel. Each processor starts at the coarsest level, integrates the patches at this level, and performs intra-level or "ghost" communications to update the boundaries of the patches. It then recursively operates on the finer grids using the refined time steps - i.e., for each step on a parent grid, there are multiple steps (equal to the time refinement factor) on a child grid. When the parent and child grids are at the same physical time, inter-level communications are used to inject information from the child to its parent. The solution error at different levels of the SAMR grid hierarchy is evaluated at regular intervals and this error is used to determine the regions where the hierarchy needs to be locally refined or coarsened. Dynamic repartitioning and redistribution is typically required after this step.

The timing diagram (note that this diagram is not to scale) in Figure 3 illustrates the operation of the SAMR algorithm described above using a 3-level grid hierarchy [11]. For simplicity, only the computation and communication behaviors of processors P1 and P2 are shown. The three components of communication overheads are illustrated in the enlarged portion of the time line. Note that the timing diagram shows that there is one time step on the coarsest level (level 0) of the grid hierarchy followed by two time steps on the first refinement level and four time steps on the second level, before the second time step on level 0 can start. Also, note that the computation and communication for each refinement level are interleaved.

The overall efficiency of parallel SAMR applications is limited by the ability to partition the underlying grid hierarchies at runtime to expose all inherent parallelism, minimize communication and synchronization overheads, and balance load. A critical requirement while partitioning these adaptive grid hierarchies is the maintenance of logical locality, both across different levels of the hierarchy under expansion and contraction of the adaptive grid hierarchy structure, and within partitions of grids at all levels when they are decomposed and mapped across processors. The former enables efficient computational access to the grids while the latter minimizes the total communication and synchronization overheads. Furthermore, the grid hierarchy is dynamic and application adaptation results in application grids being dynamically created, moved, and deleted. This behavior makes it necessary and quite challenging to efficiently re-partition the hierarchy at runtime to both balance load and minimize communication/synchronization overheads.

As shown in Figure 3, the communication overheads of parallel SAMR applications primarily consist of three components: (1) *Inter-level communications* are defined between component grids at different levels of the grid hierarchy and consist of prolongations (coarse to fine transfer and interpolation) and restrictions (fine to coarse transfer and interpolation); (2) *Intra-level communications* are required to update the grid-elements along the boundaries of local portions of a

distributed grid and consist of near-neighbor exchanges. These communications can be scheduled so as to be overlapped with computations on the interior region; and (3) *Synchronization costs* occur when load is not balanced among processors at any time step and at any refinement level. Note that there are additional communication costs due to the data movement required during dynamic load-balancing and redistribution.

Clearly, an optimal partitioning of the SAMR grid hierarchy and scalable SAMR implementations require a careful consideration of the timing pattern and the communication overheads described above. A critical observation from Figure 3 is that, in addition to balancing the total load assigned to each processor and maintaining parent-child locality, the load balance at each refinement level and the communication and synchronization costs within a level need to be addressed.

4. Partitioning Quality Metric

The partitioning requirements for an adaptive application (and the performance of a particular partitioner) depend on the current state of the application and the computing environment. Therefore, it is of little consequence to discuss the absolute "goodness" of a certain partitioning technique. We base our characterization of partitioning behavior on the tuple {partitioner, application, computer system}, (PAC) [12]. Furthermore, we define a five-component metric to evaluate each PAC. The goal of the metric is to capture the overall runtime behavior of the partitioner. We believe that this is critical to understanding the suitability of a particular partitioner or why one PAC works better than another PAC. The proposed metric for characterizing the quality of a PAC include:

1. *Load imbalance*: This component measures the load imbalance created by a partitioner. It occurs when any processor has more than average load, and can cause performance to deteriorate rapidly. Note that the load associated with a patch is the summation of the work associated with each cell/vertex in the patch (all cells/vertices in a patch may not require the

same amount of work). This metric component assumes greater significance in a computation-dominated environment.

2. *Communication requirement*: This component is a measure of the inter-processor communication required by the SAMR algorithm, given a partitioning of the grid hierarchy. It includes communication between neighboring grid components and communication across refinement levels. This cost is determined by the geometrical intersections between SAMR grid components. This metric component assumes greater significance in a communication-dominated environment.
3. *Amount of data migration*: This component is a measure of the amount of data that has to be moved between processors when the application transitions from one partitioning to the next. It represents the ability of the partitioner to consider the existing distribution. This component is particularly important for SAMR applications as they require repartitioning at regular intervals. This metric component assumes greater significance for highly dynamic applications and scattered adaptations.
4. *Partitioning induced overhead*: This component attempts to capture the quality of the partitions generated by the partitioner, using the number of sub-grids and their aspect ratio. A large number of small grids may lead to a better load-balance, but can result in increased communication causing performance to deteriorate. Similarly, bad aspect ratio can adversely affect the computation-to-communication ratio, and can lead to bad memory access patterns.
5. *Partitioning time*: This component represents how fast the partitioning algorithm computes the new partitioning, given the current partitioning. It indicates the parallel efficiency of the partitioning algorithms employed. This component is particularly important for SAMR applications as they require repartitioning at regular intervals.

