

Dynamic Load Balancing for Structured Adaptive Mesh Refinement Applications *

Zhiling Lan, Valerie E. Taylor
Department of Electrical and Computer Engineering
Northwestern University, Evanston, IL 60208
{zlan, taylor}@ece.nwu.edu

Greg Bryan
Massachusetts Institute of Technology
Cambridge, MA 02139
gbryan@mit.edu

Abstract

Adaptive Mesh Refinement (AMR) is a type of multiscale algorithm that achieves high resolution in localized regions of dynamic, multidimensional numerical simulations. One of the key issues related to AMR is dynamic load balancing (DLB), which allows large-scale adaptive applications to run efficiently on parallel systems. In this paper, we present an efficient DLB scheme for Structured AMR (SAMR) applications. Our DLB scheme combines a grid-splitting technique with direct grid movements (e.g., direct movement from an overloaded processor to an underloaded processor), for which the objective is to efficiently redistribute workload among all the processors so as to reduce the parallel execution time. The potential benefits of our DLB scheme are examined by incorporating our techniques into a parallel, cosmological application that uses SAMR techniques. Experiments show that by using our scheme, the parallel execution time can be reduced by up to 47% and the quality of load-balancing can be improved by a factor of four.

1 Introduction

Adaptive Mesh Refinement (AMR) is a type of multiscale algorithm that achieves high resolution in localized regions of dynamic, multidimensional numerical simulations. It shows incredible potential as a means of expanding the tractability of a variety of numerical experiments and has been successfully applied to model multiscale phenomena in a range of disciplines, such as computational fluid dynamics, computational astrophysics, meteorological simulations, structural dynamics, magnetics, and thermal dynamics. The adaptive structure of AMR applications, however, results in load imbalance among processors on parallel and distributed systems. Dynamic load balancing (DLB)

is an essential technique to solve this problem. In this paper, we present a new DLB scheme that combines a *grid-splitting* technique with direct grid movements. We illustrate the advantages of this technique with a real cosmological application that uses Structured AMR (SAMR) algorithm developed by Marsha Berger et al. [1] in the 1980s.

With any DLB scheme, the major issues to be addressed are the identification of overloaded versus underloaded processors, the quantity of data to be transferred from the overloaded processor to the underloaded processor, and the overhead that the DLB scheme imposes on the application. In investigating DLB schemes, we first analyzed the requirements imposed by the applications. In particular, we completed a detailed analysis of an SAMR application, the ENZO cosmological application, and determined the unique characteristics that impose several challenges on DLB schemes. ENZO, developed by G. Bryan and M. Norman [3], is one of the successful parallel implementations of SAMR for astrophysical and cosmological applications on distributed-memory systems. It entails solving the coupled equations of gas dynamics, collisionless dark matter dynamics, self-gravity, and cosmic expansion in three dimensions and at high spatial resolution. ENZO was developed as a community code and is currently in use in over different sites.

The results of the detailed analysis of ENZO provided four unique characteristics relating to DLB requirements: (1) coarse granularity, (2) high dynamicity, (3) high imbalance and different dispersion, and (4) an implementation that maintains some global information. First, the basic entity of SAMR applications is a "grid", which has a minimum size requirement, usually in the range of dozens of kilobytes. Often, the basic entity, the grid, is much larger than this minimum size. Thus the granularity, the size of basic entity for data movement, is coarse. Second, there is high dynamicity whereby the frequency of adaptations is high (usually every 3.5 seconds or less for a 5000+ simulation) and the data structures required to manage the dynamic grid hierarchy are complex. For example, linked-lists are used to manage the adaptations to avoid the overhead of reallocating arrays or using very large arrays with wasted

*Zhiling Lan is supported by a grant from the National Computational Science Alliance (ACI-9619019), Valerie Taylor is supported in part by a NSF NGS grant (EIA-9974960), and Greg Bryan is supported in part by a NASA Hubble Fellowship grant (HF-01104.01-98A).

space. Third, the amount of refinement is in the range of zero and 90% with each adaptive process, which results in a high imbalance among all the processors. Further, the distribution of imbalance varies with different datasets; it may occur in several localized regions or in a continuous way. Fourth, and lastly, ENZO employs a global method to manage the dynamic grid hierarchy, that is, each processor stores a small amount of grid information about the other processors. This information can be used by a DLB to aid in balancing the load.

Dynamic load balancing has been intensively studied for more than ten years and a large number of schemes have been presented to date [5, 6, 7, 8, 9, 10, 12, 13, 14]. Each of these schemes can be classified as either *Scratch-and-Remap schemes*[11] or *Diffusion-based schemes*[5, 8]. In [11], it was determined that *Diffusion-based schemes* generally provide better results for the problems in which imbalance occurs globally throughout the computational domain, while *Scratch-and-Remap schemes* are advantageous to the problems in which high magnitude imbalance occurs in localized regions. The third characteristic, high imbalance and different dispersion, however, implies that an appropriate DLB scheme should provide good load-balancing for both situations. Further, the second characteristic, the high frequency of adaptation and the use of complex data structures, results in *Scratch-and-Remap schemes* being intolerable because of the demand to completely modify the data structures without considering the previous load distribution. In a diffusion method, each processor "diffuses" fractions of its workload to its underloaded neighbors and receive workload from its overloaded neighbors simultaneously at each iteration step. Global balance is achieved by successive migration of workload from overloaded processors to underloaded processors. *Diffusion-based schemes* employs the neighboring information to redistribute the load between adjacent processors, thereby requiring multiple diffusive steps to achieve global load balance. The number of diffusive steps depends on the load distribution of applications. An efficient DLB for AMR must also address the first characteristic, the large grid sizes, to equalize the load among the processes.

