Autonomic Proactive Runtime Partitioning Strategies for SAMR Applications*

Yeliang Zhang, Jingmei Yang, Salim Hariri
HPDC Laboratory
University of Arizona
{zhang, jm_yang, hariri}@ece.arizona.edu

Sumir Chandra, Manish Parashar
The Applied Software Systems Laboratory
ECE/CAIP, Rutgers University
{sumir, parashar}@caip.rutgers.edu

Abstract

Dynamic structured adaptive mesh refinement (SAMR) techniques along with the emergence of the computational Grid offer the potential for realistic scientific and engineering simulations of complex physical phenomena. However, the inherent dynamic nature of SAMR applications coupled with the heterogeneity and dynamism of the underlying Grid environment present significant research challenges. This paper presents proactive runtime partitioning strategies based on performance prediction functions that are experimentally formulated in terms of system parameters such as CPU load and available memory. These proactive partitioning strategies form a part of the GridARM autonomic framework which enables self-managing, self-adapting, and self-optimizing SAMR applications on the Grid. Experimental evaluation of the proactive schemes using the 3-D Richtmyer-Meshkov compressible fluid dynamics kernel for different system configurations and workloads demonstrates the improvement in overall runtime performance.

1 Introduction

The emergence of the computational Grid and the potential for seamless aggregation, integration, and interactions has made it possible to conceive a new generation of realistic, scientific and engineering simulations of complex physical phenomena. These next-generation Grid applications will provide new and important insights into complex systems such as interacting black holes and neutron stars, formations of galaxies, subsurface flows in oil reservoirs and aquifers, and dynamic response of materials to detonation.

Dynamically adaptive simulations based on structured adaptive mesh refinement (SAMR) techniques can yield highly advantageous ratios for cost/accuracy when compared to methods based on static uniform approximations, and can be effectively used to enable large-scale, physically realistic scientific and engineering simulations. SAMR techniques start with a coarse base grid with minimum acceptable resolution that covers the entire computational domain. As the simulation progresses, regions in the domain requiring additional resolution are identified and are dynamically refined. Parallel/distributed implementations of SAMR applications lead to interesting research problems in dynamic resource allocation, data-distribution and load balancing, and communication and coordination. Furthermore, the underlying Grid infrastructure is dynamic and heterogeneous in nature. As a result, configuring, managing, and optimizing the execution of dynamic SAMR applications to exploit the underlying computational power of the heterogeneous Grid environment remains a significant challenge.

This paper presents proactive runtime partitioning strategies based on performance prediction functions to optimize the performance of SAMR applications in distributed and dynamic Grid execution environments. In these environments, application performance may be severely degraded due to changes in the available CPU, network load, and memory resources. The performance prediction functions presented in this paper are experimentally formulated [6] in terms of the current system state parameters and estimate the expected performance of a particular application distribution, given the current system state. Though CPU load, available memory, and bandwidth are the primary system parameters considered here, the same strategies can apply to other parameters such as cache size etc., once the performance function is obtained. The partitioning strategies then use the performance functions to identify a redistribution of the application domain that addresses changes in system state and available resources, and maximizes performance.

These proactive partitioning strategies form a part of the GridARM autonomic runtime framework which enables self-managing, self-adapting, and self-optimizing SAMR applications on the Grid. Experimental evaluation of the proactive schemes using the 3-D adaptive Richtmyer-Meshkov compressible fluid dynamics kernel (RM3D[1])

*This research was supported in part by NSF via grants ACI 9984357 (CAREERS), EIA-0103674 (NGS), and EIA-0120934 (ITR).

[1] RM3D has been developed by Ravi Samtaney as part of the virtual test facility at the Caltech ASCI/ASAP Center.

for different system configurations and workloads demonstrates the improvement in overall runtime performance.

The rest of this paper is organized as follows. Section 2 presents an overview of the GridARM framework and its components. Section 3 details the proactive runtime partitioning strategies for distributed SAMR applications. Section 4 describes the experimental evaluation of the proactive schemes for different system configurations and workloads. Section 5 presents concluding remarks and future work.

