

Dynamic Load Balancing for Adaptive Mesh Refinement Applications: Improvements and Sensitivity Analysis *

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 dynamically achieves high resolution in localized regions of multidimensional numerical simulations. A dynamic load balancing (DLB) scheme for structured AMR applications was proposed in [19]. Unfortunately, the overhead introduced by this DLB scheme is significant. Further, a parameter called *threshold* is used in this scheme which determines whether a load balancing process should be invoked; its value directly influences the efficiency of the overall DLB scheme. In this paper, we first present two improvements on this DLB scheme to reduce its overhead. Then a detailed sensitivity analysis is provided to identify an optimal value for the parameter *threshold*. Experiments show that by interleaving grid splitting with direct grid movement and by employing nonblocking communication, the execution time can be improved by up to 32% and the overhead can be reduced as much as 18%; and setting *threshold* to be around 1.25 is optimal.

KEY WORDS

dynamic load balancing, adaptive mesh refinement, sensitivity analysis

1. Introduction

Adaptive Mesh Refinement (AMR) is a class of multiscale algorithms that dynamically achieves high resolution in localized regions of 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. ENZO, developed by G. Bryan and M. Norman [3], is one of the successful parallel implementations for structured AMR (SAMR) algorithm presented by M. Berger et al. [1] on distributed-memory systems. In [19], a DLB scheme was proposed for this application and the experiments showed that this scheme can significantly improve

the performance of SAMR applications. However, the overhead introduced by this DLB scheme is significant. Further, there is one parameter called *threshold* used in this scheme and its value is not specified in [19]. In this paper, we first present two techniques to reduce the overhead of this DLB scheme so as to improve the overall performance; then provide a detailed sensitivity analysis of *threshold* and identify an optimal value for this parameter.

The adaptive structure of SAMR applications results in load imbalance among processors on parallel and distributed systems, and dynamic load balancing is an essential technique to solve this problem. After considering those unique adaptive characteristics of SAMR applications [19], we proposed a DLB scheme for this type of applications. Basically, after each refinement step, the load imbalance among the processors (defined as $\max(\text{load})/\text{average}(\text{load})$) is compared to a predefined threshold. If the imbalance is larger than the threshold, the load balancing process is invoked. Each load balancing consists of two separate phases: *moving-grid phase* and *splitting-grid phase*. The *moving-grid phase* is executed first. For this phase, suitable sized grids are moved directly from overloaded processors to underloaded processors in parallel; then the *splitting-grid phase* is invoked next if direct grid movements cannot balance the grids among processors.

The experiments with the above DLB scheme show that the overhead ranges from zero to almost 40% of the execution time. It was observed that the grid splitting phase introduces more overhead as compared to the grid movement phase. Further, the most time-consuming component of the overhead is attributed to the one-to-all broadcast to share grid information during the *splitting-grid phase*. Reducing the number of grid splitting attempts and the communication cost can significantly reduce overhead. Two improvements explored in this paper consist of (1) interleaving grid splitting with direct grid movement and (2) replacing synchronous communication with nonblocking communication. The benefits of these improvements show a reduction in execution time by up to 32% and the overhead with respect to the execution time can be reduced by up to 18%. Further, we determine good values for the threshold value.

In this paper, two real datasets (*AMR64* and *Shock-*

*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).

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

Table 1. Two Experimental Datasets

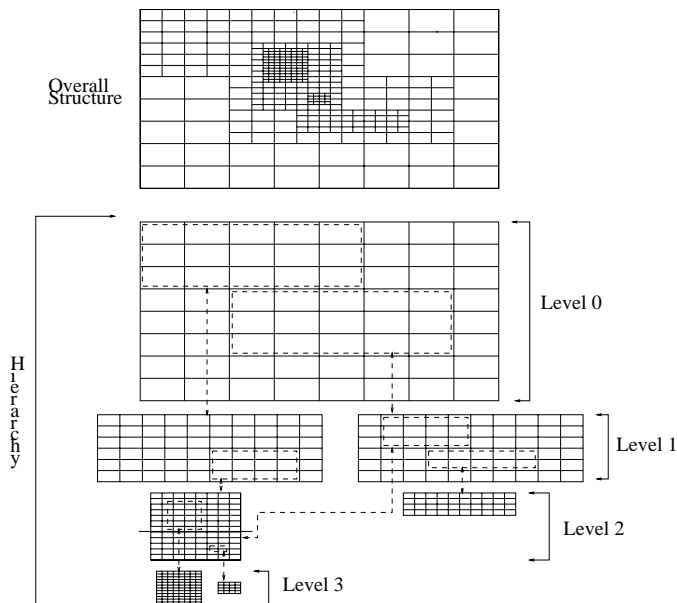


Figure 1. AMR Grid Hierarchy

Pool3D) are used. *AMR64* is designed to simulate the formation of a cluster of galaxies, which creates 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. It creates more and more grids along the moving shock wave plane. The sizes of these datasets are given in Table 1. All the experiments in this paper were executed on the 250MHz R10000 SGI Origin2000 machines at National Center for Supercomputing Applications.

