ACADEMIC PRESS

J. Parallel Distrib. Comput. 62 (2002) 1763–1781

Journal of Parallel and Distributed Computing

www.academicpress.com

A novel dynamic load balancing scheme for parallel systems

Zhiling Lan,[a,*,1] Valerie E. Taylor,[b,2] and Greg Bryan[c,3]

[a] *Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA*
[b] *Electrical and Computer Engineering Department, Northwestern University, Evanston, IL 60208, USA*
[c] *Nuclear and Astrophysics Laboratory, Oxford University, Oxford OX13RH, UK*

Received 23 October 2001; received in revised form 24 June 2002; accepted 10 September 2002

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. This scheme interleaves 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 SAMR cosmology application, the ENZO code. Experiments show that by using our scheme, the parallel execution time can be reduced by up to 57% and the quality of load balancing can be improved by a factor of six, as compared to the original DLB scheme used in ENZO.
© 2002 Elsevier Science (USA). All rights reserved.

Keywords: Dynamic load balancing; Adaptive mesh refinement; Parallel systems

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

*Corresponding author. Tel.: +312-567-5710; fax: +312-567-5067.
E-mail addresses: lan@iit.edu (Z. Lan), taylor@ece.nwu.edu (V.E. Taylor), gbryan@astro.ox.ac.uk (G. Bryan).
[1] Supported by a grant from the National Computational Science Alliance (ACI-9619019).
[2] Supported in part by NSF NGS grant EIA-9974960 and two NSF ITR grants (ITR-0086044 and ITR-0085952).
[3] Supported in part by a NASA Hubble Fellowship grant (HF-01104.01-98A).

dynamics, etc. The adaptive structure of AMR applications, however, results in load imbalance among processors on parallel systems. Dynamic load balancing (DLB) is an essential technique to solve this problem. In this paper, we present a novel DLB scheme that integrates a grid-splitting technique with direct grid movements. We illustrate the advantages of this scheme with a real cosmological application that uses structured AMR (SAMR) algorithm developed by Berger and Colella [1] in the 1980s.

Dynamic load balancing has been intensively studied for more than 10 years and a large number of schemes have been presented to date [2,5–8,10,12,14,16–18]. Each of these schemes can be classified as either *Scratch-and-Remap* or *Diffusion-based* schemes [13]. In *Scratch-and-Remap* schemes, the workload is repartitioned from scratch and then remapped to the original partition. *Diffusion-based schemes* employ the neighboring information to redistribute the load between adjacent processors such that global balance is achieved by successive migration of workload from overloaded processors to underloaded processors.

With any DLB scheme, the major issues to be addressed are the identification of overloaded versus underloaded processors, the amount of data to be redistributed from the overloaded processors to the underloaded processors, and the overhead that the DLB scheme imposes on the application. In investigating DLB schemes, we first analyze the requirements imposed by the applications. In particular, we complete a detailed analysis of the ENZO application, a parallel implementation of SAMR in astrophysics and cosmology [3], and identify the unique characteristics that impose challenges on DLB schemes. The results of the detailed analysis of ENZO provide four unique adaptive characteristics relating to DLB requirements: (1) coarse granularity, (2) high magnitude of imbalance, (3) different patterns of imbalance, and (4) high frequency of adaptations. In addition, ENZO employs an implementation that maintains some global information.

The fourth characteristic, the high frequency of adaptations, and the use of complex data structures result in Scratch–Remap schemes being intolerable because of the demand to completely modify the data structures without considering the previous load distribution. In contrast, Diffusion-based schemes are local schemes that do not utilize the global information provided by ENZO. In [13], it was determined that Scratch-and-Remap schemes are advantageous for problems in which high magnitude of imbalance occurs in localized regions, while Diffusion-based schemes generally provide better results for the problems in which imbalance occurs globally throughout the computational domain. The third characteristic, different patterns of imbalance, implies that an appropriate DLB scheme should provide good balancing for both situations. Further, the first characteristic, coarse granularity, is a challenge for a DLB scheme because it limits the quality of load balancing. Lastly, ENZO employs a global method to manage the dynamic grid hierarchy, that is, each processor stores a small amount of grid information about other processors. This information can be used by a DLB to aid in redistribution.

Utilizing the information obtained from the detailed analysis of ENZO, we develop a DLB scheme that integrates a grid-splitting option with direct data movement. In this scheme, each load-balancing step consists of one or more iterations 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. The use of direct communication to move the grids eliminates the variability in time to reach the equal balance and avoids chances of *thrashing* [15]. The splitting-grid phase splits a grid into two smaller grids along the longest dimension, thereby addressing the first characteristic, coarse granularity. These two phases are interleaved and executed in parallel. For each load-balancing step, the moving-grid phase is invoked first; then splitting-grid phase may be invoked if there are no more direct movements. If significant imbalance still exists, another round of two phases may be invoked. Further, in order to minimize communicational cost of each load balancing step, *nonblocking communication* is employed in this scheme and several computation functions are overlapped with these nonblocking calls.

The efficiency of our DLB scheme on SAMR applications is measured by both the execution time and the quality of load balancing. In this paper, three metrics are proposed to measure the quality of load balancing. Our experiments show that integrating our DLB scheme into ENZO results in significant performance improvement for all metrics. For example, the execution time of the *AMR64* dataset, a $32 \times 32 \times 32$ initial grid, on 32 processors is reduced by 57%, and the quality of load balancing is improved by a factor of six.

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 presents three load-balancing metrics followed by the experimental results exploring the impact of our DLB scheme on real SAMR applications. Section 6 gives a detailed sensitivity analysis of the parameter used in this proposed DLB scheme. Section 7 describes related work and compares our scheme with some widely used schemes. Finally, Section 8 summarizes the paper.

2. Overview of SAMR

This section gives an overview of the SAMR method, developed by Berger et al., and the ENZO code, a parallel implementation of this method for astrophysical and cosmological applications. Additional details about ENZO and the SAMR method can be found in [1,3,4,9].