2 GridARM Autonomic Framework

The overall goal of the GridARM autonomic runtime framework is to reactively and proactively manage and optimize SAMR application execution using current system and application state, online predictive models for system behavior and application performance, and an agent based control network. The framework manages physical Grid resources, allocates them "on-demand", and spatially and temporally maps virtual resources to physical nodes. The conceptual GridARM framework, shown in Figure 1, has three components: (1) services for monitoring Grid resource capabilities and application dynamics and characterizing the monitored state into application units; (2) performance analysis module and deduction engine that define the appropriate optimization strategy based on runtime state and policies; and (3) autonomic runtime manager which is responsible for hierarchically partitioning, scheduling, and mapping application units onto available resources, and tuning execution within the Grid environment.

The monitoring component within the GridARM framework is responsible for detecting conditions under which the parameters affecting the application execution deviate from their acceptable behavior or operation. For example, application performance may degrade severely due to increased computational and/or network load, low available memory, or due to software or hardware failures. The characterization of current state is then used to drive the predictive performance functions and models that can estimate its performance in the near future. The performance analysis module is responsible for describing the behavior of a system component, subsystem or compound system through Performance Functions, developed in previous research [6]. The deduction engine determines the appropriate application reconfiguration strategy and the resources required to repartition work and optimize SAMR performance.

The work presented in this paper builds on earlier GridARM efforts on application-senstive partitioning [1, 3], system-sensitive partitioning [5], Pragma infrastructure [4], and adaptive runtime management [2, 6]. The focus of this paper are the proactive runtime partitioning strategies based on performance prediction functions, formulated from monitored system state parameters, to optimize the performance of SAMR applications in Grid environments.

3 Proactive SAMR Partitioning Strategies

Distributed SAMR applications are CPU-intensive, memory-intensive, and bandwidth-intensive programs. The application performance may degrade severely due to increased CPU and/or network loads, and with reduced available memory. To optimize application performance, the runtime partitioner uses current system parameters obtained using resource monitoring agents, current application state, and performance functions to repartition the entire application domain among processors during runtime. The overall model is as follows. Suppose that the work is to be distributed among K processors.

$$\sum_{i=0}^{K} CR_i = 1 \tag{1}$$

where CR_i represents the combined work partitioning ratio of processor ***i*** defined as follows. The work $\boldsymbol{W}_i$ assigned to the ***i***th processor can be computed as $W_i = CR_i \times W$ where $\boldsymbol{W}$ is the total work. The combined work partitioning ratio can now be computed using system information, i.e. CPU load, available memory and network load.

$$CR_i = w_c R_i^C + w_m R_i^M + w_b R_i^B \tag{2}$$

where R_i^C,R_i^M and R_i^B are work partitioning ratio based on CPU load, available memory, and link bandwidth respectively. w_c, w_m and w_b describe the weights reflecting how important CPU, memory, and bandwidth are and w_c + w_m + w_b = 1. Note that

$$\sum_{i=0}^{K} R_i^C = \sum_{i=0}^{K} R_i^M = \sum_{i=0}^{K} R_i^B = 1 \tag{3}$$

The application and system states on each processor are monitored at runtime. In the case of SAMR applications, application state is defined in terms of the current levels of refinement, the number, shape, and aspect ratio of the refined patches and the dynamism of the application [1, 3]. System state includes CPU availability, available memory, and link bandwidth. The runtime partitioner uses application configuration, state information, and the appropriate performance function to calculate the combined work partitioning ratio CR_i for each processor at runtime. These ratios are then used to redistribute the application domain among processors in subsequent time-steps.

The frequency for monitoring application and system state, recomputing system capacity, and repartitioning the application depends on the rate of system/application dynamics and SAMR adaptation. There are three extreme cases in equation (2), which lead to three different strategies for work partitioning.