Our DLB scheme combines a grid-splitting option with direct data movement. Basically, our scheme is composed of two phases: *moving-grid phase* and *splitting-grid phase*. The *moving-grid phase* utilizes the global information to send grids directly from overloaded processors to underloaded processors; only one communication is required to move a grid. The use of direct communication to move the grids eliminates the variability in time to reach the equal balance and avoids chances of *thrashing*[21]. The *splitting-grid phase* splits a grid into two smaller grids along the longest dimension, thereby addressing the first characteristic. Our DLB invokes the *moving-grid phase* after each

adaptation. This operation continues in parallel until no further movement can be done. Then the *splitting-grid phase* will be invoked if imbalance still exists. This sequence continues until the load is balanced within a given tolerance.

The efficiency of our DLB scheme on an SAMR application is measured using the execution time and the quality of load balancing. Our experiments show that integrating our DLB scheme into ENZO results in significant performance improvement. For example, the execution time of the *AMR64* dataset, a $32 \times 32 \times 32$ initial grid, on 32 processors is reduced by 47%, from 5683.68 seconds to 3011.44 seconds, and the quality of load-balancing is improved by a factor of four.

The remainder of this paper is organized as follows. Section 2 introduces SAMR algorithm and its parallel implementation ENZO. Section 3 analyzes the adaptive characteristics of SAMR applications. Section 4 describes our dynamic load balancing scheme. Section 5 introduces some load-balancing metrics followed by the experimental results exploring the impact of our DLB on the real SAMR applications. Section 6 describes related work and compares our scheme with some widely-used schemes. Finally, section 7 summarizes the paper and identifies our future work.

2 Overview of SAMR

This section gives an overview of the SAMR method, developed by M. Berger et al. and ENZO, a parallel implementation of this method for astrophysical and cosmological applications. Additional details about ENZO and the SAMR method can be found in [1, 2, 3, 4].

2.1 Layout of Grid Hierarchy

SAMR represents the grid hierarchy as a tree of grids at any instant of time. The number of levels, the number of grids, and the locations of the grids change with each adaptation. That is, a uniform mesh covers the entire computational volume and in regions that require higher resolution, a finer subgrid is added. If a region needs still more resolution, a even finer subgrid is added. This process repeats recursively with each adaptation resulting in a tree of grids like that shown in Figure 1. The top graph in this figure shows the overall structure after several adaptations. The remainder of the figure shows the grid hierarchy for the overall structure with the dotted regions corresponding to those that underwent further refinement. In this grid hierarchy, there are four levels of grids going from level 0 to level 3. Throughout execution of an SAMR application, the grid hierarchy changes with each adaptation.

For simplification, SAMR imposes some restrictions on the new subgrids. A subgrid must be uniform, rectangular, and aligned with its parent grid and must be completely

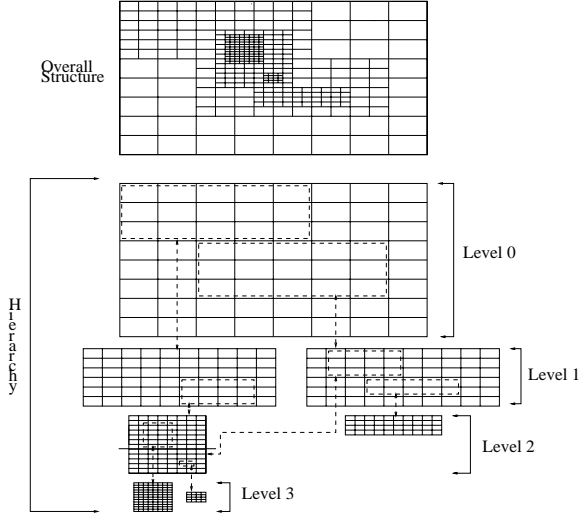


Figure 1. SAMR Grid Hierarchy

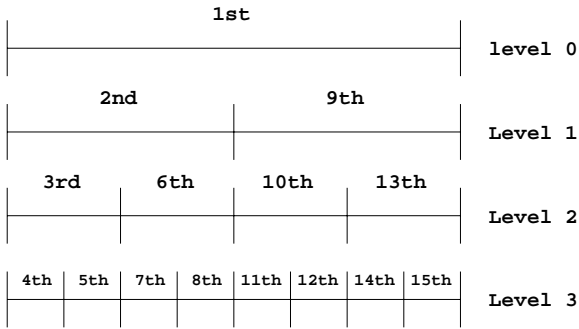


Figure 2. Integrated Execution Order (refinement factor = 2)

contained within its parent. All parent cells are either completely refined or completely unrefined. Lastly, the refinement factor must be an integer[3].