Optimizing all the above components implies conflicting objectives. For example, optimizing components (1) and (2) together constitutes an NP-hard problem. Partitioners typically optimize a subset of the components at the expense of others. Our goal in defining this metric is to be able to readily determine the trade-offs for each partitioning technique.

5. Design of the SAMR Simulator

This section presents the design and implementation of the SAMR simulator framework. The SAMR simulator is a tool for characterizing and evaluating partitioning techniques without having to run the actual application on a parallel computer. Initially, the application dumps a "trace" from a single processor run which is processed by a partitioner. The trace contains the state of the grid hierarchy (without any partitioning) at each stage of the adaptive application. The simulator reads the output from the partitioner and evaluates the quality of the partitioning in terms of the load imbalance, distribution quality, grid interaction overheads (inter-processor communication and memory copy), and data-movement overheads, as mentioned in Section 4.

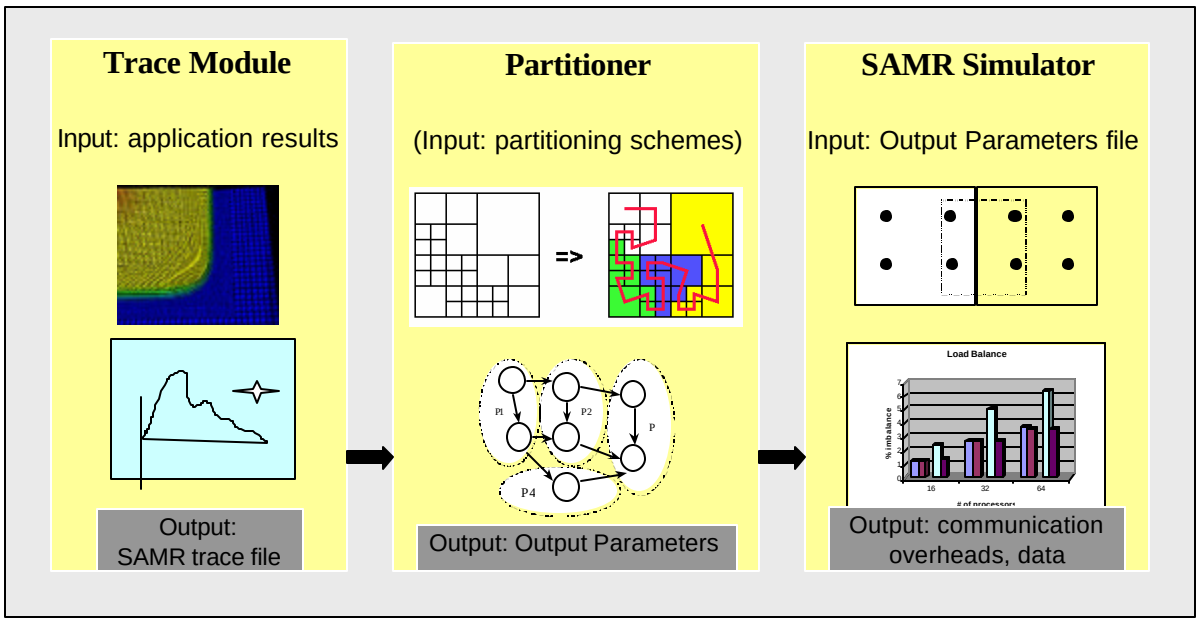

Figure 4: Conceptual view of the SAMR simulation framework

Figure 4 illustrates the conceptual view of the SAMR simulation framework [13]. The process can be summarized as follows:

1. A trace of the grid hierarchy is obtained once by performing an application run for a single processor. The resulting parameters are dumped into a "trace" file.
2. This trace file is the input to a partitioner. The partitioner implements the partitioning scheme of choice for a given number of processors. It allocates various bounding boxes to processors at different levels and time-steps. This result is dumped to an "output parameters" file.
3. This output parameters file is the input file to the SAMR simulator. The simulator measures the different partitioning quality parameters that characterize the distribution mechanisms.