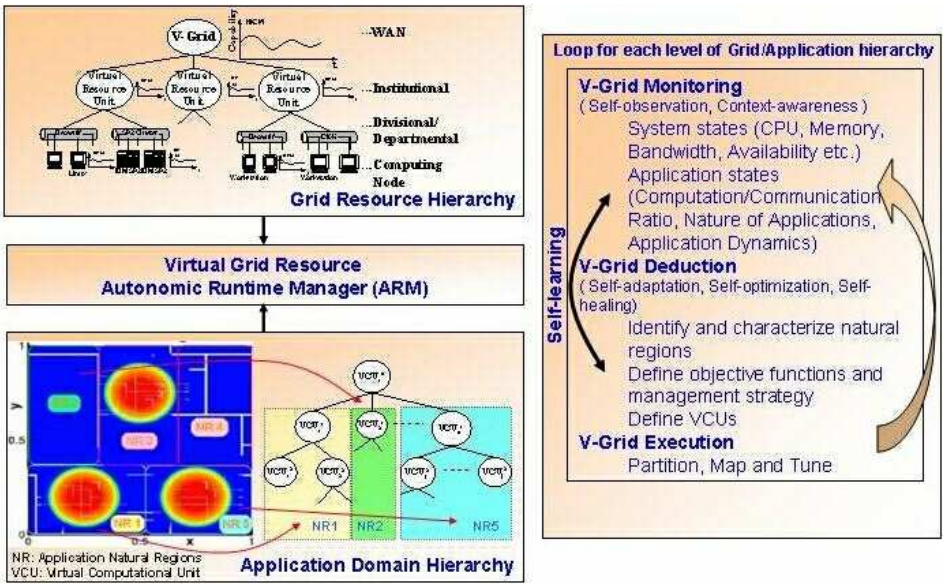

Figure 1. Conceptual model of the GridARM framework

- *CPU-based runtime partitioning:* When w_C is 1 and w_M, w_B are 0, CR_i becomes R_i^C. This strategy partitions the work among processors based on their CPU load status.
- *Memory-based runtime partitioning:* When $w_M$is 1 and w_C, w_B are 0, CR_i becomes R_i^M. This strategy partitions the work among processors based on their available memory.
- *Bandwidth-based runtime partitioning:* When $w_B$is 1 and w_C, w_M are 0, CR_i becomes R_i^B. This strategy partitions the work among processor based on their link bandwidth.

3.1 CPU-based Runtime Partitioning

The performance of parallel SAMR applications in time-shared systems may degrade due to multi-programming. It has been observed that the execution time of a computation-intensive program linearly increases with the number of jobs sharing the same processor. To optimize the SAMR application performance, our CPU-based partitioning strategy uses the CPU current load and application state to repartition the work among processors.

3.1.1 CPU Performance Function Model

Performance Functions (PF) describes the behavior of system or application in terms of changes in one or more of its attributes. We can characterize the relationship between the execution time of a SAMR application and its attributes-application work and refinement level as a time performance function $T = PF_c(W, LV)$. By using the performance function, we can estimate the time to finish work $\boldsymbol{W}$ at refinement level $\boldsymbol{LV}$ for an AMR application on one processor if this processor is dedicated to it. If this processor is shared with other applications, the AMR application would experience longer delay. Let $\boldsymbol{L}_i$ be the load index for processor $\boldsymbol{i}$, which is represented by the length of the CPU waiting queue. In such a multiprogramming case, the execution time of the AMR application can be estimated as a function of the CPU load, application work and refinement level as follows:

$$T_i = T \times L_i = PF_c(W_i, LV_i) \times L_i \tag{4}$$

The time performance function for RM3D is empirically defined as follows:

$$\begin{aligned} T &= PF_c(W, LV) \\ &= a_0 + a_1 W + a_2 LV + a_3 W \times LV \\ &\quad + a_4 W^2 + a_5 LV^2 + a_6 W^2 \times LV \\ &\quad + a_7 W \times LV^2 + a_8 W^2 \times LV^2 \end{aligned} \tag{5}$$

where a_i is heuristic coefficient derived from our previous research [2].

3.1.2 Work Partitioning Algorithm

The average execution times of an AMR application on all processors can be estimated as follows.

$$T_{avg} = \frac{\sum_{i=0}^{K} T_i}{K} \tag{6}$$

where $T_i = PF_c(W_i, LV_i) \times L_i$.

To improve the application performance, the execution time should be as equal as possible on all processors. In doing so, the work partitioning ratio of each processor is adjusted at runtime such that their execution time during the next time steps will be identical within an acceptable tolerance. The adjustment factor associated with each processor is defined as,

$$f_i(t) = \frac{T_{avg}(t)}{T_i(t)} \tag{7}$$

where $\boldsymbol{T_i(t)}$ is the estimated execution time for processor ***i*** at time step ***t***; $\boldsymbol{T_{avg}(t)}$ is the average estimated execution time for all processors at time step ***t***.