2.1. Layout of grid hierarchy

SAMR represents the grid hierarchy as a tree of grids at any instant in time. The number of levels, the number of grids, and the locations of the grids change with each adaptation. Initially, a uniform mesh covers the entire computational domain. During the computation, finer grids are added in regions that require higher resolution. This process repeats recursively with each adaptation resulting in a tree of grids like that shown in Fig. 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 require further refinement. In this grid hierarchy, there are four levels of grids from level 0 to level 3. Throughout execution of a 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, aligned with its parent grid, and completely 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

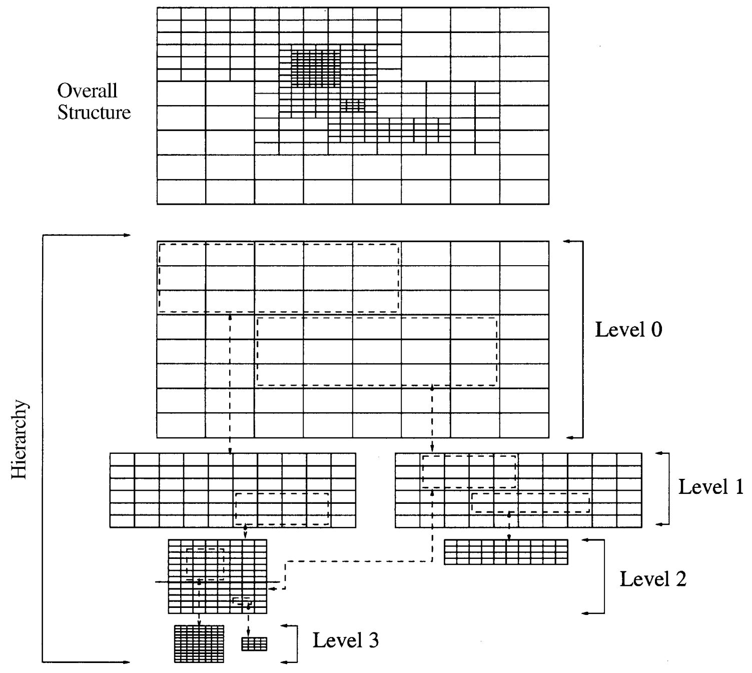

Fig. 1. SAMR grid hierarchy.

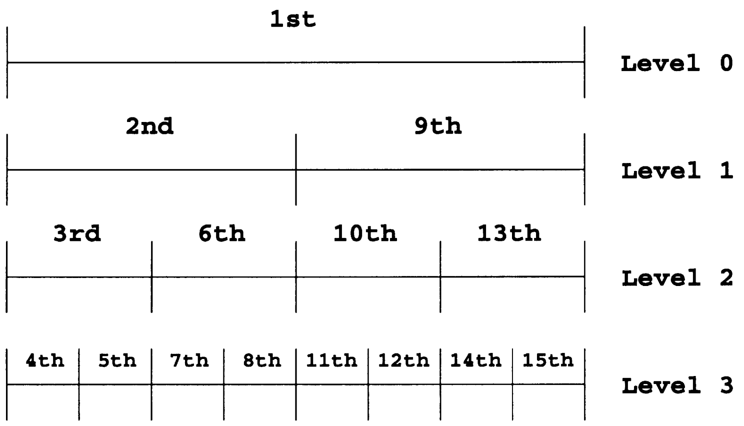

Fig. 2. Integrated execution order (refinement factor = 2).

next finer level at a smaller time step until it reaches the same physical time as that of the current level. Fig. 2 illustrates the execution sequence for an application with four levels and a refinement factor of 2. First, we start with the grids on level 0 with time step *dt*. Then the execution continues with the subgrids on level 1, with time step $dt/2$. Next, the integration continues with the subgrids on level 2, with time step $dt/4$, followed by the iteration of the subgrids on level 3 with time step $dt/8$. When the physical time at the finest level reaches that at level 0, the grids at level 0 proceed to the next iteration. The figure illustrates the order in which the subgrids are evolved 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 the 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 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. The code is written in C++ with Fortran routines for computationally intensive sections and MPI functions for message passing among processors. ENZO was developed as a community code and is currently in use at over six sites.

The ENZO implementation manages the grid hierarchy globally; that is, each processor stores the grid information of all other processors. In order to save space and reduce communication time, the notation of "real" grid and "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 replicates of this "real" subgrid, called "fake" grids. 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 and the global information, but does not address the large grid sizes (characteristic one). For this original DLB scheme, if the load-balance ratio (defined as *MaxLoad/MinLoad*) is larger than a hard-coded threshold, the load balancing process will be invoked. Here, *threshold* is used to determine whether a load-balancing process should be invoked after each refinement, and the default is set to 1.50. *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*. Fig. 3 gives the pseudocode of this scheme.

An example of grid movements that occurs with this DLB method is shown in Fig. 4. In this example, there are four processors: processor 0 is overloaded with two

The Original DLB Algorithm

```
while (MaxLoad > threshhold * MinLoad ) {
        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;
                }
        }
}
```

Fig. 3. Pseudo-code of the original DLB scheme.

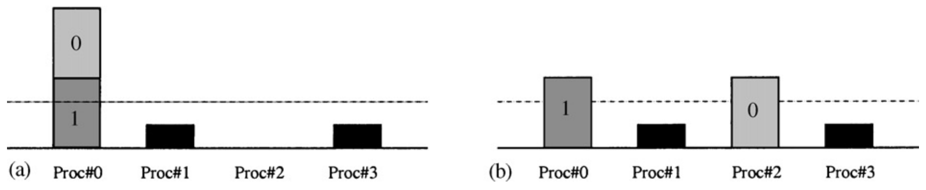

Fig. 4. An example of load movements using the original DLB scheme.

large-sized grids (0 and 1), processor 2 is idle, and processors 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 Fig. 4. 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. This issue is addressed with our DLB scheme, described in Section 4.

3. Adaptive characteristics of SAMR applications

This section provides some experimental results to illustrate the adaptive characteristics of SAMR applications running with ENZO implementation on the 250MHz R10000 SGI Origin2000 machine at the National Center for Supercomputing Applications (NCSA). Three real datasets (*AMR64*, *AMR128*, and *ShockPool3D*) are used in this experiment. Both *AMR128* and *AMR64* are designed to simulate the formation of a cluster of galaxies; *AMR128* is basically a larger version of *AMR64*. Both datasets use a hyperbolic (fluid) equation and an elliptic (Poisson's) equation as well as a set of ordinary differential equations for the particle trajectories. They 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 an increasing number of grids along the moving shock wave plane. It solves a purely hyperbolic equation. The sizes of these datasets are given in Table 1.

The adaptive characteristics of SAMR applications are analyzed from four aspects: granularity, magnitude of imbalance, patterns of imbalance, and frequency of refinements. 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.