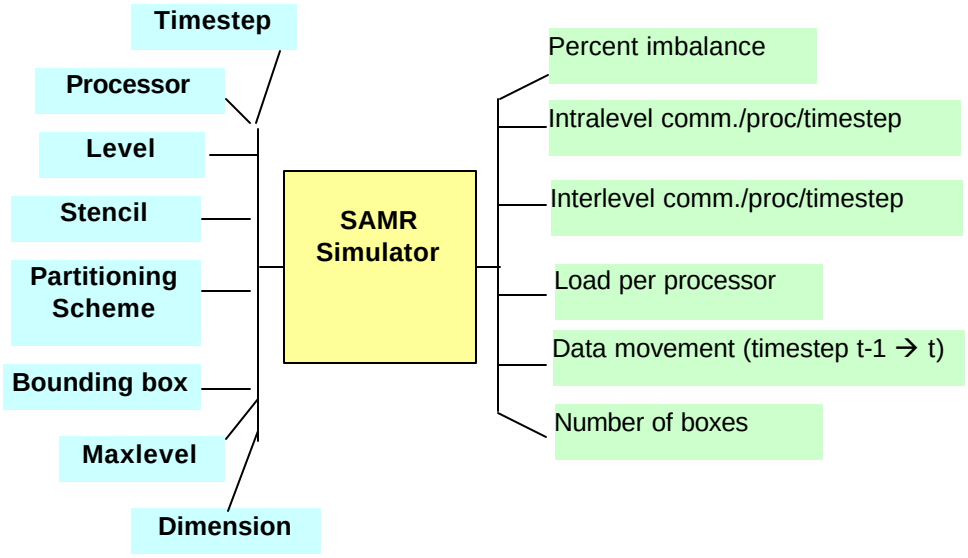

Figure 5: Inputs and outputs for the SAMR simulator

The input to the SAMR simulator is essentially a list of bounding boxes across processors at different levels of refinement of the grid hierarchy that are obtained as an output of the partitioning scheme. The user must specify the application rank (for example, 2-dimension or 3-dimension), maximum number of levels, and the number of processors through the command line interface. Additional inputs required by the simulator include the initial time step, time-stepping factor (the amount

by which a time step changes, for instance in steps of 1, 4, or 8, etc.), stencil size, and the filenames that store output values of the parameters being measured (communication overheads, data movement, load imbalance percentage, etc). The simulator reads in the time step, processor number, current refinement level, and the corresponding bounding box parameters from the input file, as illustrated in Figure 5. The SAMR simulator then creates a list of bounding boxes for each level per processor per time step. These lists are used to measure the intra-level and inter-level communication, the load imbalance created, the data movement involved, and the associated overheads. The simulator techniques for the measurement of these partitioning characterization criteria are presented in Section 6. The evaluation and analysis of partitioning characteristics for SAMR partitioners using the simulation framework is presented in Section 7.

6. Simulator Measurement of Partitioning Metric

6.1 Measurement of Intra-level Communication

At each time step, the intra-level communication is the amount of communication present between processors at the same level. This is measured as follows. From the input list, each bounding box (bb) at a particular refinement level is grown by a specified stencil size in all directions such that it maintains an overlapping zone at its boundaries with its neighboring processing elements (bb in this case) for synchronization of data during the updating process at the same level. This overlapping zone is called the "ghost region". The bounding box can be logically divided into three regions of interaction: face, corner and edge (depending on the dimension). Figure 6 illustrates the ghost regions for a bounding box and the ghost communications between two processors.

These interaction boxes for each bounding box are intersected with each of the remaining bounding boxes on the list at the same level. If there is an intersection between the two, there is indeed intra-level communication and it is measured by the size of the intersection of the two bounding boxes. The intra-level communication at a particular level is the sum of intra-level communication of all

bounding boxes in the list. The inputs to determine intra-level communication are: stencil size, list of bb at a particular time step, current level, current processor, application rank, maximum number of levels, number of processors, and current time.