Once the adjustment factor is determined, we can compute CPU-based work partitioning ratio of processor ***i*** for the next time step as follows:

$$R_i^C(t+1) = R_i^C(t) \times f_i(t) \tag{8}$$

To make sure that the sum of partitioning ratios on all processors is equal to 1, the new ratio is normalized as follows:

$$R_i^C(t+1)' = \frac{R_i^C(t+1)}{\sum_{i=0}^{K} R_i^C(t+1)} \tag{9}$$

3.2 Memory-based Runtime Partitioning

The performance of distributed SAMR applications may degrade on heavily loaded processors that have little available memory because page faults occur frequently. Thus, the memory-based partitioning strategy optimizes performance by minimizing the number of page faults and balancing work among processors.

3.2.1 Memory Function Model

Using a memory function model, the memory usage of a SAMR application can be characterized as:

$$AM = PF_M(W) \tag{10}$$

where ***AM*** is the amount of memory used by a given work ***W*** of the AMR application.

The memory function for the RM3D application is empirically defined as follows.

$$AM = PF_M(W) = a_0 + a_1 W \tag{11}$$

where: a_0 = 8187.5036; a_1 = 0.1348959 memory needed for each processor at runtime based on the current workload assigned to each processor. In what follows, we describe the algorithm used to proactively partition the work among the processors according to memory availability.

3.2.2 Processor Grouping

In order to efficiently repartition the work among the processors, the memory-based strategy must first identify which processors are heavily loaded, lightly loaded, or in between. Consequently, we divide the processors on which the application is running into different groups according to their available physical memory space.

Let $\boldsymbol{M_i}$ be the amount of available physical memory space on processor ***i***, ***i***=1,. . . ,***K***. We use two-level threshold $\boldsymbol{MT_1}$ and $\boldsymbol{MT_2}$ to describe the memory characteristic of processors. If its available memory $\boldsymbol{M_i}$ is less than $\boldsymbol{MT_1}$, processor ***i*** is heavily loaded and page faults occur frequently. If $\boldsymbol{M_i}$ is greater than $\boldsymbol{MT_2}$, processor ***i*** is lightly loaded and page faults rarely occur. If $\boldsymbol{M_i}$ is between $\boldsymbol{MT_1}$ and $\boldsymbol{MT_2}$, processor ***i*** is moderately loaded and page faults occur occasionally. According to the amount of available memory $\boldsymbol{M_i}$ and two-level threshold $\boldsymbol{MT_1}$ and $\boldsymbol{MT_2}$, processors can be grouped into the following three groups.

- *Low memory group* ($\boldsymbol{X^-}$): If $\boldsymbol{M_i} < \boldsymbol{MT_1}$, processor ***i*** is in group $\boldsymbol{X^-}$ which has little available memory. The number of processors in group $\boldsymbol{X^-}$ is represented by $\boldsymbol{N^-}$.
- *High memory group* ($\boldsymbol{X^+}$): If $\boldsymbol{M_i} > \boldsymbol{MT_2}$, processor ***i*** is in group $\boldsymbol{X^+}$ which has large amount of available memory. The number of processors in group $\boldsymbol{X^+}$ is represented by $\boldsymbol{N^+}$
- *Border memory group* (***X***): If $\boldsymbol{M_i}$ is between $\boldsymbol{MT_1}$ and $\boldsymbol{MT_2}$, processor ***i*** belongs to group ***X***. The number of processors in group ***X*** is represented by ***N***.

To optimize the execution of the application, some work on the processors in group $\boldsymbol{X^-}$ should be transferred to the processors in group $\boldsymbol{X^+}$ and the work of the processors in group ***X*** should be kept unchanged. This will lead to better performance due to reduction in number of page faults. There are three important cases that have to be dealt with.

- Certain processors have too little available memory and certain processors have excess available memory, i.e. both $\boldsymbol{N^+}$ and $\boldsymbol{N^-}$ are greater than zero.
- Some processors have excess available memory while no processors have little available memory. $\boldsymbol{N^+}$ is greater than zero and $\boldsymbol{N^-}$ is equal to zero.
- Certain processors have little available memory while no processors have excess available memory. $\boldsymbol{N^-}$ is greater than zero and $\boldsymbol{N^+}$ is equal to zero.

In the first case, the work assigned to processors in group $\boldsymbol{X^-}$ should be partially transferred to the processors in group $\boldsymbol{X^+}$. In the second case, there are processors with excess available memory but no processor is heavily loaded.

Therefore the work assignment of all processors would not change. In the third case, there is an absolute shortage of memory but no processor has excess available memory. Thus, we keep the current assignment of work unchanged.