Coarse granularity: Here, the granularity denotes the size of basic entity for data movement. For SAMR applications, the basic entity for data movement is a grid. Each grid consists of a *computational interior* and a *ghost zone* as shown in Fig. 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 3D 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, resulting in each grid having at least $(3 + 2 + 3)^3 = 512$ cells. The grids are often

Table 1
Three experimental datasets

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

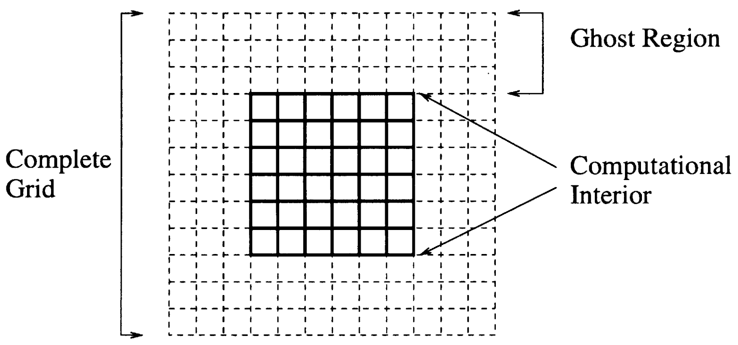

Fig. 5. Components of grids.

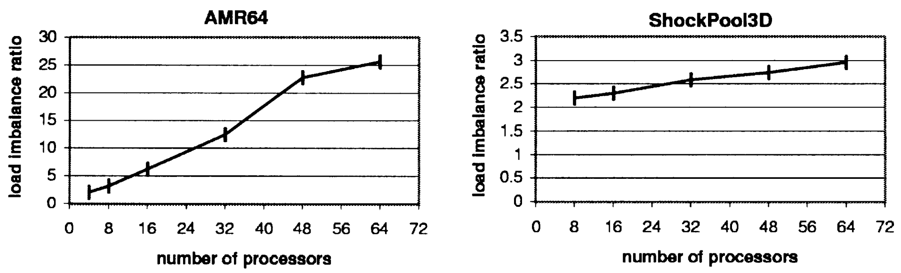

Fig. 6. Load imbalance ratio $\left(\text{defined as } \frac{max(load)}{average(load)}\right)$.

much larger than this minimum size. Usually, the amount of data associated with grids varies ranging from 100 KB to 10 MB. Thus the granularity of a typical SAMR application is very coarse, thereby making it very hard to achieve a good load balance by solely moving these basic entities. Coarse granularity is a challenge for a DLB scheme because it limits the quality of load balancing, thus a desired DLB scheme should address this issue.

High magnitude of imbalance: Fig. 6 shows the load imbalance ratio, defined as $max(load)/average(load)$, for *AMR64* and *ShockPool3D*, respectively. Here, load is defined as the total amount of grids in bytes on a processor and the ratio is an average over all the adaptations. The ideal case corresponds to the ratio equal to 1.0. The figure indicates that the load imbalance deteriorates as the number of processors increases. For *AMR64*, when the number of processors increases from 4 to 64, the load imbalance ratio increases from 2.02 to 25.64. For *AMR128*, the results are similar to those of *AMR64*. For *ShockPool3D*, this ratio increases from 2.19 to 2.96 as the number of processors increases from 8 to 64. For both cases, the ratio is

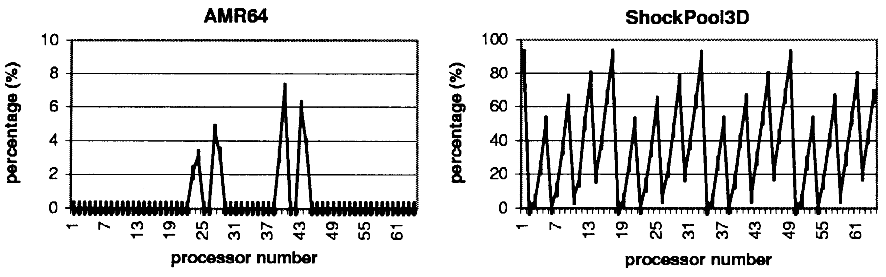

Fig. 7. Percentage of refinement per processor.

always larger than 2.0, which means a very high magnitude of imbalance exists for both datasets. Therefore, a DLB scheme is an essential technique for efficient execution of SAMR applications on parallel systems.

Different patterns of imbalance: Fig. 7 illustrates the different patterns of load imbalance between *AMR64* and *ShockPool3D*. The x-axis represents the processor number, and the y-axis represents the average percentage of refinement per processor. Processors that have large percentage of refinement will become overloaded. Thus, the figure also represents the imbalance pattern, whereby processors that have large percentage of refinement correspond to the regions that are overloaded. For *AMR64*, there are only a few processors whose loads 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) with significant percentages of refinements. As mentioned above, the dataset *AMR128* is essentially a larger 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%. Therefore, the imbalance occurs throughout the whole computational domain for *ShockPool3D*. The figure indicates that for SAMR applications, different datasets exhibit different imbalance distributions. The underlying DLB scheme should provide high quality of load balancing for all cases.

High frequency of refinements: 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 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 s, which means the adaptation process is invoked every 3.4 s on average. For the larger dataset *AMR128*, there are more than 5000 adaptations. High frequency of adaptation requires the underlying DLB method to execute very fast.

4. Our DLB scheme

After taking into consideration the adaptive characteristics of the SAMR application, we developed a novel DLB scheme which interleaves a grid-splitting option with direct data movement. In this scheme, each load-balancing step consists of one or more iterations of two phases: moving-grid phase and splitting-grid phase. The moving-grid phase redistributes grids directly from overloaded processors to underloaded processors by the guidance of the global information; and the splitting-grid phase splits a grid into two smaller grids along the longest dimension. For each load-balancing step, the moving-grid phase is invoked first; then splitting-grid phase may be invoked if no more direct movement can occur. If significant imbalance still exists, another round of two phases may be invoked. Fig. 8 gives the pseudocode of our scheme, and the details are given below.

Moving-grid phase: After each adaptation, our DLB scheme is triggered by checking whether $MaxLoad/AvgLoad > threshold$. The *MaxProc* moves its grid directly to *MinProc* under the condition that the redistribution of this grid will make the workload of *MinProc* reach *AvgLoad*. Here, *AvgLoad* denotes the required load for which all the processors would have an equal load. Thus, if there is a suitable sized grid, one direct grid movement is enough to balance an underloaded processor by utilizing the global information. This phase continues until either the load-balancing ratio is satisfied or no grid residing on the *MaxProc* is suitable to be moved.