The remainder of this paper is organized as follows. Section 2 introduces structured AMR algorithm. Section 3 gives an overview of the DLB scheme we proposed for this application. Section 4 illustrates the techniques to reduce overhead of the above scheme. Section 5 gives a detailed sensitivity analysis of the parameter used in the DLB scheme and identifies an optimal value. Finally, section 5 summarizes the paper.

2. Structured Adaptive Mesh Refinement Algorithm

This section gives an overview of SAMR developed by M. Berger et al. [1]. It 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. 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.

ENZO [2, 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. 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).

3. Dynamic Load Balancing Scheme

A dynamic load balancing scheme was proposed for SAMR applications in [19]. This scheme combines a *grid-splitting* technique with direct movement from overloaded processors to underloaded processors. After each refinement step, the load balancing process is triggered if $\max(\text{load})/\text{average}(\text{load}) > \text{threshold}$.

The *moving-grid phase* is executed first. The *Max-Proc* which has the maximal load moves its grid directly to *Min-Proc* which has the minimal load under the condi-

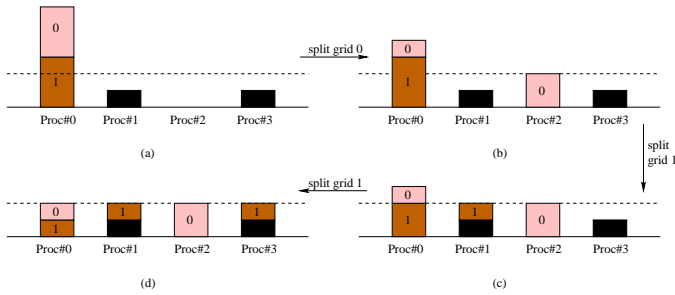


Figure 2. An Example of Load Movements using the DLB Scheme

tion that the computational load of this grid is no more than $(threshold \times AvgLoad - MinLoad)$. $AvgLoad$ denotes the required load for which all the processors would have an equal load. This phase continues until either the load-balancing ratio is satisfied or all grids residing on the *MaxProc* are too large to be moved.

The *splitting-grid phase* will be invoked if the direct grid movements cannot balance the load among processors. The *MaxProc* first finds the largest grid it owns (denoted as *MaxGrid*). If the size of *MaxGrid* is larger than $(AvgLoad - MinLoad)$ which is the amount of load needed by *MinProc*, the grid will be split along its longest dimension into two smaller grids. One of two split grids, whose size is around $(AvgLoad - MinLoad)$, will be redistributed to *MinProc*. 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.

Figure 2 provides an example to illustrate the concepts of the DLB scheme. 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. First, load imbalance is detected, so the *moving-grid phase* is executed first. Since those two grids on overloaded processor 0 are larger than $(threshold \times AvgLoad - MinLoad)$, no grid is moved 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. This phase continues until the load balancing is satisfying. Note that both direct grid movements and grid splitting are executed in parallel.

4. Improvements

The experiments in [19] show that the overhead introduced by the above DLB scheme is significant, ranging from zero to almost 40.0% of the execution time as the number of pro-