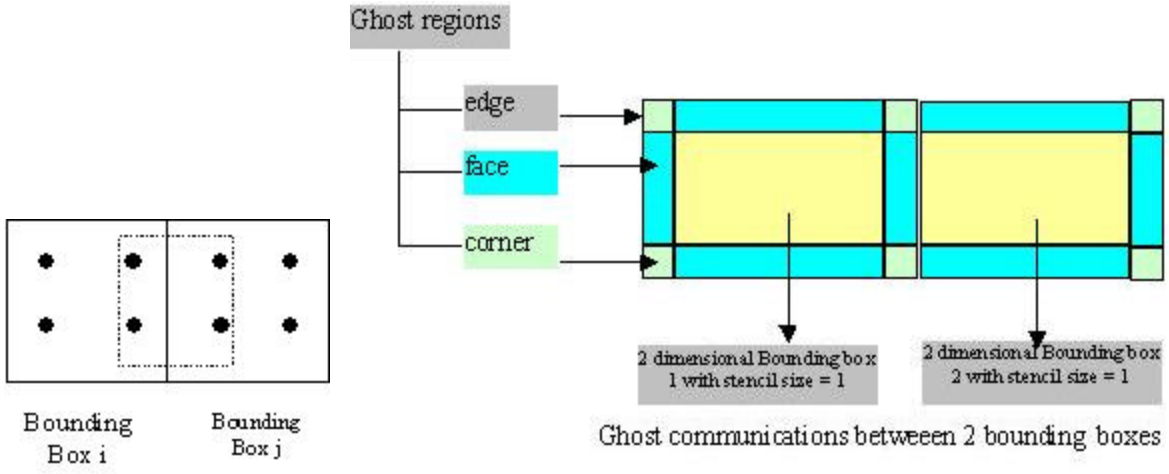

Figure 6: Bounding box ghost regions and ghost communications

Amount of intra-level communication = $\sum_{i=1}^{n} \text{intra-level communication of bounding box } i$

Now, the intra-level communication for bounding box *i* is the sum of the intra-level communication it has with all other bounding boxes. We can update the above equation as

Intra-level communication = $\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} \text{intra-level communication of bounding box } i \text{ with bounding box } j$

The amount of intra-level communication for bounding box *i* with bounding box *j* can be found by calculating half the size of intersection of the two bounding boxes. The calculation above needs to be adjusted for the stencil used by multiplying with a stencil factor, illustrations of which are shown in Figure 7.

Finally, the amount of intra-level communication can be expressed by the following equation:

Amount of intra-level communication = $\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} (\text{stencilFactor} \times \text{size}(\text{intersection}(bb_i, bb_j)) / 2)$

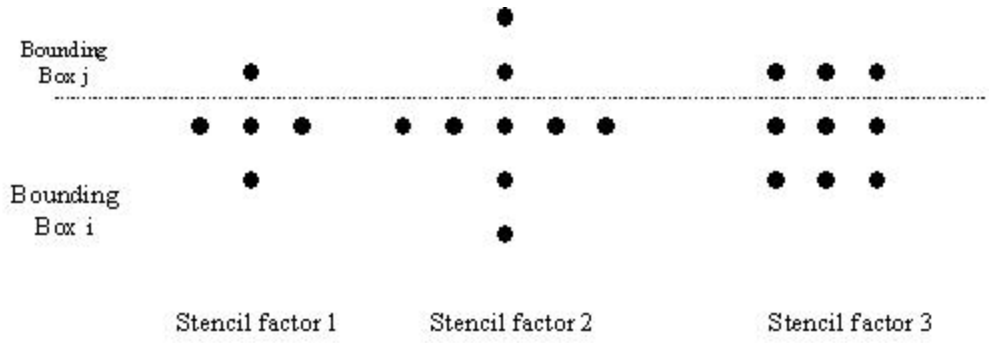

Figure 7: Stencil factors for SAMR intra-level communication

The algorithm implementing the measurement of intra-level communication is summarized as shown.

Algorithm for intra-level communication

```
At each timestep
        for (level=0 to level=max_levels) {
          for (proc=0 to proc=num_proc-1) {
                /* find out the intersection of each bounding box (bb) at this timestep, for this
                processor and level, with all the bbs that belong to all other processors at the
                same level. If there is an intersection between two such bbs, then there indeed
                is intra-level communication, else there is no intersection for this bb */
            for (each bb in bblist[timestep][processor][level] and each interaction in all directions) {
              find the interaction bb, which is given by the Ibbox routine of DAGHGhostinteraction
             intersect this with all bbs for other processors at the same level to obtain intersection bb
             if (intersection bb is != null)
                  intra-level communication += intersection_bb.size( )
        }} }
```

6.2 Measurement of Inter-level Communication

When a refined grid is created, its boundaries are usually interior to some coarser grid values. After updating the function values on a fine grid, the underlying coarse grid values are updated through restriction. This results in inter-level communication, which is measured across processors that contain intersecting bounding boxes at different levels of refinement for each time-step. The measurement process is as follows. At a given timestep, each bounding box from a list of bounding boxes for a given processor and level is compared to each bounding box from a bounding box list that is either one level greater (child) or one level below (parent). If there is an intersection between the 2 bounding boxes, then inter-level communication exists and is measured by the size of the intersection bounding box,

similar to the approach for measuring intra-level communication. The algorithm that determines the inter-level communication for a given processor and timestep is summarized as shown.