Note that this phase differs from the original DLB scheme (Fig. 3) from several aspects. First, from Fig. 4, it is observed that the original DLB scheme may cause the previous underloaded processor (processor 2) to be overloaded by solely moving a grid from an overloaded processor to an underloaded processor. Our DLB algorithm overcomes this problem by making sure that any grid movement will make an underloaded processor reach, but not exceed, the average load.

DLB Algorithm

```
Done = 0; LastMax=LastMin=0;
while (MaxLoad > threshhold * AvgLoad  && Done == 0 ) {
    for (j=0;j<NumberOfGrids;j++)  {                              //moving-grid phase
       for (i=0;i<NumberOfGrids;i++) {
          if (grid(i) resides on MaxProc && grid(i) > AvgLoad/threshold-MinLoad
                                   && grid(i) < AvgLoad*threshold -MinLoad) {
             move grid(i) from MaxProc to MinProc;
             update load information of MaxProc and MinProc;
             find new MaxProc and MinProc;
             break;
          }
       }
       if (i == NumberOfGrids)
          break;
    }

    if (MaxLoad > threshold * AvgLoad ) {                         //splitting-grid phase
       if (MaxProc == LastMax && MinProc == LastMin)
          Done = 1;
       LastMax = MaxProc; LastMin = MinProc;
       find the largest grid MaxGrid on MaxProc;
       split MaxGrid into two and one of them redistributed to MinProc;
       update MaxProc, MinProc, MaxLoad, MinLoad, and AvgLoad;
    } else {
       Done = 1;
    }
}
```

Fig. 8. Pseudo-code of our DLB scheme.

Second, in this phase, a different metric ($MaxLoad/AvgLoad > threshold$) is used to identify when to invoke load-balancing process in contrast to the metric ($MaxLoad/MinLoad > threshold$) for the original scheme. The new metric results in fewer invocation of load-balancing steps, thereby low overhead. For example, consider two cases with the same average load of 10.0 for a six-processor system: the load distribution is $(20, 8, 8, 8, 8, 8)$ and $(12, 12, 12, 12, 12, 0)$, respectively. The distribution of the second case is preferred over the first case because the maximum run-time is less. If the *threshold* is set to 1.50, this moving-grid phase will be invoked for the first case because $\frac{MaxLoad}{AvgLoad} = 2.0$ is larger than the *threshold*, while it will not be invoked for the second case ($\frac{MaxLoad}{AvgLoad} = 1.20 < 1.50$). However, the original DLB scheme will invoke grid movements for both cases because the metric ($\frac{MaxLoad}{MinLoad}$) is larger than the *threshold* for both cases, e.g. $\frac{MaxLoad}{MinLoad} = 2.5$ for the first case and $\frac{MaxLoad}{MinLoad} = \infty$ for the second case. Hence, the new metric $\frac{MaxLoad}{AvgLoad} > threshold$ is more accurate and results in fewer load-balancing steps.

Splitting-grid phase: If no more direct grid movements can be employed and imbalance still exists, the splitting-grid 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 directly 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 about ($AvgLoad - MinLoad$), will be redistributed to *MinProc*. After such a splitting step, *MinProc* reaches the average load. Note that splitting does not mean splitting into equal pieces. Instead, the splitting is done exactly to fill the "hole" on the *MinProc*.

After such a splitting phase, if the imbalance still exists, another attempt of interleaving moving-grid phase and splitting-grid phase will continue. Eventually, either the load is balanced, which is our goal, or there are not enough grids to be redistributed among all the processors.

To illustrate the use of our DLB scheme, versus the original DLB scheme, we use the same example given in Fig. 4. In this example, the grid movements of our DLB scheme is shown in Fig. 9. The two grids on overloaded processor 0 are larger than ($threshold \times AvgLoad - MinLoad$), so no work can be 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. This is the first attempt of load balancing. The second attempt of load balancing begins with the direct grid movement of grid 0 from processor 0 to processor 1, followed by the grid splitting of grid 1 from processor 0 to processor 3. After two attempts of load balancing, the workload is equally redistributed to all the processors. As we can observe, compared with the

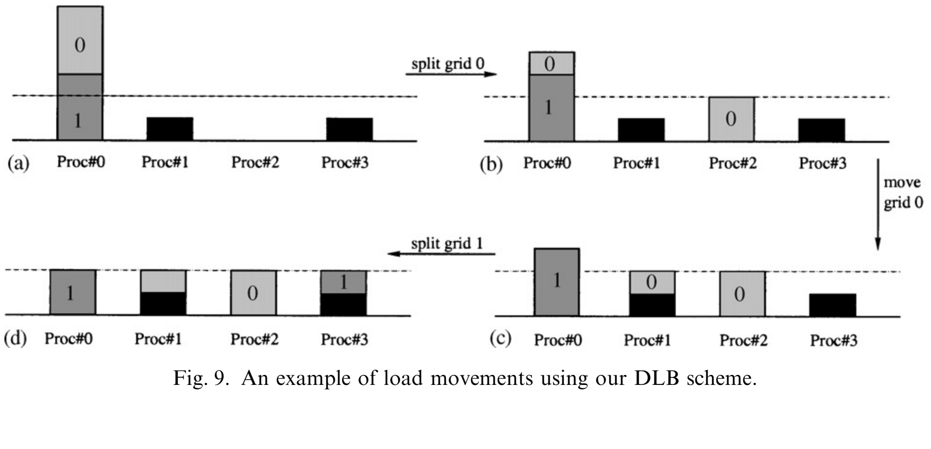
0
1
(a) Proc#0 Proc#1 Proc#2 Proc#3
split grid 0
(b) Proc#0 Proc#1 Proc#2 Proc#3
move grid 0
split grid 1
(c) Proc#0 Proc#1 Proc#2 Proc#3
(d) Proc#0 Proc#1 Proc#2 Proc#3

Fig. 9. An example of load movements using our DLB scheme.

grid movements of the original DLB scheme shown in Fig. 4, our DLB scheme invokes the grid-splitting phase for the case when the direct movement of grids is not enough to handle load imbalance. Our DLB scheme interleaves the grid-splitting technique with direct grid movements, thereby improving the load balance.