2.2 Integration Execution Order

The SAMR integration algorithm goes through the various adaptation levels advancing each level by an appropriate time step, then recursively advancing to the next finer level at a smaller time step until it reaches the same physical time as that of the current level. Figure 2 illustrates the execution sequence for an application with four levels and a refinement factor of 2. First we start with the first grid on level 0 with time step dt . Then the integration continues with one of the subgrids, found on level one, with time step $dt/2$. Next, the integration continues with one of the subgrids on level 2, with time step $dt/4$, followed by the analysis of the subgrids on level 3 with time step $dt/8$. The

```

Original DLB Algorithm

Done = FALSE;
while ( MaxLoad > Threshold * MinLoad ) && Done = FALSE) {
  for ( i = 0 ; i < NumberOfGrids; i++){
    if ( grid (i) resides on MaxProc && size (grid(i)) < (MaxLoad - MinLoad) / 2) {
      Move grid (i) from MaxProc to MinProc;
      Update load information of MaxProc and MinProc;
      Find new MaxProc (MaxLoad) and MinProc (MinLoad);
      Break;
    }
  }
  if (i == NumberOfGrids)
    Done = TRUE;
}

```

Figure 3. Pseudo-code of the original DLB scheme

figure illustrates the order for which the subgrids are analyzed with the integration algorithm.

2.3 ENZO: A Parallel Implementation of SAMR

Although the SAMR strategy shows incredible potential as a means for simulating multiscale phenomena and has been available for over two decades, it is still not widely used due to the difficulty with implementation. The algorithm is complicated because of dynamic nature of memory usage, the interactions between different subgrids and the algorithm itself. ENZO [3] is one of the successful parallel implementations of SAMR, which is primarily intended for use in astrophysics and cosmology. It is written in C++ with Fortran routines for computationally intensive sections and MPI functions for message passing among processors.

ENZO implementation employs a global way to the manage grid hierarchy; that is, each processor stores the grid information of all other processors. In order to save space and reduce communication time, the notation of a "real" grid and a "fake" grid is used for sharing grid information among processors. Each subgrid in the grid hierarchy resides on one processor and this processor holds the "real" subgrid. All other processors have the replications of this "real" subgrid, which is called "fake" grid. Usually, the "fake" grid contains the information such as dimensional size of the "real" grid, and the processor where the "real" grid resides. The data associated with a "fake" grid is small (usually a few hundred bytes), while the amount of data associated with a "real" grid is large (ranging from several hundred kilobytes to dozens of megabytes).

The current implementation of ENZO uses a simple DLB scheme that utilizes the previous load information (characteristic two) and the global information (characteristic four), but does not address the large grid sizes (characteristic one). For the original DLB scheme, if the load-balance ratio (defined as $MaxLoad/MinLoad$) is larger

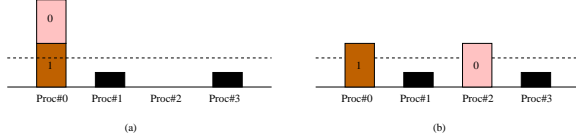


Figure 4. An Example of Load Movements using the Original DLB

than a hard-coded threshold, the load balancing process will be invoked. *MaxProc*, which has the maximal load, attempts to transfer its portion of grids to *MinProc*, which has the minimal load, under the condition that *MaxProc* can find a suitable sized grid for transferring. Here, the suitable size means the size is no more than half of the load difference between *MaxProc* and *MinProc*. Figure 3 gives the pseudocode of this scheme.

An example of grid movements that occurs with this DLB method is shown in Figure 4. In this example, there are four processors: processor 0 is overloaded with two large-sized grids 0 and 1, processor 2 is idle, and processor 1 and 3 are underloaded. The dash line shows the required load for which all the processors would have an equal load. After one step of movement with the original DLB, grid 0 is moved to processor 2 as shown in Figure 4 (b). At this point, the original DLB stops because no other grids can be moved. However, as the figure illustrates, the load is not balanced among the processors. Hence, the original DLB suffers from the problem of the coarse granularity of the grids.

3 Adaptive Characteristics of SAMR Applications

This section provides some experimental results illustrating the adaptive characteristics of SAMR applications running with ENZO implementation (discussed in Section 1). The experiments are analyzed from four aspects: *granularity* (characteristic one), *dynamicity* (characteristic two), *imbalance* and *dispersion* (characteristic three) [16]. All the figures shown in this section are obtained by executing ENZO without any DLB. This is done to demonstrate the characterization independent of any DLB.