Algorithm for inter-level communication

```
Find out the total number of levels
If the number of levels > 1 then
        for (each level starting from level=1) {
                /* bounding boxes of level 0 have children at level 1,
                 we are concerned with the children */
          for (each bounding box at the higher level) {
                /* inter-level communication is twice the boundary points of itself,
                because both prolongation and restriction are involved */
            calculate the upper bound, lower bound, step size
            inter-level communication = 2* ((ub[x]-lb[x])/step[d]) + 1
        }}
```

6.3 Measurement of Data Migration

At the end of each timestep, the partitioning scheme redistributes different bounding boxes to different processors due to the change in the grid hierarchy as a result of refinement. Hence, the bounding boxes that belonged to one processor may or may not belong to the same processor at the next time step. Data movement is characterized by the migration of any such bounding boxes (number of data points) from one processor at one time step to another at the next time step after every regridding. The inputs to determine data movement are: two bounding box lists – one holding the boxes of the current timestep and the other holding the boxes of the previous timestep, number of processors, and the maximum number of levels. Data migration is measured as follows. At each time step, compare the lists of bounding boxes at the current and previous time steps. If there is an intersection between the 2 bounding boxes, then there exists data movement and is measured by the size of the intersection bounding box. The redistribution from time step t1 to time step t2 is given as:

Let the respective loads on the processors be load1, load2, ..., loadn at time step t1

Let the respective loads on the processors be load1a, load2a, ..., loadna at time step t2

Data movement for processor 1 = dm1 = load1 n load1a

Data movement for processor 2 = dm2 = load2 n load2a

:

Data movement for processor n = dmn = loadn n loadna

Thus, total data movement = dm1 + dm2 + ... + dmn

The algorithm to determine the data migration is summarized as shown.

Algorithm to determine data migration

```
At each timestep
for (processor=0 to processor=num_proc-1) {
        for (each box in bblPrev) {
           intersect with each box in bblCurrent to obtain the intersection_bb
          if (intersection_bb != NULL)
             data movement+= intersecton_bb.size( )
}}
```

6.4 Measurement of Load Imbalance

The load on each processor is computed as the workload for its contained regions at a certain level of refinement. Considering the refinement in space and time (with a refinement factor of 2) and the size of the bounding box (bb) as the workload parameter, the work for a processor is given by

Actual load = Work = bb.size() * (2 ^ level) …. for all bounding boxes for the processor

Sum of the loads on all the bounding boxes of all the processors gives the total workload. Ideal load on each processor is the average of the total workload across all processors.

Total load + = Work ….. for all processors

Ideal load on each processor = Total load /num_procs …. num_procs = number of processors

The load imbalance on each processor at a timestep can then be calculated as

Load Imbalance = ((Actual load - Ideal load) / Ideal load) * 100 %

The following algorithm summarizes the process of determining the load imbalance, using the number of time steps, average work, current processor, and the number of processors as inputs.

Algorithm to determine load imbalance

Sum of loads on all the bounding boxes of all the processors

$$Total\ load\ +\!= \sum_{i=0}^{nproc-1} \sum_{j=1, i \neq j}^{n} (\ \text{size}(bb_i))$$

load on each processor, i.e. average work = Total load / num_procs
for (each overloaded processor) {
difference in work = processor work - average work
total difference in work += difference in work
average difference in work = total difference in work/num_procs
*percent imbalance = (average difference in work*100)/average work*
cumulative percent imbalance += percent imbalance
final percent imbalance = cumulative percent imbalance/number of timesteps
}

6.5 Measurement of Intra-level and Inter-level Memory Copies

This is the amount of grid memory copies for co-located grid components. It is determined in the same way as intra- or inter-level communication except that at each time step, the communication overheads within each processor are considered.

6.6 Number of Boxes

Partitioning overheads in terms of number of boxes can be computed as the number of boxes in the bounding box lists associated with each processor at a given time step. This is equal to the number of lines obtained in the "output parameters" file that is obtained as an output from the partitioner and is the input to the SAMR simulator.

7. Evaluation of Partitioning Characteristics using the SAMR Simulator

This section presents a manual analysis of the partitioner characteristics for an illustrative SAMR application evaluated using the SAMR simulator. Based on this analysis, the adaptation policies are manually formulated and are used to adaptively select and invoke the most suitable partitioner from among the available SFC, G-MISP+SP, and pBD-ISP domain-based partitioners, belonging to GrACE [14, 15] and Vampire [16] software infrastructures. The SAMR application used in this evaluation is a basic, rudimentary version of the RM3D compressible turbulence kernel ($RM3D_b$) solving the 3-D

Richtmyer-Meshkov instability. The Richtmyer-Meshkov instability is a fingering instability which occurs at a material interface accelerated by a shock wave. This instability plays an important role in studies of supernova and inertial confinement fusion.