The use of global load information to move and split the grids eliminates the variability in time to reach the equal balance and avoids chances of *thrashing* [15]. In other words, the situation that multiple overloaded processors send their workload to an underloaded processor and make it overloaded will not occur by using the proposed DLB. Note that both the moving-grid phase and splitting-grid phase execute in parallel. For example, suppose there are eight processors as shown in Fig. 10. If the *MaxProc* and *MinProc* are processors 0 and 5, respectively, all the processors know which grid will be moved/split from processor 0 to processor 5 by the guidance of the global information. Then processor 0 moves/splits its grid (i.e. the "real" grid) to processor 5. In parallel, other processors first update their view of load distribution of this grid movement from processor 0 to processor 5, then continue load-balancing process. If the new *MaxProc* and *MinProc* are processor 1 and 2 respectively, then processor 1 will move/split its grid (i.e. "real" grid) to processor 2, and this process will be overlapped with the movement from processor 0 to processor 5. The remaining processors (3, 4, 6 and 7) continue with first updating their view of load distribution of the grid movement from processor 1 to processor 2 followed by calculating the next *MaxProc* and *MinProc*. Because the new *MaxProc* and *MinProc* are processor 0 and 3, respectively, so processor 3 has to wait for processor 0 which is still in the process of transferring its workload to processor 5.

In order to minimize the overhead of the scheme, *nonblocking communication* is explored in this scheme. In the mode of nonblocking communication, a nonblocking post-send initiates a send operation and returns before the message is copied out of the send buffer. A separate complete-send call is needed to complete the communication. The nonblocking receive is proceeded similarly. In this manner, the transfer of data may proceed concurrently with computations done at both the sender and the receiver sides. In the splitting-grid phase, nonblocking calls are explored and being overlapped with several computation functions.

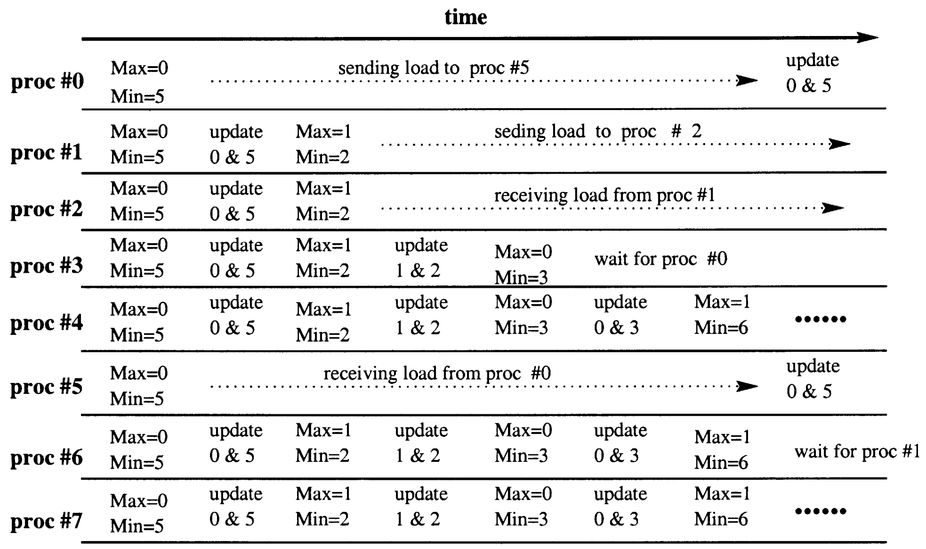

Fig. 10. An illustration of the parallelism in our DLB scheme.

5. Experimental results

The potential benefits of our DLB scheme were examined by executing real SAMR applications running ENZO on parallel systems. All the experiments were executed on the 250 MHz R10000 SGI Origin2000 machines at NCSA. The code was instrumented with performance counters and timers, which do not require any I/O at run-time. The *threshold* is set to 1.20.

5.1. Comparison metrics

The effectiveness of our DLB scheme is measured by both the execution time and the quality of load balancing. In this paper, three metrics are proposed to measure the quality of load balancing. Note that each metric below is arithmetic average over all the adaptations.

Imbalance ratio is defined as

$$imbalance\_ratio = \frac{\sum_{j=1}^{N} \frac{MaxLoad(j)}{AvgLoad(j)}}{N}, \tag{1}$$

where N is number of adaptations, $MaxLoad(j)$ denotes the maximal amount of load of a processor for the jth adaptation, and $AvgLoad(j)$ denotes the average load of all the processors for the jth adaptation. It is clear that *imbalance_ratio* is greater or equal to 1.0. The closer it is to 1.0 the better; the value of 1.0 implies equal load distribution among all processors.

Standard deviation of imbalance ratio is defined as

$$avg\_std = \frac{\sum_{j=1}^{N} \sqrt{\frac{\sum_{i=1}^{P} (\frac{MaxLoad(j)}{L_i(j)} - \frac{MaxLoad(j)}{AvgLoad(j)})^2}{P-1}}}{N}, \tag{2}$$

where P is number of processors and $L_i(j)$ denotes the workload of ith processor for the jth adaptation. By definition, the capacity to keep *avg_std* low during the execution is one of the main quality metrics for an efficient DLB. The capacity to have *avg_std* be a small fraction of *imbalance_ratio* indicates that *imbalance_ratio* can truly represent the imbalance over all the adaptations.

Percentage of idle processors is defined as

$$idle\_procs = \frac{\sum_{j=1}^{N} idle(j)}{N}, \tag{3}$$

where $idle(j)$ is the percentage of idle processors for the jth adaptation. Here, an idle processor is the processor whose amount of load is zero. As mentioned above, due to the coarse granularity and the minimal size requirement of SAMR applications, it is possible that there may be some idle processors for each iteration. Obviously, the smaller this metric, the better the load is balanced among the processors.

5.2. Execution time

Fig. 11 compares the total execution times with varying numbers of processors by comparing our DLB scheme with the original DLB scheme for the datasets *AMR64*, *AMR128*, and *ShockPool3D*. Table 2 summarizes the relative improvements by using our DLB scheme. It is observed that our DLB scheme 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 12.60% and 57.54% for *AMR64*, between −4.84% and 20.17% for *AMR128*, and between 8.07% and

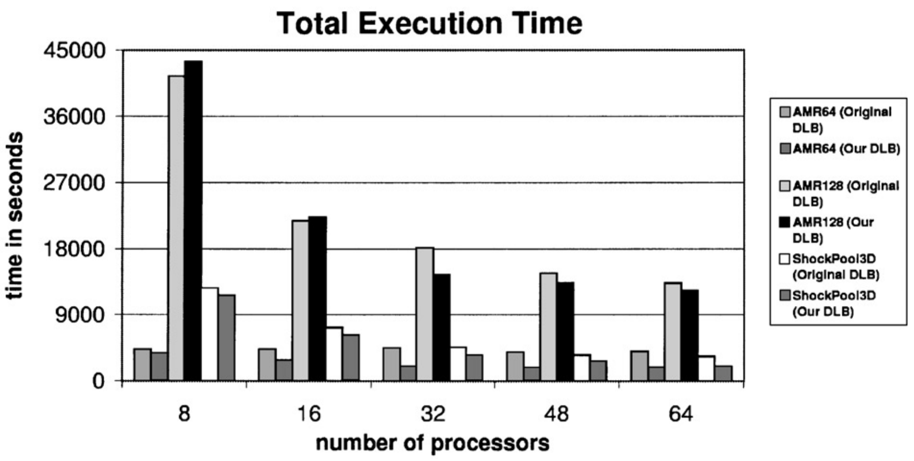

Fig. 11. Total execution time for *AMR64*, *AMR128*, and *ShockPool3D*.