3.2.3 Work Partitioning Algorithm

After identifying the amount of available memory in each processor and dividing them into groups, one must calculate how much work should ideally be transferred from group $\boldsymbol{X}^-$ to group $\boldsymbol{X}^+$. Let $\boldsymbol{W}^-$ be the whole work assigned to processors in group $\boldsymbol{X}^-$.

$$W^- = \sum_{i=0}^{N^-} W_i \tag{12}$$

To avoid overload the processors in group $\boldsymbol{X}^+$, we initially transfer part of the whole work $\boldsymbol{W}^-$. We use $\boldsymbol{P}_1$ to represent the transferring percentage and the work to be transferred is $W^- \times P_1$. The remaining work on processors in group $\boldsymbol{X}^-$is $W^- \times (1 - P_1)$. To balance the work transfer, the work should be moved to processors in group X^+ as evenly as possible. So we define $\boldsymbol{U}^-$ as the unit of work being transferred to one processor.

$$U^- = \frac{W^- \times P_1}{N^+} \tag{13}$$

In order to achieve the desired optimization, a processor in group $\boldsymbol{X}^+$ must guarantee that its available memory space is greater than $\boldsymbol{MT}_1$ after it accepts the work transfer. In doing so, it must estimate the memory usage of its current work $\boldsymbol{W}_i$ and the memory usage of its work after the transfer. Let the work of processor $\boldsymbol{i}$ after the transfer be $\boldsymbol{W}_i^{'}$ and $W_i' = W_i + U^-$. For RM3D application, these two memory usages can be estimated by using the memory function shown in Equation (10). In order to guarantee its available memory greater than $\boldsymbol{MT}_1$ after the transfer, the following condition must be met for each processor in group $\boldsymbol{X}^+$.

$$M_i - (PF_M(W_i') - PF_M(W_i)) > MT_1 \tag{14}$$

We sort the processors in group $\boldsymbol{X}^+$ in the ascending order of available memory $\boldsymbol{M}_i$.Thus we check with the processor having least available memory first. Then the processor with the second least available memory is checked and so forth. In what follows, we will analyze the behavior of the algorithm in terms of four cases:

Upon substituting $W_i' = W_i + U^-$ into (13), we check for processor $\boldsymbol{i}$ in group $\boldsymbol{X}^+$ if condition in (13) can be met.

1. *Case 1:* If the condition in (13) cannot be met, we reduce the size of the work to be transferred. Let $\boldsymbol{P}_2$ be the reduction percentage and the work to be transferred then becomes $U^- \times (1 - P_2)$. Then we check the condition in (13) again by substituting $W_i' = W_i + U^- \times (1 - P_2)$.
   (a) *Case 1-1:* If the condition can be met, the new work of processor $\boldsymbol{i}$ after the transfer would be $W_i' = W_i + U^-$.
   (b) *Case 1-2:* If the condition still cannot be met, more reduction would be made. If after the whole unit of work $\boldsymbol{U}^-$ is reduced and the condition still cannot be made, which means processor $\boldsymbol{i}$ cannot accept any additional work, the current work of processor $\boldsymbol{i}$ would be kept unchanged $W_i' = W_i$.
2. *Case 2:* If the condition in (13) can be met after transferring unit of work $\boldsymbol{U}^-$ to processor $\boldsymbol{i}$, we will check if there is remaining work left from the previous processor ***i-1***. Let $\boldsymbol{W}_{rm}$***(i-1)*** be the remaining work from the previous processor ***i-1***. If there exists $\boldsymbol{W}_{rm}$***(i-1)***, we will try to transfer it to processor $\boldsymbol{i}$ as well. Then the work of processor $\boldsymbol{i}$ after the transfer becomes $W_i' = W_i + U^- + W_{rm}(i-1)$. The condition in (13) is checked again with the new $\boldsymbol{W}_i^{'}$.
   (a) *Case 2-1:* If the condition can be met, $W_i' = W_i + U^- + W_{rm}(i-1)$ would be the new work of processor $\boldsymbol{i}$ after the transfer.
   (b) *Case 2-2:* If the condition cannot be met, the same reduction method would be applied to $\boldsymbol{W}_{rm}$***(i-1)***. After all the checking and reduction on processor $\boldsymbol{i}$, the remaining work from processor $\boldsymbol{i}$ to next processor $\boldsymbol{W}_{rm}$***(i)*** would be updated. If there is still some work left after checking all the processors in group $\boldsymbol{X}^+(W_{rm}(N^+) > 0)$, it would be assigned back to the processors in group $\boldsymbol{X}^-$ according to their contributions to $\boldsymbol{W}^-$.