Three real datasets (*AMR64*, *AMR128*, and *ShockPool3D*) are used in this paper. Both *AMR128* and *AMR64* are designed to simulate the formation of a cluster of galaxies; *AMR128* is basically involves a larger grid than *AMR64*. Both datasets create many grids randomly distributed across the computational domain. *ShockPool3D* is designed to simulate the movement of a shock wave (i.e., a plane) that is slightly tilted with respect to the edges of the computational domain. This dataset creates more and more grids along the

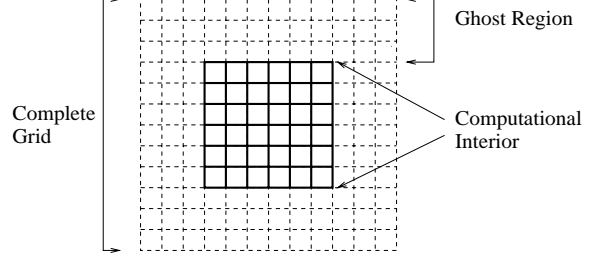


Figure 5. Components of Grids

moving shock wave plane. The sizes of these datasets are given in Table 1.

- *Granularity: Size of Basic Entity for Data Movement*

The basic entity for data movement is a grid. Each grid consists of a *computational interior* and a *ghost zone* as shown in Figure 5. The *computational interior* is the region of interest that has been refined from the immediately coarser level; the *ghost zone* is the part added to exterior of computational interior in order to obtain boundary information. For the *computational interior*, there is a requirement for the minimum number of cells, which is equal to the refinement ratio to the power of the number of dimensions. For example, for a 3-dimensional problem, if the refinement ratio is 2, then the computational interior will have at least $2^3 = 8$ cells. The default size for the ghost zones is set to 3, so each grid should have at least $(3 + 2 + 3)^3 = 512$ cells. However, the grids are often much larger than this minimum size. Usually, the amount of data associated with grids varies ranging from 100KB to 10MB. Thus the granularity of a typical SAMR application is very coarse, thereby making it very hard to achieve a good load balance by moving the basic entities.

- *Dynamicity: Frequency of Load Changes*

After each time-step of every level, the adaptation process is invoked based on one or more refinement criteria defined at the beginning of the simulation. The local regions satisfying the criteria will be refined. The number of adaptations varies for different datasets. For *ShockPool3D*, there are about 600 adaptations throughout the evolution. For some SAMR applications, more frequent adaptation is sometimes needed to get the required level of detail. For example, for the medium-sized dataset *AMR64* with initial problem size $32 \times 32 \times 32$ running on 32 processors, there are more than 2500 adaptations with the execution time of about 8500 seconds, which means the adaptation process is invoked every 3.4 seconds on average. For the larger dataset *AMR128*, there are more than 5000

Dataset	Initial Problem Size	Final Problem Size	Number of Adaptations
AMR64	$32 \times 32 \times 32$	$4096 \times 4096 \times 4096$	2500
AMR128	$64 \times 64 \times 64$	$8192 \times 8192 \times 8192$	5000
ShockPool3D	$50 \times 50 \times 50$	$6000 \times 6000 \times 6000$	600

Table 1. Three Experimental Datasets

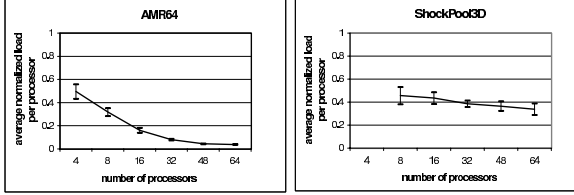


Figure 6. Average Normalized Load Per Processor

adaptations. High frequency of adaptation requires the underlying DLB method to execute very fast, as well as to maintain high quality of load balancing.

- *Load Imbalance*

Figure 6 shows the average load per processor and the standard deviation for *AMR64* and *ShockPool3D* respectively. The line shows the average load per processor and the bar on the line gives the standard deviation. The load shown in the figures is normalized to the maximal load. The ideal balanced load occurs when the average load is 1.0. The figure indicates that the average load is decreasing as the number of processors goes up. For *AMR64*, when the number of processors increases from 4 to 64, the average load decreases from 0.496 to 0.0394 and the standard deviation goes down from 0.126 to 0.005. For *AMR128*, the results are the similar to those of *AMR64*. For *ShockPool3D*, the average load decreases from 0.456 to 0.338 as the number of processors increases from 8 to 64. For both datasets, the standard deviation is pretty small compared to the average load, which means that the average load reflects the entire load distribution. For both cases, the average load is less than 0.4 when the number of processors is more than 16, which means that most of the processors are underloaded. Hence, a very high imbalance exists for the two datasets, with the imbalance increasing with increase in number of processors.