Table 2
Relative improvement for three datasets

Relative improvement	8 Procs. (%)	16 Procs. (%)	32 Procs. (%)	48 Procs. (%)	64 Procs. (%)
AMR64	12.60	34.15	57.54	54.19	53.69
AMR128	−4.84	−2.26	20.17	9.14	7.53
ShockPool3D	8.07	14.18	23.73	24.15	42.13

42.13% for *ShockPool3D*. We may notice that there are two exceptions for which our DLB results in larger execution times. When executing *AMR128* on 8 or 16 processors, our DLB scheme has worse performance compared to the original DLB scheme achieves. The reason is that our DLB scheme tries to improve the load balance by using the grid-splitting technique, which entails some communication and computation overheads. When more smaller grids are introduced across processors, more communication is needed to transfer data among processors and more computational load is added because each grid requires a "ghost zone" to store boundary information. Hence, for these cases when the number of processors is small and the original DLB scheme provides relatively good load balancing, the overheads introduced by our scheme may be larger than the gain. However, when the original DLB scheme is inefficient, especially when the number of processors is larger than 16, our DLB scheme 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.

5.3. Quality of load balancing

The first load-balancing metric *imbalance_ratio* is given in Fig. 12. The solid lines represent the results by using our proposed DLB scheme (denoted as parallel DLB), and the dash lines represent the results by using the original DLB scheme. The *imbalance_ratio* increases as the number of processors increases by using either of two methods. This is reasonable because it is more likely that there are not enough

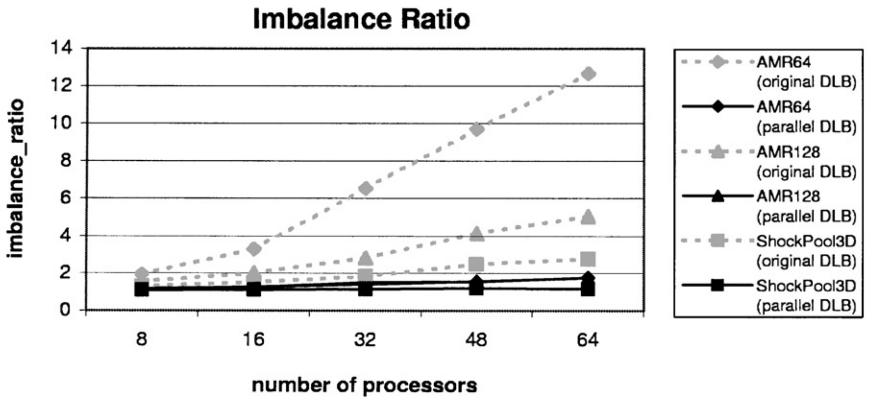

Fig. 12. Imbalance ratio for three datasets.

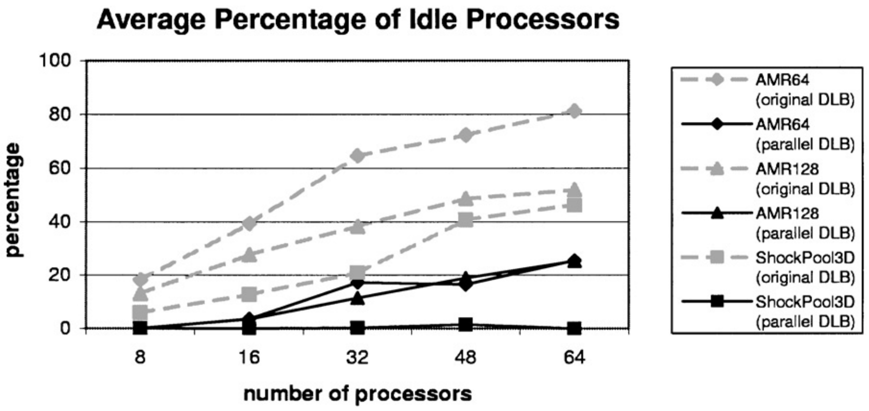

Fig. 13. Average percentage of idle processors.

grids to be redistributed among processors if there are more processors. Our proposed DLB, however, is able to significantly reduce the imbalance ratio by interleaving direct grid movement with grid splitting for all cases. Further, the amount of improvement gets larger as the number of processors increases. In general, the *imbalance_ratio* by using our DLB scheme is always less than 1.80, which is significant. As compared to the original DLB scheme, the relative improvement of *imbalance_ratio* is in the range of 33%–615% by using our DLB scheme. The result of the second load-balancing metric, *avg_std* (not shown), indicates that the standard deviation of *imbalance_ratio* is quite low (less than 0.035), which means that *imbalance_ratio* can truly represent the load imbalance among all processors.

The third metric *percentage of idle processors* is shown in Fig. 13. The solid lines and the dash lines represent results by using our DLB (denoted as parallel DLB) and the original DLB scheme respectively. The results indicate that the metric increases as the number of processors increases for both schemes. This is due to the fact that it is more likely there are not enough grids for movement when there are more processors. However, the percentage of idle processors is much lower by using our DLB scheme; the metric ranges from zero to approximately 25% for our DLB scheme, as compared to 5.9%–81.2% for the original DLB scheme. Larger percentage of idle processors means more computing resources are wasted.

6. Sensitivity analysis

A parameter called *threshold* is used in our DLB scheme (see Fig. 8), which determines whether a load-balancing process should be invoked after each refinement. Intuitively, the ideal value should be 1.0, which means all the processors are evenly and equally balanced. However, the closer this threshold is to 1.0, the more load-balancing actions are entailed, so the more overhead may be introduced. Furthermore, for SAMR applications, the basic entity is a "grid" which has a minimal size requirement. Thus the ideal situation in which the load is perfectly balanced may not be obtained. The *threshold* is used to adjust the quality of load balancing, whose value influences the efficiency of the overall DLB scheme. What is the optimal value for this threshold is the topic of this section. We give the experimental results to compare the performance and the quality of load balancing by varying *threshold* from a small value of 1.10 to a large value of 2.00 and identify the optimal value for the parameter.