After the work transfer, each processor has a new work $\boldsymbol{W}_i^{'}$. Then the new memory-based work partitioning ratios can be computed as follows:

$$WP_i^M = \frac{W_i'}{\sum_{i=0}^{K} W_i'} \tag{15}$$

4 Experimental Results

The autonomic proactive runtime partitioning system has been integrated into the GrACE (Grid Adaptive Computational Engine) data management framework for parallel/distributed SAMR applications. The autonomic runtime partitioning strategies are evaluated using the RM3D CFD kernel on Beowulf clusters at Rutgers University and University of Arizona. We establish three scenarios to evaluate our proactive partitioning approaches.

- *Lightly loaded scenario*: Among the processors executing the RM3D application, 75% processors are lightly loaded and the other 25% are heavily loaded.

- *Moderately loaded scenario*: Among the processors executing the RM3D application, 50% processors are lightly loaded and the other 50% are heavily loaded.
- *Heavily loaded scenario*: Among the processors executing the RM3D application, 25% processors are lightly loaded and the other 75% are heavily loaded.

4.1 CPU-based Proactive Partitioning

In this subsection, we quantify the performance gain that can be achieved by using the CPU-based proactive partitioning approach to adapt to CPU load dynamics. A synthetic program is used to control the CPU load dynamics among the processors and establish the three different load scenarios discussed previously. we compare the performance of the RM3D application with and without the CPU-based proactive partitioning approach.

Figure 2 presents CPU load situation on 16 processors and the corresponding work assignment of the RM3D application with and without CPU load adaptation under the moderately loaded scenario. The CPU load is measured as the length of the CPU waiting queue and is monitored by system monitoring tool at runtime. The work is the size of computation point set of the RM3D application. Without CPU load adaptation, the work is assigned almost evenly among the processors. However, by using proactive CPU partitioning algorithm, work is assigned to processors based on its CPU load status. Tables 1, 2, 3, and 4 present the performance gain of RM3D with CPU-based proactive partitioning strategy for different base size and different number of processors under different load scenarios.

Table 1. CPU-based proactive partitioning performance gain on 8 processors. (Base grid size: 64*16*16)

Scenarios	Execution time w/o CPU adaptation (seconds)	Execution time with CPU adaptation (seconds)	Percentage Improvement
Lightly loaded	2618.2	1560.87	40.38%
Moderately loaded	2706.84	1964.87	27.41%
Heavily loaded	2727.51	2127.32	22%

The above results show that RM3D experiences longer delays when some processors are heavily loaded. Without the CPU-based proactive partitioning algorithm, the application work is partitioned equally among the processors regardless of their CPU load status that leads to a longer application execution time. However with our CPU-based proactive runtime partitioning strategy the application work is partitioned according to the processors' CPU load status and the performance of RM3D could be significantly improved. For example, in Table 1, for lightly loaded scenario we can obtain 40% percentage improvement. The results also demonstrate that better performance can be achieved under lightly and moderately loaded scenarios. The reason is that under heavily loaded scenario most processors are heavily loaded and there are not enough lightly loaded processors to accept more work. Furthermore, better performance can be obtained with large number of processors. For example, in Table 2, 65.8% percentage improvement is obtained on 16 processors under lightly loaded scenario with the same base grid size of 64*16*16 as shown in Table 1. With more processors, the work assigned to each processor would be reduced.

Table 2. CPU-based proactive partitioning performance gain on 16 processors. (Base grid size: 64*16*16)

Scenarios	Execution time w/o CPU adaptation (seconds)	Execution time with CPU adaptation (seconds)	Percentage Improvement
Lightly loaded	2126.06	727.17	65.8%
Moderately loaded	2301.15	1641.73	28.66%
Heavily loaded	2378.25	1624.15	31.71%

Table 3. CPU-based proactive partitioning performance gain on 32 processors (Base grid size: 128*32*32)

Scenarios	Execution time w/o CPU adaptation (seconds)	Execution time with CPU adaptation (seconds)	Percentage Improvement
Lightly loaded	4908.79	2901.1	40.9%
Moderately loaded	4976.78	3378.65	31.35%
Heavily loaded	5170.52	4140.56	20.45%

4.2 Memory-based Proactive Partitioning

In this subsection, we quantify the performance gain that can be achieved if the work assignment takes into consideration the amount of available memory at each processor. In our experiment, the memory availability of processors is controlled by a synthetic memory consuming program.