- *Dispersion: Load Distribution*

Figure 7 illustrates the difference of load distribution between *AMR64* and *ShockPool3D*. The x-axis represents the processor number, and the y-axis represents the percentage of refinement per processor. For *AMR64*, there are only a few processors whose loads

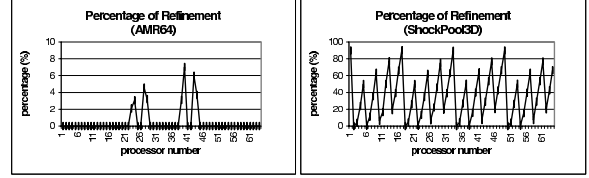


Figure 7. Percentage of Refinement for Each Processor using 64 processors

are increased dramatically and most processors have little or no change. For example, running *AMR64* on 64 processors, there are only 8 processors (processor number 22, 23, 26, 27, 38, 39, 42, 43) whose loads are increased while loads of all the other processors remain the same. As mentioned above, the dataset *AMR128* is essentially a bigger version of *AMR64*, so high imbalance occurs locally for both *AMR64* and *AMR128*. For *ShockPool3D*, the percentage of refinement per processor has some regular behavior: all the processors can be grouped into four subgroups and each subgroup has similar characteristics with the percentage of refinement ranging from zero to 86%. These figures indicate that different datasets exhibit different load distribution, and the underlying DLB scheme should provide high quality of load balancing for all these datasets.

4 Our DLB Scheme

After taking into consideration the adaptive characteristics of the SAMR application, we developed an improved DLB scheme. Our DLB is composed of two steps: *moving-grid phase* and *splitting-grid phase*. The *splitting-grid phase* splits the grid, along the longest dimension, into two subgrids. Figure 8 gives the pseudocode of our scheme. The details of each phase are given below.

- *Moving-Grid Phase*

After each adaptation, our DLB is triggered if the load is imbalanced, that is, $MaxLoad/AvgLoad > threshold$. The *MaxProc* moves its grid directly to *MinProc* under the condition that the computational load of this grid is no more than $(threshold \times$

```

Our DLB Algorithm
MoveFlag = 1; SplitFlag = 1; LastMax = 0; LastMin = 0;
while (MaxLoad > threshold * AvgLoad && MoveFlag == 1) { //moving-grid phase
  for (i=0; i < NumberOfGrids; i++) {
    if ( grid (i) resides on MaxProc && size(grid (i)) < ( threshold * AvgLoad - MinLoad)) {
      Move grid (i) from MaxProc to MinProc;
      Update load information of MaxProc and MinProc;
      Find new MaxProc (MaxLoad) and MinProc (MinLoad);
      Break;
    }
  }
  if ( i == NumberOfGrids)
    MoveFlag = 0;
}
while ( MaxLoad > Threshold * AvgLoad && SplitFlag == 1) { //split-grid phase
  Find the largest grid MaxGrid residing on MaxProc;
  if ( size(MaxGrid) <= (AvgLoad - MinLoad) ) {
    Move MaxGrid from MaxProc to MinProc;
  } else {
    Split MaxGrid into two by following the requirement of SAMR algorithm;
    Redistribute one of split grids to MinProc;
  }
  Update load information of MaxProc and MinProc;
  LastMax = MaxProc; LastMin = MinProc;
  Find new MaxProc (MaxLoad) and MinProc (MinLoad);
  if (LastMax == MaxProc && LastMin == MinProc)
    SplitGrid = 0;
}
}

```

Figure 8. Pseudo-code of our DLB scheme

$AvgLoad - MinLoad$). $AvgLoad$ denotes the required load for which all the processors would have an equal load. Thus our DLB does not make any underloaded processor become overloaded. This phase continues until either the load-balancing ratio is satisfied or all grids residing on the $MaxProc$ are too large to be moved. Further, this phase uses the global information to move a grid directly from the $MaxProc$ to $MinProc$ in one communication step.

Note that no sorting process is performed in this phase, so the first grid on $MaxProc$ whose size satisfies the requirement will be moved to $MinProc$. Further, this phase differs from the original DLB scheme (Figure 3) from several aspects. First, from Figure 4, it is observed that the original DLB may cause the previous underloaded processor (processor 2) to be overloaded by solely moving grids from overloaded processors to underloaded processors. This phase overcomes this problem by making sure that any grid movement will make an underloaded processor reach, but not exceed, the average load. Second, in this phase, a different metric ($MaxLoad/AvgLoad > threshold$) is used to measure load imbalancing. This metric is more accurate than the metric ($MaxLoad/MinLoad > threshold$) used in the original DLB scheme. For example, suppose two cases, the loads of the first case are (20, 8, 8, 8, 8, 8) and the second are (12, 12, 12, 12, 12, 0). The second distribution is preferred over the first case because the maximum runtime is less. By setting $threshold$ to 1.50, this *moving-*

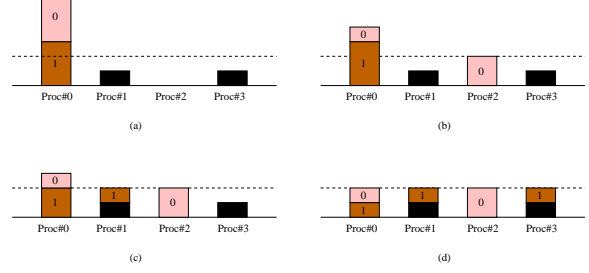


Figure 9. An Example of Load Movements using our DLB

grid phase will be entailed for the first case. However, by using the original DLB with $threshold$ set to 2.50, load-balancing process will be invoked for the second case, but not for the first case.