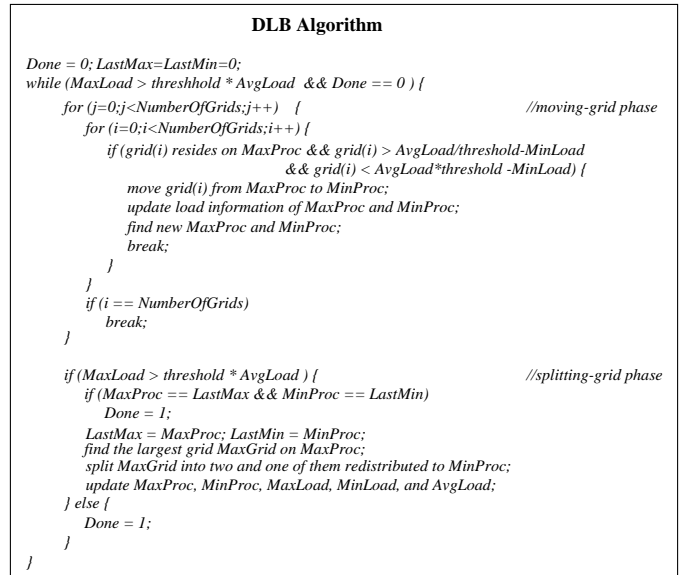


Figure 3. Pseudocode of the Improved DLB Scheme

cessors increases. The above scheme separates the *moving-grid phase* from the *splitting-grid phase*, that is, no more direct grid movement is allowed after the *splitting-grid phase* is executed during each load balancing process. It was observed that grid splitting introduces more overhead as compared to the direct grid movement. Further, the most time-consuming component of the overhead is attributed to the one-to-all communication to share grid information during the *splitting-grid phase*. Reducing the number of grid splitting attempts and the communication cost can significantly reduce the overhead.

In order to reduce the number of grid splitting attempts, we modified the above DLB scheme by interleaving the *moving-grid phase* with *splitting-grid phase*. During each load balancing process, the *moving-grid phase* is executed first. Only the grid, whose size is larger than $(AvgLoad/threshold - MinLoad)$ and smaller than $(AvgLoad \times threshold - MinLoad)$, is moved directly from *MaxProc* to *MinProc*. If none of grids is suitable to move, *splitting-grid phase* is invoked. After such a grid splitting, the load balancing process goes back to *moving-grid phase*. The principal difference is that the *splitting-grid phase* will be allowed only if no more direct grid movement can be done while load imbalance still exists. The pseudocode of this improved scheme is illustrated in the Figure 3.

To illustrate the improvement over the DLB described in section 3, we use the same example given in Figure 2. For this example, the grid movements of the improved DLB is shown in Figure 4. The improved scheme employs the direct grid movement of grid 0 from processor 0 to processor 1 (from (b) to (c)), as compared to the grid splitting of grid 1 from processor 0 to processor 1 shown in the Figure 2 (from (b) to (c)). Compared with the grid

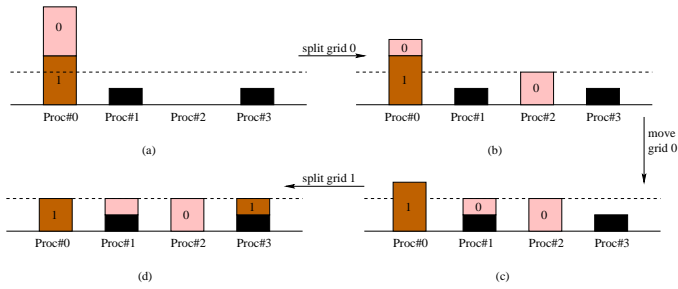


Figure 4. An Example of Load Movements using the Improved DLB Scheme

movements shown in Figure 2, the number of grid splitting is reduced, which results in two performance improvements. First, less overhead will be entailed in the load balancing, since direct grid movement introduces less overhead as compared to grid splitting. Secondly, less amount of workload will be introduced into the computation, since each split grid requires some "ghost region" to obtain the boundary information[19].

An efficient mechanism to reduce the communication cost is to replace blocking communication with nonblocking communication, which is the second improvement we made to the DLB scheme discussed in section 3. 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. We replaced the one-to-all broadcasting with nonblocking calls and overlapping some computation functions with these nonblocking communications. The experimental results turned out to be very promising.

Figure 5 and Table 2 illustrate the benefits offered by employing these two improvements. Figure 5 represents the total execution times with varying number of processors; Table 2 summarizes their relative improvements. The results indicate that these improvements can greatly reduce the execution time, especially when the number of processors is more than 16. For example, for both datasets, when the number of processors is larger than 16, the relative improvement is in the range of 19% - 32%.

The overhead with respect to the total execution time is summarized in the Figure 6. It is shown that the overhead introduced by the improved DLB scheme can be reduced considerably as well. For *AMR64*, the overhead is reduced by 5% - 18%; for *ShockPool3D*, the overhead is reduced by zero - 12%. The figure also indicates that the reduction of overhead gets larger as the number of processors increases.