Application characterization consists of two steps. First, the adaptive behavior of the application is captured in an adaptation trace generated using a single processor run. The adaptation trace contains snap-shots of the SAMR grid hierarchy at each regrid step. This trace is then analyzed using the octant approach (in this case, manually) and the adaptive partitioning strategy is defined. The application is executed for 800 coarse level time steps and the trace consists of over 200 snap-shots. A selection of these snap-shots is shown in Figure 2 and illustrates the dynamics of $RM3D_b$ application.

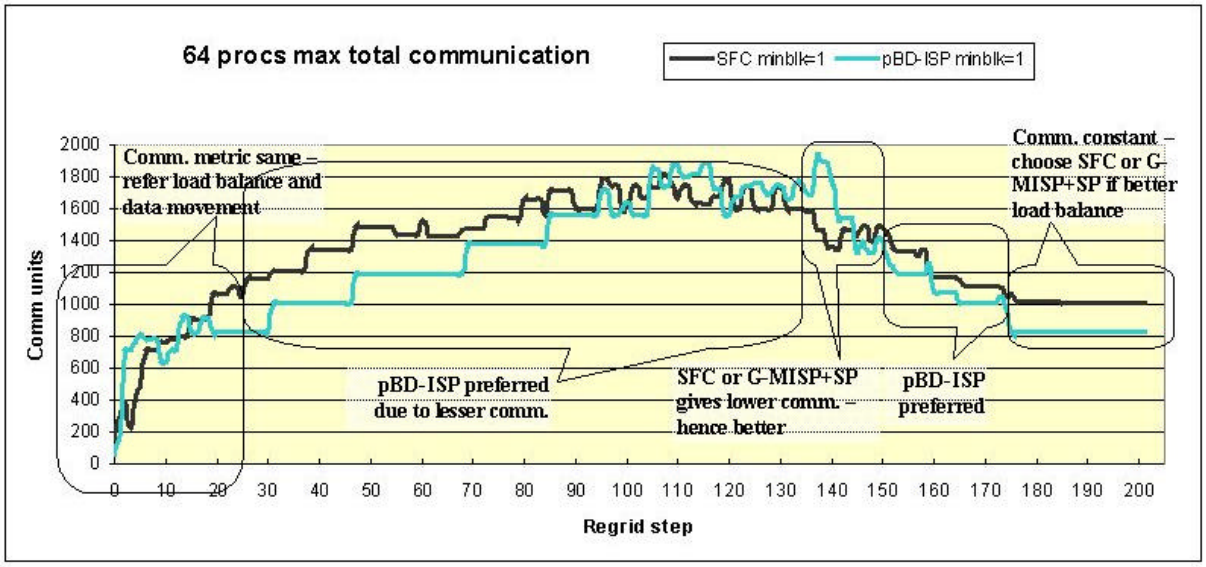

Figure 8: Total communication analysis for SAMR partitioners using the simulator on 64 processors

Once the $RM3D_b$ application trace containing snap-shots of the SAMR grid hierarchy is obtained, the SAMR simulator is used to analyze the behavior of a partitioner for the application on 64 processors [17]. The SAMR simulator uses the trace to determine the performance of the partitioners at each regrid step in terms of the primary components of the quality metric (communication, data movement, load imbalance). Note that the other two metric, partitioning overhead and partitioning time, are intrinsic characteristics of a partitioner. The partitioning time is directly dependent on the three

primary components that determine the runtime performance of the partitioner. Figures 8, 9, and 10 plot the maximum values of the total communication, data movement, and load imbalance components of the quality metric, respectively, for SFC and pBD-ISP partitioners on 64 processors.