- *Splitting-Grid Phase*

After the *moving-grid phase*, if imbalance still exists, this phase will be invoked. First, the $MaxProc$ finds the largest grid it owns (denoted as $MaxGrid$). If the size of $MaxGrid$ is no more than $(AvgLoad - MinLoad)$ which is the amount of load needed by $MinProc$, the grid will be moved to $MinProc$ from $MaxProc$; otherwise, $MaxProc$ splits this grid along the longest dimension into two smaller grids. One of the two split grids, whose size is around $(AvgLoad - MinLoad)$, will be redistributed to $MinProc$. After such a splitting step, $MinProc$ may reach the average load. Further splitting steps attempt to make other underloaded processors reach the average load. Eventually, either the load is balanced, which is our goal, or we reached the minimum allowable grid size.

Note that both the *moving-grid phase* and *splitting-grid phase* execute in parallel. For example, in the *moving-grid phase*, when processor 0 moves one of its grids to processor 5, all the other processors continue on the *moving-grid phase*. If new $MaxProc$ and $MinProc$ are processor 1 and 2 respectively, then processor 1 will move one of its grids to processor 2 and this process is overlapped with the movement from processor 0 to processor 5. The same overlapping occurs in the *splitting-grid phase*.

To illustrate the use of our DLB, versus the original DLB, we use the same example given in Figure 4. For this example, the grid movements of our DLB is shown in Figure 9. The two grids on overloaded processor 0 are larger than $(threshold \times AvgLoad - MinLoad)$, so there is no work done in the *moving-grid phase* and *splitting-grid phase* begins. First, grid 0 is split into two smaller grids and one of them is transferred to processor 2. Then grid 1 is split into two and one of them is moved to processor 1. Since

grid 1 residing on processor 0 is still large enough, it is split again and one of them is migrated to processor 3. As we can observe, compared with the grid movements of the original DLB shown in Figure 4, our DLB combines the grid-splitting technique with direct grid movements, thereby improving the load balance.

5 Experimental Results

The potential benefits of our DLB scheme are examined by executing real SAMR applications running ENZO on parallel systems. All the experiments were executed on the 195 MHz R10000 SGI Origin2000 machines at NCSA¹. The code was instrumented with performance counters and timers, which do not require any I/O.

5.1 Load Balancing Metrics

The effectiveness of our DLB scheme is measured by both the execution time and the quality of load balancing. First, the following metrics are proposed to measure the quality of load balancing:

- *Average Load* is the average of the normalized load among all the processors for all of the adaptations:

$$m_1 = \frac{\sum_{j=1}^N \frac{\sum_{i=1}^P L_i(j)}{P}}{N} = \frac{\sum_{j=1}^N \frac{AvgLoad(j)}{MaxLoad(j)}}{N}$$

Where N is number of adaptations, P is number of processors, and $L_i(j)$ is normalized to the maximal load for the i th processor for the j th adaptation. The closer m_1 is to 1.0 the better; the value of 1.0 implies equal load distribution among all the processors.

- *Average Standard Deviation of Average Load* is defined as:

$$m_2 = \frac{\sum_{j=1}^N \sqrt{\frac{\sum_{i=1}^P (L_i(j) - m_1(j))^2}{P-1}}}{N}$$

By definition, the capacity to keep m_2 low during the execution is one of the main quality metrics for an efficient DLB. The capacity to have m_2 a small fraction of m_1 indicates an efficient DLB.

- *Average Percentage of Idle Processors* is defined as

$$m_3 = \frac{\sum_{j=1}^N p(j)}{N}$$

Where $p(j)$ is the percentage of idle processors for the j th adaptation. As mentioned above, due to the coarse

granularity of SAMR applications, it is possible that there may be some idle processors for each iteration. Obviously, the smaller this metric is, the better load balancing is.

5.2 Total Execution Time

Table 2 summarizes the total execution times with varying numbers of processors by using our DLB and the original DLB. It is observed that our DLB greatly reduces the execution time, especially when the number of processors is more than 16. The relative improvements of execution time are as follows: between 8.5% and 47% for *AMR64*, between -4.8% and 20.2% for *AMR128*, and between 9.6% and 26.1% for *ShockPool3D*. For all the datasets, the largest improvement occurs with 32 processors. However, we may notice that there are two exceptions. When executing *AMR128* on 8 or 16 processors, our DLB has worse performance compared to the original DLB. The reason is that our DLB tries to improve the load balance by using grid-splitting technique, which entails some communication and computation overheads. For example, more smaller grids are introduced across processors which require more communications to transfer data among processors. Further, more computational load is added because each grid requires a "ghost zone" to store boundary information. When the number of processors is not large and the original DLB provides relatively good load balancing, the overheads introduced by our scheme may be larger than the gain provided by using our scheme. When the original DLB is not efficient, especially when the number of processors is larger than 16, our DLB is able to redistribute load more evenly among all the processors, thereby utilizing the computing resource more efficiently so as to improve the overall performance. The last row of the table shows the average relative improvement over all the datasets with different number of processors. Hence, on average, our DLB provides an improvement over the original DLB, especially when the number of processors is larger than 16.