Figs. 14 and 15 illustrate the relative performances by normalizing the execution times to the minimal time for each *threshold* value. The figures indicate that the smaller value for this *threshold* may result in worse performance because more overheads are introduced. For example, for *AMR64* running on 32, 48, and 64 processors, the relative execution times by setting this parameter to 1.10 are usually two times above the minimal execution times. Secondly, a larger value for this

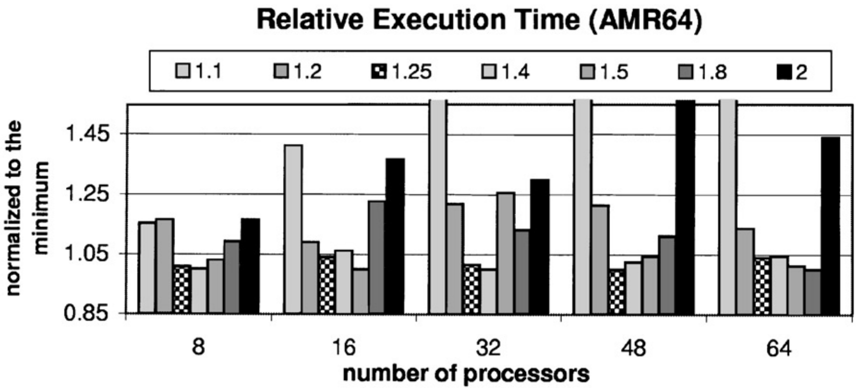

Fig. 14. Relative execution time for *AMR64* (various *threshold*).

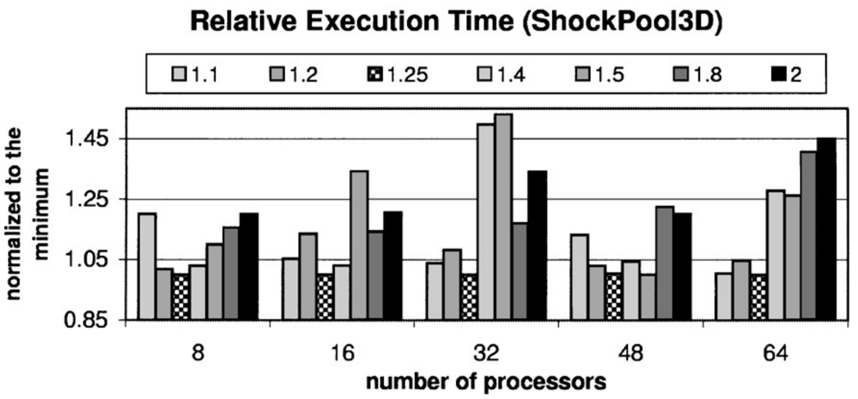

Fig. 15. Relative execution time for *ShockPool3D* (various *threshold*).

threshold may also result in a worse performance due to the poorer quality of load balancing. For example, for *ShockPool3D*, by setting this parameter to 2.00, the relative execution times are 20–45% above the minimal execution times. Both figures indicate that setting *threshold* to 1.25 results in the best performance in terms of execution time because the relative execution times are always no more than 5% above the minimal execution times.

Figs. 16 and 17 illustrate the quality of load balancing with varying values of *threshold* for *AMR64* and *ShockPool3D*. Here, the quality of load balancing is measured by the *imbalance_ratio*. From both figures, it is clear that the smaller is the *threshold*, the smaller is the *imbalance_ratio*, thereby resulting in a higher quality of load balancing. A smaller value of *threshold* indicates that more load-balancing actions may be entailed to balance the workload among the processors. Further, we can observe that *imbalance_ratio* gets larger as the number of processors increases. When the number of processors is increased, there may not be enough grids to be distributed among the processors, which results in lower quality of load balancing. It seems that the best case is to set *threshold* to a smaller value, such as 1.10. However, this small value means more load-balancing attempts are invoked, which would introduce more overhead and the overall performance may be deteriorated as shown in Figs. 14 and 15. For both datasets, our results of *idle_procs* (not shown) indicate

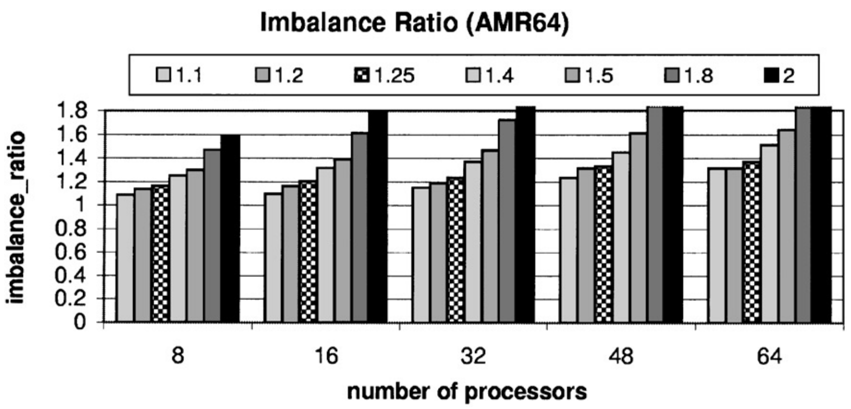

Fig. 16. Quality of load balancing for *AMR64* (various *threshold*).

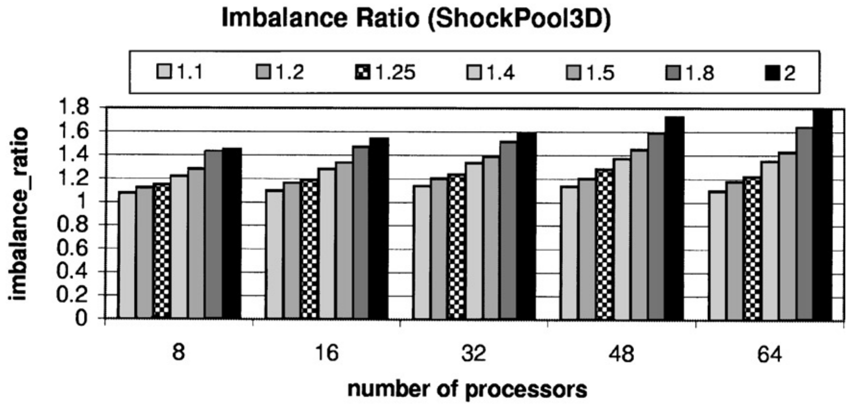

Fig. 17. Quality of load balancing for *ShockPool3D* (various *threshold*).

that there is no significant difference of its values with the varying values of *threshold*.