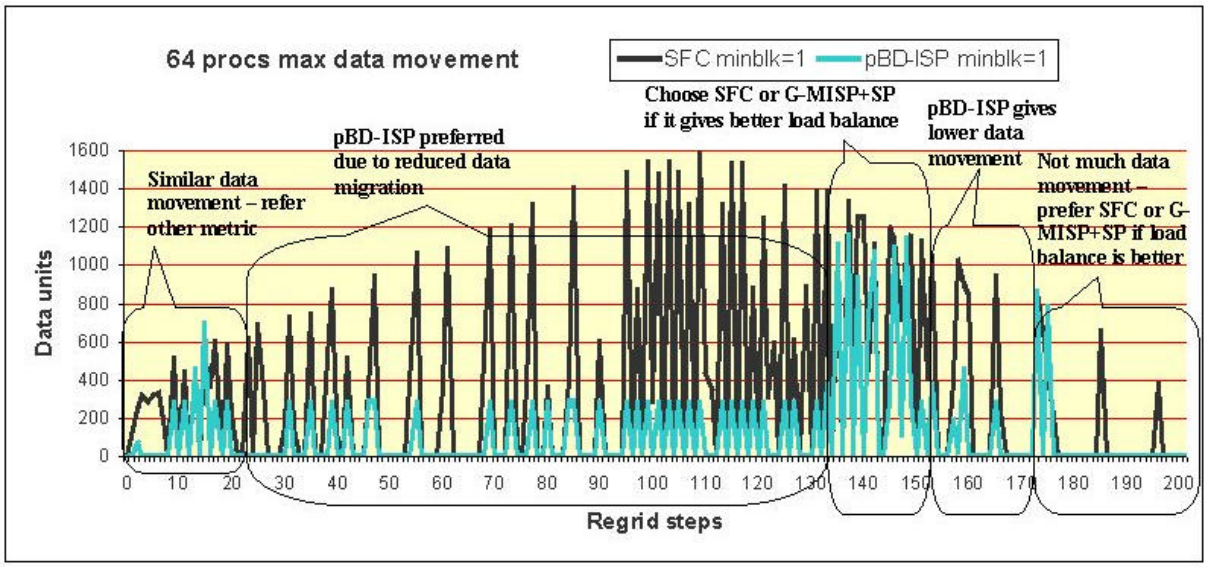

Figure 9: Data migration analysis for SAMR partitioners using the simulator on 64 processors

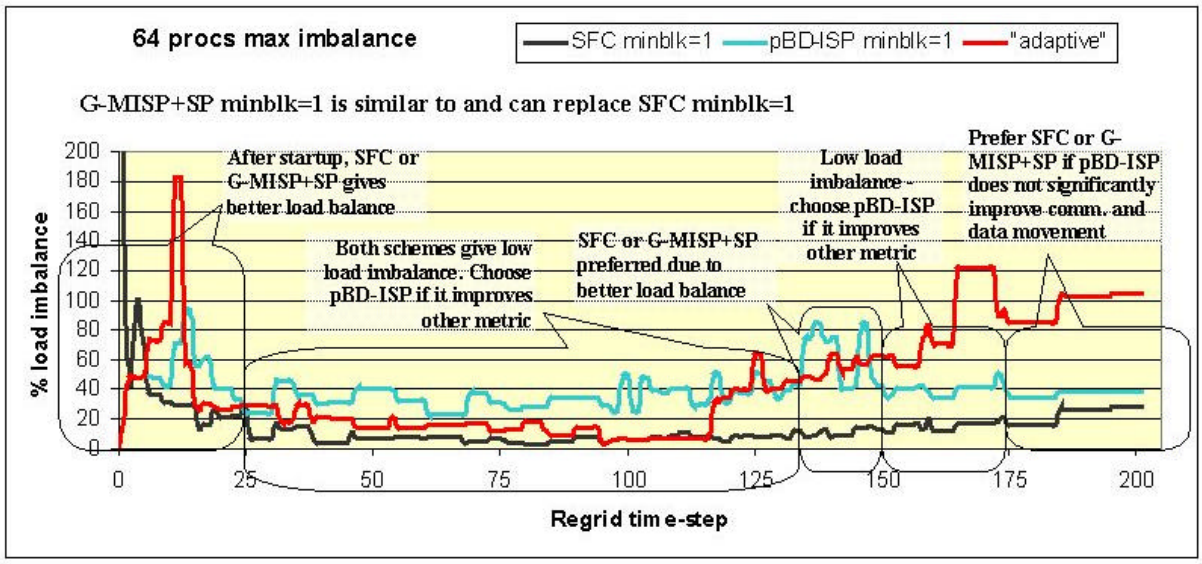

Figure 10: Load imbalance analysis for SAMR partitioners using the simulator on 64 processors

An application-sensitive partitioner adaptation policy for $RM3D_b$ can be formulated based on the partitioner characterization using the simulator analysis. The regrid step can be used as the switching point where a new partitioner is selected, configured, and invoked.

8. Conclusion

This report presents a simulation framework that requires a post-partitioning trace as input and evaluates the runtime characteristics of SAMR applications in terms of the quality metric – data migration, intra- and inter-level communication, and load imbalance. The simulation framework serves as a useful tool for evaluating the runtime behavior and performance of different SAMR partitioners.

The SAMR simulator uses the lists of bounding boxes obtained as an output from the partitioning stage as the input to the simulation framework and measures the intra-level and inter-level communication, the load imbalance created, the data movement involved, and the associated overheads. The simulator has been used to evaluate the characteristics of domain-based SAMR partitioning techniques for a variety of parallel, dynamic SAMR applications. Such an evaluation using the SAMR simulator has helped to better understand the runtime partitioning requirements of dynamic applications [18] and match partitioner behavior to corresponding application state, thereby enabling a feasibility study [19] for adaptive, application-sensitive partitioning. These experiments and evaluations have helped to improve the overall runtime performance of SAMR applications [20].