5.3 Quality of Load Balancing

The first load-balancing metric *Average Load* is given in Figure 10. It indicates that the average load is decreasing with increasing number of processors by using either of two methods. This results because it is more likely that there are not enough grids to be redistributed among processors if there are more processors. Our scheme, however, is able to significantly improve this metric by splitting large-sized grids for all cases. Further, the amount of improvement gets larger as the number of processors increases. In general, the relative improvement of the *Average Load* metric ranges between 23% and 605% by using our DLB. In

¹National Center for Supercomputing Applications, Urbana, IL

Dataset	8 procs	16 procs	32 procs	48 procs	64 procs
AMR64 (Original DLB)	5466.97	5429.76	5683.68	4999.11	5098.19
AMR64 (Our DLB)	5004.32 (8.46%)	3561.32 (34.41%)	3011.44 (47.02%)	3078.38 (38.42%)	3474.83 (31.84%)
AMR128 (Original DLB)	53111.15	27919.77	23239.27	18821.85	17083.73
AMR128 (Our DLB)	55683.05 (-4.84%)	28550.15 (-2.26%)	18552.89 (20.17%)	17101.82 (9.14%)	15798.05 (7.53%)
ShockPool3D (Original DLB)	16188.88	9315.85	5823.76	4530.50	4303.47
ShockPool3D (our DLB)	14636.63 (9.59%)	8374.81 (10.10%)	4301.76 (26.13%)	3396.65 (25.03%)	3331.12 (22.59%)
Average Relative Improvement	4.4%	14.1%	31.1%	24.2%	20.7%

Table 2. Total Execution Time

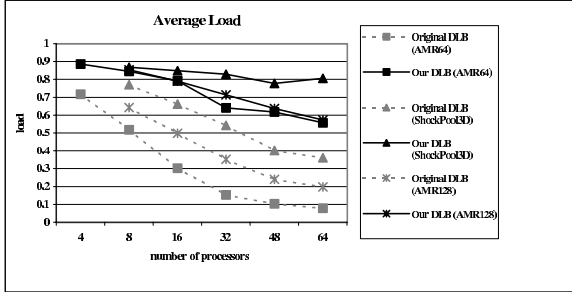


Figure 10. Average Load (normalized to the maximal load)

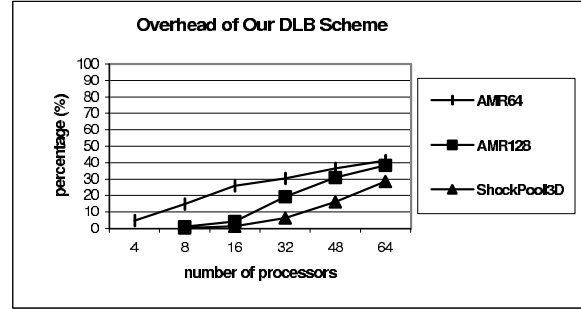


Figure 12. Overhead of our DLB

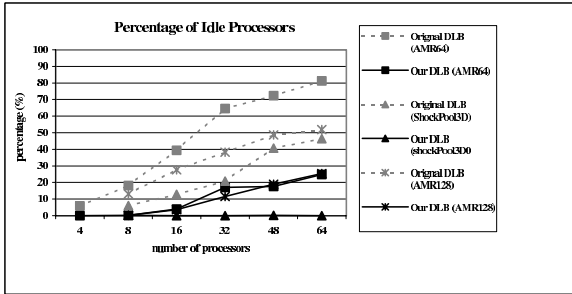


Figure 11. Percentage of Idle Processors

all cases, the average load for our DLB was always greater than 0.5, which is significant. Our results of the second load-balancing metric (not shown) indicate the standard deviation of this metric is quite low (less than 0.096), which means *Average Load* metric can truly represent the load distribution of all the processors.

Average Percentage of Idle Processors is shown in Figure 11. The average percentage increases as the number of processors increases for both methods. As mentioned above, it is due to the fact that there are not enough grids for movement when there are more processors. However, the percentage of idle processors is much less by using our DLB; the percentage ranges from zero to approximately

25% for our DLB, as compared to 5.9% to 81.2% for the original DLB. Larger percentage of idle processors means more computing resources are wasted.

5.4 Overhead of Our DLB Scheme

The overhead of our DLB with respect to the total execution time is summarized in Figure 12. It shows that the overhead is increasing from 4.7% to 41.2% as the number of processors increases. The most time-consuming portion of our DLB is the communication time to share the "fake" grid information among all processors after each *splitting-grid* phase. This overhead is due to the fact that ENZO employs a global method to manage the adaptive grid hierarchy.

6 Related Work

Grid-splitting is a well-known technique and has been applied in several research works[17, 18, 19, 20, 22]. J. Rantakokko uses this technique in his static load balancing scheme [15] which is based on Recursive Spectral Bisection. The main purpose of Rantakokko's static scheme is to efficiently divide the computational domain and reuse a solver for a rectangular domain. In our scheme, the grid-splitting technique is combined with direct grid movements to provide an efficient dynamic load balancing

scheme. Here, grid-splitting is used to reduce the granularity of data moved from overloaded to underloaded processors, thereby resulting in equalizing load throughout execution of the SAMR application.