By combining the results in Figs. 14–17, we determine that setting *threshold* to be around 1.25 would result in the best performance in terms of execution time, as well as the acceptable quality of load balancing.

7. Related work

Our DLB scheme is not a Scratch–Remap scheme because it takes into consideration the previous load distribution during the current redistribution process. As compared to Diffusion-based 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 employs the direct data movement between overloaded and underloaded processors by the guidance of global load information.

Grid splitting is a well-known technique and has been applied in several research works. Rantakokko uses this technique in his static load-balancing scheme [11] which is based on *recursive spectral bisection*. The main purpose of using grid splitting is to 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.

8. Summary

In this paper, we presented a novel dynamic load-balancing scheme for SAMR applications. Each load-balancing step of this scheme consists of two phases: moving-grid phase and splitting-grid phase. The potential benefits of our scheme were examined by incorporating our DLB scheme 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, as compared to the original DLB scheme. By using our DLB scheme, the total execution time of SAMR applications was reduced up to 57%, and the quality of load balancing was improved significantly especially when the number of processors is larger than 16.

While the focus of this paper is on SAMR, some techniques can be easily extended to other applications, such as using grid splitting to address coarse granularity of basic entity, utilizing some global load information for data redistribution, interleaving direct data movements with splitting technique, and the imbalance detection by using $max(load)/average(load)$ to measure load imbalance. Further, we believe the techniques are not limited to SAMR applications and should have broader applicability, such as unstructured AMR. For example, it could be extended to any application with the following characteristics: (1) it has coarse granularity and (2) each processor has a global view of load distribution, that is, each processor should be aware of the load distribution of other processors.

Acknowledgments

The authors thank Michael Norman at UCSD for numerous comments and suggestions that contributed to this work. We also acknowledge the National Center for Supercomputing Application (NCSA) for the use of the SGI Origin machines.

References

[1] M. Berger, P. Colella, Local adaptive mesh refinement for shock hydrodynamics, J. Comput. Phys. 82 (1) (May 1989) 64–84.
[2] E. Boman, K. Devine, B. Hendrickson, W. Mitchell, M. John, C. Vaughan, Zoltan: a dynamic load-balancing library for parallel applications, World Wide Web, http://www.snl.gov/zoltan.
[3] G. Bryan, Fluid in the universe: adaptive mesh refinement in cosmology, Comput. Sci. Eng. 1 (2) (March/April 1999) 46–53.
[4] G. Bryan, T. Abel, M. Norman, Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: resolving primordial star formation, in: Proceedings of SC2001, Denver, CO, 2001.
[5] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, IEEE Trans. Parallel Distrib. Systems 7 (October 1989) 279–301.
[6] K. Dragon, J. Gustafson, A low-cost hypercube load balance algorithm, in: Proceedings of the Fourth Conference on Hypercubes, Concurrent Computations and Applications, 1989, pp. 583–590.
[7] G. Horton, A multilevel diffusion method for dynamic load balancing, Parallel Comput. (19) (1993) 209–218.
[8] F. Lin, R. Keller, The gradient model load balancing methods, IEEE Trans. Software Eng. 13 (1) (January 1987) 8–12.
[9] M. Norman, G. Bryan, Cosmological adaptive mesh refinement, Comput. Astrophys. (1998).
[10] L. Oliker, R. Biswas, Plum: parallel load balancing for adaptive refined meshes, J. Parallel Distrib. Comput. 47 (2) (1997) 109–124.
[11] J. Rantakokko, A framework for partitioning domains with inhomogeneous workload, Technical Report, Uppsala University, Sweden, 1997.
[12] K. Schloegel, G. Karypis, V. Kumar, Multilevel diffusion schemes for repartitioning of adaptive meshes, J. Parallel Distrib. Comput. 47 (2) (1997) 109–124.
[13] K. Schloegel, G. Karypis, V. Kumar, A performance study of diffusive vs. remapped load-balancing schemes, in: Proceedings of the 11th International Conference on Parallel and Distributed Computing, 1998.
[14] A. Sohn, H. Simon, Jove: a dynamic load balancing framework for adaptive computations on an sp-2 distributed multiprocessor, NJIT CIS Technical Report, New Jersey, 1994.
[15] F. Stangenberg, Recognizing and avoiding thrashing in dynamic load balancing, Technical Report, EPCC-SS94-04, September 1994.
[16] KeLP Team, The KeLP programming system, World Wide Web, http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html, 1999.
[17] C. Walshaw, Jostle: partitioning of unstructured meshes for massively parallel machines, in: Proceedings of Parallel CFD'94, 1994.
[18] M. Willebeek-LeMair, A. Reeves, Strategies for dynamic load balancing on highly parallel computers, IEEE Trans. Parallel Distrib. Systems 4 (9) (September 1993) 979–993.

Zhiling Lan received her BS in mathematics from Beijing Normal University and her MS in applied mathematics from Chinese Academy of Sciences in 1992 and 1995, respectively; she received her Ph.D. in computer engineering from Northwestern University in 2002. She is currently a faculty member in the Department of Computer Science at Illinois Institute of Technology starting from August 2002. Her research interests are in the area of parallel and distributed systems, and performance analysis and modeling. She is a member of IEEE and ACM.

Valerie E. Taylor is a professor in the Electrical and Computer Engineering Department at Northwestern University, where she leads the CELERO Performance Analysis Research Group. She received her B.S. in computer and electrical engineering and M.S. in electrical engineering from Purdue University in 1985 and 1986, respectively; she received her Ph.D. in electrical engineering and computer science from University of California at Berkeley in 1991. Valerie Taylor holds an US patent for her dissertation work on sparse matrices. She received a National Science Foundation National Young Investigator Award in 1993. Her research interests are in the area of high performance computing, with particular emphasis on performance analysis of parallel and distributed scientific applications. She has published over 70 papers in the area of high performance computing. Currently, she is a member of ACM and a senior member of IEEE.

Greg Bryan received a B.Sc. from the University of Calgary in 1991 and a Ph.D. in astrophysics from the University of Illinois in 1996. He is currently a faculty member in the Department of Physics at Oxford University and performs research both in formation of structure in the universe, and in the development of computational techniques for modeling the formation and evolution of multi-scale systems.