References

[1] E. Steinthorsson and D. Modiano, "Advanced Methodology for Simulation of Complex Flows using Structured Grid Systems", *ICOMP*, 28, 1995.

[2] M. Berger, G. Hedstrom, J. Oliger, and G. Rodrigue, "Adaptive Mesh Refinement for 1-Dimensional Gas Dynamics", *Scientific Computing* (IMACS/North Holland Publishing), pp. 43-47, 1983.

[3] R. Pember, J. Bell, P. Colella, W. Crutchfield, and M Welcome, "Adaptive Cartesian Grid Methods for Representing Geometry in Inviscid Compressible Flow", *11th AIAA Computational Fluid Dynamics Conference*, Orlando, FL, July 6-9, 1993.

[4] M. Choptuik, "Experiences with an Adaptive Mesh Refinement Algorithm in Numerical Relativity", *Frontiers in Numerical Relativity*, Cambridge University Press (London), editors: C. Evans, L. Finn, and D. Hobill, pp. 206-221, 1989.

[5] S. Hawley and M. Choptuik, "Boson Stars Driven to the Brink of Black Hole Formation", Phys. Rev. D 62:104024, 2000.

[6] G. Bryan, "Fluids in the Universe: Adaptive Mesh Refinement in Cosmology", *Computing in Science and Engineering*, pp. 46-53, March-April 1999.

[7] M. Norman and G. Bryan, "Cosmological Adaptive Mesh Refinement", *Numerical Astrophysics*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[8] M. Parashar, J. Wheeler, G. Pope, K. Wang, and P. Wang, "A New Generation EOS Compositional Reservoir Simulator: Part II - Framework and Multiprocessing", *Society of Petroleum Engineering Reservoir Simulation Symposium*, Dallas, TX, June 1997.

[9] P. Wang, I. Yotov, T. Arbogast, C. Dawson, M. Parashar, and K. Sepehrnoori, "A New Generation EOS Compositional Reservoir Simulator: Part I - Formulation and Discretization", *Society of Petroleum Engineering Reservoir Simulation Symposium*, Dallas, TX, June 1997.

[10] M. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations", *Journal of Computational Physics*, Vol. 53, pp. 484-512, 1984.

[11] S. Chandra, X. Li, and M. Parashar, "Engineering an Autonomic Partitioning Framework for Grid-based SAMR Applications", "High Performance Scientific and Engineering Computing: Hardware/Software Support", Kluwer Academic Publishers, pp. 169-187, March 2004.

[12] J. Steensland, S. Chandra, and M. Parashar, "An Application-Centric Characterization of Domain-Based SFC Partitioners for Parallel SAMR", *Transactions on Parallel and Distributed Systems*, IEEE Computer Society Press, Vol. 13:12, pp. 1275-1289, December 2002.

[13] M. Shee, "Evaluation and Optimization of Load Balancing/Distribution Techniques for Adaptive Grid Hierarchies", M.S. Thesis, Graduate School, Rutgers University, NJ, 2000.

[14] M. Parashar, GrACE homepage: www.caip.rutgers.edu//TASSL/Projects/GrACE.

[15] M. Parashar and J.C. Browne, "On Partitioning Dynamic Adaptive Grid Hierarchies", *29th Annual Hawaii International Conference on System Sciences*, pp. 604-613, January 1996.

[16] J. Steensland, Vampire homepage: http://www.caip.rutgers.edu/~johans/vampire, 2000.

[17] S. Chandra, J. Steensland, and M. Parashar, "An Experimental Study of Adaptive Application Sensitive Partitioning Strategies for SAMR Applications", Research poster presentation at *Supercomputing Conference*, Denver, CO, November 2001. Judged as best research poster.

[18] S. Chandra and M. Parashar, "An Evaluation of Partitioners for Parallel SAMR Applications", *Euro-Par 2001*, Springer-Verlag Lecture Notes in Computer Science, editors: R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, Vol. 2150, pp. 171-174, August 2001.

[19] S. Chandra, J. Steensland, M. Parashar, and J. Cummings, "An Experimental Study of Adaptive Application Sensitive Partitioning Strategies for SAMR Applications", *2nd Los Alamos Computer Science Institute Symposium*, October 2001.

[20] S. Chandra and M. Parashar, "Towards Autonomic Application-Sensitive Partitioning for SAMR Applications", accepted for publication in the *Journal of Parallel and Distributed Computing*, December 2004.