Figures 3, 4, and 5 demonstrate the memory availability on 8 processors and the corresponding work assignment by using the memory-based proactive partitioning approach for three different scenarios. During the execution of the RM3D application, the memory-based proactive partitioner

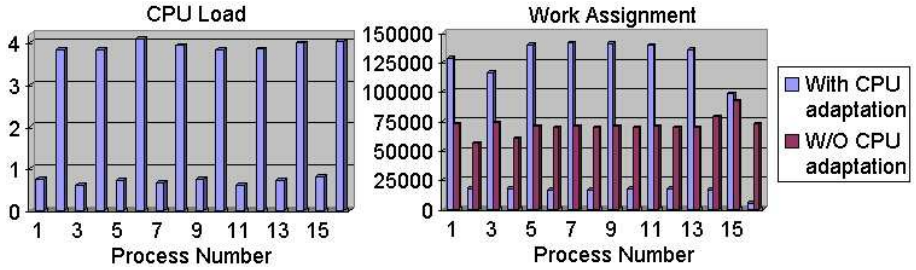

Figure 2. Moderately loaded scenario on 16 processors: CPU load distribution (left) and work assignment for 16 processors (right)

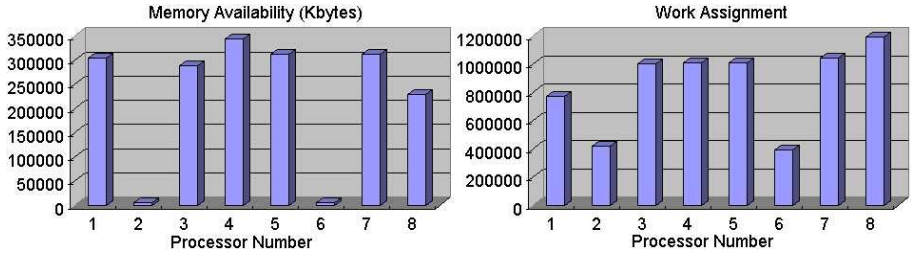

Figure 3. Lightly loaded scenario on 8 processors: Memory availability (left) and work assignment for memory-based proactive partitioning algorithm (right)

Table 4. CPU-based proactive partitioning performance gain on 64 processors. (Base grid size: 128*32*32)

Scenarios	Execution time w/o CPU adaptation (seconds)	Execution time with CPU adaptation (seconds)	Percentage Improvement
Lightly loaded	4904.78	2827.29	42.36%
Moderately loaded	5049.72	5049.72	35.02%
Heavily loaded	5119.89	3587.89	29.92%

Table 5. Memory-based proactive partitioning performance gain on 8 processors (Base grid size: 128*32*32)

Scenarios	Execution time w/o memory adaptation (seconds)	Execution time with memory adaptation (seconds)	Percentage Improvement
Lightly loaded	6922.14	5210.87	24.72%
Moderately loaded	15890.47	7401.61	53.42%
Heavily loaded	16962.1	8284.84	51.16%

assigns larger work to processors with high available memory. On the other hand, processors with little available memory are assigned relatively smaller workload.

Table 5 compares the performance of RM3D with and without the memory-based proactive partitioning strategy and shows the substantial improvement due to memory-based partitioning. Without the memory-based partitioning algorithm, the application work is partitioned evenly among the processors. Due to large number of page faults on heavily loaded processors, the RM3D application experiences long delays. However, with the memory-based algorithm, the application work is reassigned to processors according to memory availability, resulting in reduced overall execution times. The above results also show that better performance can be achieved under moderately and heavily loaded scenarios since most processors have very little available memory and the page faults occur frequently, resulting in extremely long application delays. Using memory-based partitioning strategy to assign work based on processors' memory availability, the heavily loaded processors will be assigned less computation and, consequently, the performance can be significantly improved.