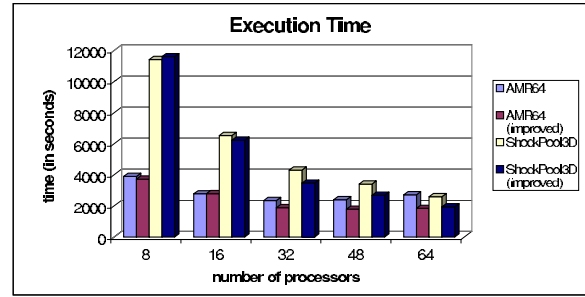


Figure 5. Comparison of Execution Time

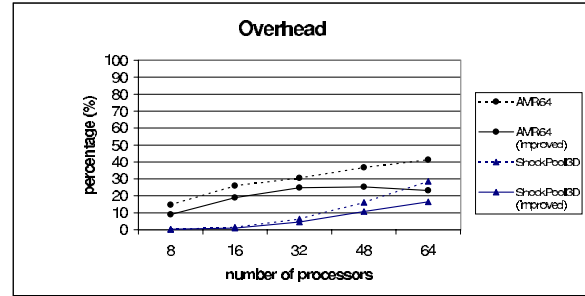


Figure 6. Comparison of Overhead

5. Sensitive Analysis

A parameter called *threshold* is used in the DLB scheme (see Figure 3), 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 processes are entailed, so the more overhead may be introduced. Furthermore, for SAMR applications, the basic entity is a "grid" which cannot be split infinitely. Thus the ideal case that all the processors are evenly and equally balanced may not be obtained. The *threshold* 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.

Figures 7- 10 compare the execution times with varying values of *threshold* for the dataset *AMR64* and *ShockPool3D* respectively. The figures on the top shows the total execution times; and the figures on the bottom 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 overhead 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

Relative Improvement	8 procs	16 procs	32 procs	48 procs	64 procs
AMR64	4.52%	-0.40%	19.87%	25.57%	32.08%
ShockPool3D	-1.68%	4.54%	19.46%	21.09%	25.24%

Table 2. Relative Improvement By Using Nonblocking Communication

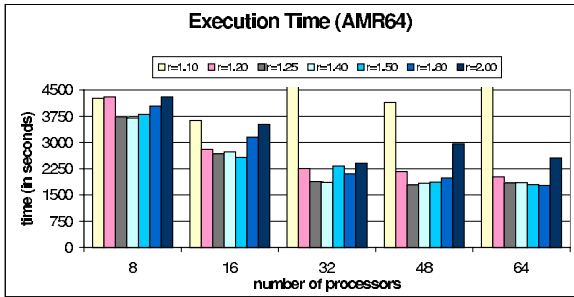


Figure 7. Execution Times for AMR64

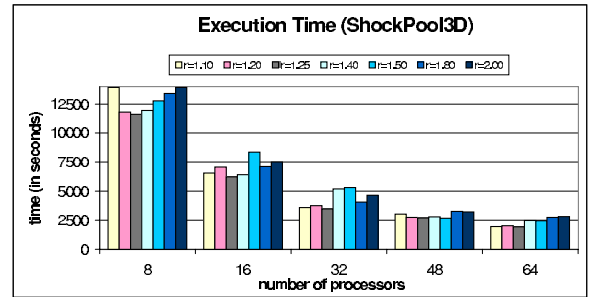


Figure 9. Execution Times for ShockPool3D

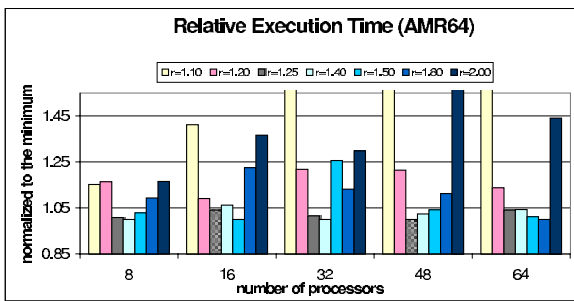


Figure 8. Relative Execution Times for AMR64

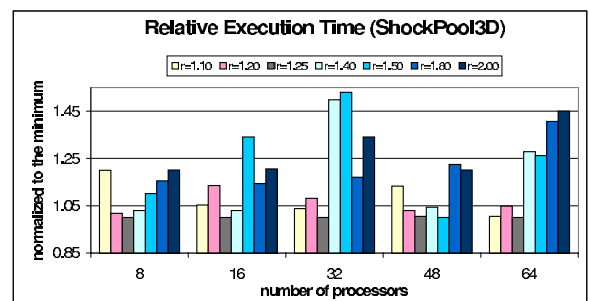


Figure 10. Relative Execution Times for ShockPool3D

the minimal execution times. Secondly, a larger value for this *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. Further, 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 for both datasets.