Our DLB is not a *Scratch-Remap Scheme* because it takes into consideration the previous load distribution during the current redistribution process. As compared to *Diffusion Scheme*, our DLB scheme differs from it in two manners. First, our DLB scheme addresses the issue of coarse granularity of SAMR applications. It splits large-sized grids located on overloaded processors if just the movement of grids is not enough to handle load imbalance. Second, our DLB scheme chooses the direct data movement between overloaded and underloaded processors instead of just between neighboring processors.

7 Summary

In this paper, we proposed a dynamic load balancing scheme for SAMR applications. Our DLB scheme includes two phases: *moving-grid phase* and *splitting-grid phase*. The potential benefits of our scheme are examined by incorporating our DLB into a real SAMR application, ENZO. The experiments show that our scheme can significantly improve the quality of load balancing and reduce the total execution time, compared to the original DLB. By using our DLB, the total execution time of SAMR applications was reduced up to 47%, and the quality of load balancing was improved by more than two times especially when the number of processors is larger than 16. While the focus of this paper is on SAMR, the techniques can be easily extended to other SAMR applications. For example, for unstructured AMR applications, if some global load information is provided, we can perform load balancing by direct load movement between overloaded and underloaded processors; if the basic entity for data movement is large and solely performing grid movements among processors cannot obtain satisfying load balancing, a grid-splitting technique can be exploited and the largest grid on an overloaded processor can be split along the longest edge.

Future work includes decreasing the communication overhead introduced by our scheme, sensitivity analysis of parameters (such as threshold) used in our scheme, and extending our work to distributed systems which is composed of heterogeneous processors and networks.

References

- [1] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. In *Journal of Computational Physics*, 82(1):64–84, May 1989.
- [2] M. Norman, J. Shalf, S. Levy, and G. Bryan. Diving deep: Data-management and visualization strategies for adaptive mesh refinement simulations. In *Computing in Science and Engineering*, 1(4):36–46, July/August, 1999.
- [3] G. Bryan. Fluid in the universe: Adaptive mesh refinement in cosmology. In *Computing in Science and Engineering*, 1(2):46–53, March/April, 1999.
- [4] M. Norman and G. Bryan. Cosmological adaptive mesh refinement. In *Computational Astrophysics*, 1998.
- [5] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Computing*, 7:279–301, October 1989.
- [6] K. Dragon and J. Gustafson. A low-cost hypercube load balance algorithm. In *Proc. Fourth Conf. Hypercubes, Concurrent Comput. and Appl.*, pages 583–590, 1989.
- [7] F. Lin and R. Keller. The gradient model load balancing methods. In *IEEE Transactions on Software Engineering*, 13(1):8–12, January 1987.
- [8] G. Horton. A multilevel diffusion method for dynamic load balancing. In *Parallel Computing*, (19):209–218, 1993.
- [9] L. Oliker and R. Biswas. PLUM:parallel load balancing for adaptive refined meshes. In *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.
- [10] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. In *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.
- [11] K. Schloegel, G. Karypis, and V. Kumar. A performance study of diffusive vs remapped load-balancing schemes. In *Proc. of Eleventh International Conference on Parallel and Distributed Computing*, 1998.
- [12] A. Sohn and H. Simon. Jove: A dynamic load balancing framework for adaptive computations on an sp-2 distributed multiprocessor. In *NJIT CIS Technical Report*, New Jersey, 1994.
- [13] C. Walshaw. Jostle:partitioning of unstructured meshes for massively parallel machines. In *Proc. Parallel CFD'94*, 1994.
- [14] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.
- [15] J. Rantakokko. A Framework for Partitioning Domains with Inhomogeneous Workload. In *Technical Report 194*, Department of Scientific Computing, Uppsala University, Uppsala, Sweden, 1997.
- [16] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. In *IEEE Concurrency*, pages 22–31, January-March 1999.
- [17] T. Bui et al. . Graph Bisection Algorithms with Good Average Behavior. In *IEEE FOCS*, pp. 181-191, 1984.
- [18] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In *The Bell System Technical Journal*, 49(2):291-307, 1970.
- [19] T. Chan and T. Mathew. Domain Decomposition Algorithms. pp. 61-143, *Acta Numerica*, 1994.
- [20] J. Castanos and J. Savage. Parallel Refinement of Unstructured Meshes. In *Proc. of IASTED International Conference Parallel and Distributed Computing and Systems*, 1999.
- [21] F. Stangenberg. Recognizing and Avoiding Thrashing in Dynamic Load Balancing. Technical Report, EPCC-SS94-04, September 1994.
- [22] K. Lie, V. Haugse, and K. Karlsen. Dimensional Splitting with Front Tracking and Adaptive Grid Refinement. Accepted in *Numerical Methods Partial Differential Equations*.