5 Conclusions and Future Work

This paper presented proactive runtime partitioning strategies based on performance prediction functions to optimize the performance of SAMR applications in distributed

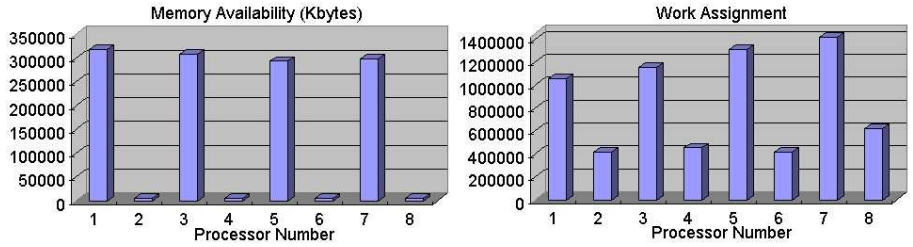

Figure 4. Moderately loaded scenario on 8 processors: Memory availability (left) and work assignment for memory-based proactive partitioning algorithm (right)

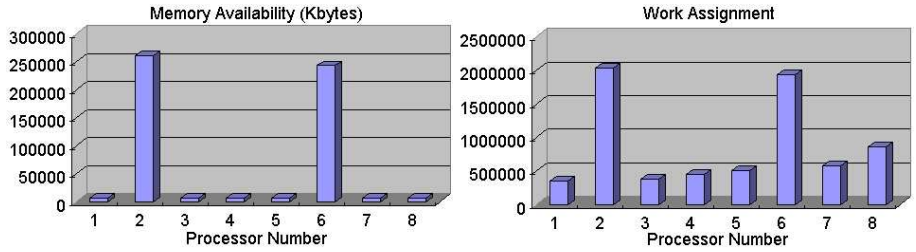

Figure 5. Heavily loaded scenario on 8 processors: Memory availability (left) and work assignment for memory-based proactive partitioning algorithm (right)

and dynamic Grid execution environments. The performance prediction functions are experimentally formulated in terms of the current monitored state of the system (CPU load, available memory, and bandwidth). These proactive partitioning strategies form a part of the GridARM autonomic runtime framework which enables self-managing, self-adapting, and self-optimizing SAMR applications on the Grid. The experimental evaluation for the proactive strategies using the 3-D adaptive Richtmyer-Meshkov compressible fluid dynamics application kernel for different system configuration and workloads demonstrated the improvement in overall runtime performance of SAMR applications. The CPU-based scheme achieves performance improvement of up to 65.8% and the memory-based strategy can achieve performance improvement of up to 53%. Note that further improvements in performance can be achieved for larger numbers of processors and under heavy load. Future work in this research aims to define a hybrid strategy that combines several system parameters such as CPU, memory, and bandwidth, and addresses the issues about selection consideration and sensitivity of system weights and their effect on overall performance of the hybrid strategy.

References

[1] S. Chandra and M. Parashar. "ARMaDA: An Adaptive Application-Sensitive Partitioning Framework for Structured Adaptive Mesh Refi nement Applications", *Proc. of Parallel and Distributed Computing Systems* (PDCS 02), Cambridge, MA, pp. 446–451, November 2002.

[2] S. Chandra, S. Sinha, M. Parashar, Y. Zhang, J. Yang, and S. Hariri. "Adaptive Runtime Management of SAMR Applications", *Proc. of High Performance Computing* (HiPC 02), LNCS, India, Vol. 2552, pp. 564–574, December 2002.

[3] S. Chandra, J. Steensland, M. Parashar, and J. Cummings. "An Experimental Study of Adaptive Application Sensitive Partitioning Strategies for SAMR Applications", *Proc. of 2nd LACSI Symposium* (best poster SC'01), October 2001.

[4] M. Parashar and S. Hariri. "PRAGMA: An Infrastructure for Runtime Management of Grid Applications", *Proc. of NSF NGS Program Workshop, IEEE/ACM IPDPS*, Fort Lauderdale, FL, CDROM, 8 pages, April 2002.

[5] S. Sinha and M. Parashar. "Adaptive Runtime Partitioning of AMR Applications on Heterogeneous Clusters", *Proc. of Cluster Computing*, Newport Beach, CA, IEEE Computer Society Press, pp. 435–442, October 2001.

[6] H. Zhu, M. Parashar, J. Yang, Y. Zhang, S. Rao, and S. Hariri. "Self Adapting, Self Optimizing Runtime Management of Grid Applications using PRAGMA", *Proc. of NSF NGS Program Workshop, IEEE/ACM 17th IPDPS*, Nice, France, CDROM, 7 pages, April 2003.