Figures 11 and 12 present the qualities of load-balancing with varying values of *threshold*. Here, the quality of load-balancing is measured by the *Average Load*, which is the relative workload normalized to the maximal load during the computation. The closer this metric to 1.0 the better; the value of 1.0 implies a perfect load balancing among all the processors. Generally, the smaller this parameter is, the higher quality the load-balancing results in. It is reasonable because smaller value means more load-balancing processes are entailed to balance the workload among processors. Further, we can observe that the quality of load-balancing is decreased as the number of processors increased. When the number of processors is increased, there may be not 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 Figure 7-Figure 10.

By combining the results in Figure 7-Figure 12, we determine that setting *threshold* to be around 1.25 would result in the best performance in terms of execution time, as well as acceptable quality of load-balancing.

6. Summary

In [19], a parallel DLB scheme was proposed for SAMR applications. The experiments showed that this scheme can significantly improve the performance of SAMR applications; however, the overhead of this DLB is significant. In this paper, two improvements were made on this DLB scheme to reduce the overhead: interleaving grid splitting with direct grid movement and replacing the one-to-all broadcasting with nonblocking communication. The experiments on two real datasets show that the execution times

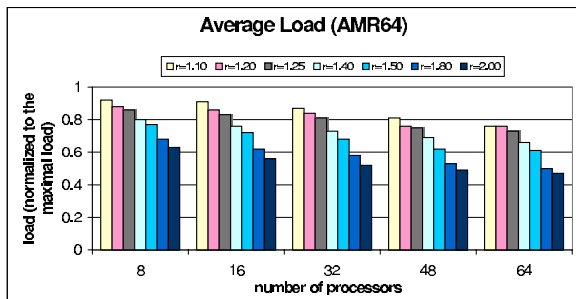


Figure 11. Quality of Load Balancing with Different Parameter Values for AMR64

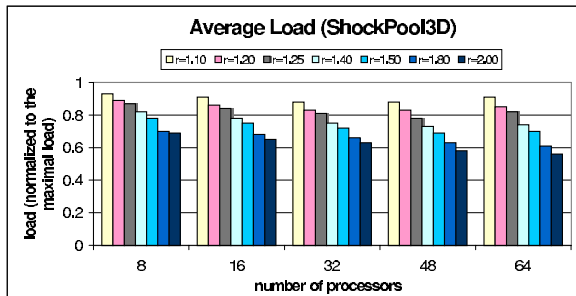


Figure 12. Quality of Load Balancing with Different Parameter Values for ShockPool3D

can be improved by up to 32% and the overhead can be reduced as much as 18%. Further, a detailed sensitivity analysis of the parameter *threshold* used in the DLB scheme is presented. The experiments show that setting this parameter to be around 1.25 would result in the best performance in terms of execution time as well as acceptable quality of load balancing.

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] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Computing*, 7:279–301, October 1989.
- [5] 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.
- [6] F. Lin and R. Keller. The gradient model load balancing methods. In *IEEE Transactions on Software Engineering*, 13(1):8–12, January 1987.
- [7] G. Horton. A multilevel diffusion method for dynamic load balancing. In *Parallel Computing*, (19):209–218, 1993.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] C. Walshaw. Jostle:partitioning of unstructured meshes for massively parallel machines. In *Proc. Parallel CFD'94*, 1994.
- [13] 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.
- [14] J. Rantakokko. A Framework for Partitioning Domains with Inhomogeneous Workload. In *Technical Report 194*, Department of Scientific Computing, Uppsala University, Uppsala, Sweden, 1997.
- [15] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. In *IEEE Concurrency*, pages 22–31, January-March 1999.
- [16] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In *The Bell System Technical Journal*, 49(2):291–307, 1970.
- [17] J. Castanos and J. Savage. Parallel Refinement of Unstructured Meshes. In *Proc. of IASTED International Conference Parallel and Distributed Computing and Systems*, 1999.
- [18] F. Stangenberg. Recognising and Avoiding Thrashing in Dynamic Load Balancing. Technical Report, EPCC-SS94-04, September 1994.
- [19] Z. Lan, V. Taylor, and G. Bryan. Dynamic Load Balancing For Structured Adaptive Mesh Refinement Applications. In *Proc. Of ICPP'2001*, Spain, September, 2001.