Autonomic Runtime Manager for Large Scale Adaptive Distributed Applications

Jingmei Yang, Huoping Chen, Salim Hariri
University of Arizona
Emails:
{jm_yang,hpchen,hariri}@ece.arizona.edu

Manish Parashar
Rutgers, The State University of New Jersey
Email: parashar@caip.rutgers.edu

Abstract

Large-scale distributed applications are highly adaptive and heterogeneous in terms of their computational requirements. The computational complexity associated with each computational region or domain varies continuously and dramatically both in space and time throughout the whole life cycle of the application execution. Consequently, static scheduling techniques are inefficient to optimize the execution of these applications at runtime. In this paper, we present an Autonomic Runtime Manager (ARM) that uses the application spatial and temporal characteristics as the main criteria to self-optimize the execution of distributed applications at runtime. The wildfire spread simulation is used as a running example to demonstrate the ARM effectiveness to control and manage the application's execution. The behavior of the wildfire simulation depends on many complex factors that contribute to the adaptive and heterogeneous behaviors such as fuel characteristics and configurations, chemical reactions, balances between different modes of heat transfer, topography, and fire/atmosphere interactions. Consequently, the application execution cannot be predicted a priori and that makes static parallel or distributed algorithms very inefficient. The ARM is implemented using two modules: 1) Online Monitoring and Analysis Module, and 2) Autonomic Planning and Scheduling Module. The online monitoring and analysis module interfaces with different kinds of application and system sensors that collect information to accurately determine the current state of the fire simulation in terms of the number and locations of burning and unburned cells as well as the states of the resources, and decides whether the autonomic planning and scheduling module should be invoked. The autonomic planning and scheduling module uses the resource capability models as well as the current state of the computations to repartition the whole computational workload into available processors. Our experimental results show that by using ARM the performance of the wildfire simulation has been improved by 45% when compared with a static partitioning algorithm. We also evaluate the performance of ARM using two partitioning strategies. One approach is to partition the wildfire simulation domain into Natural Regions (NR), where each region has the same temporal and spatial characteristics (e.g., burned (NR1), burning (NR2), and unburned regions (NR3)), and schedule each region into available processors. The second approach is to view the wildfire domain as a graph and use a graph partitioning tool (e.g., ParMetis tool) to partition the graph into different domains.

1. Introduction

Large-scale distributed applications are highly adaptive and heterogeneous in terms of their computational requirements. The computational complexity associated with each computational region or domain varies continuously and dramatically both in space and time throughout the whole life cycle of the application execution. An example of such an application is a wildfire simulation of a national park, which simulates the wildfire spread behavior by taking into considerations many factors such as fuel characteristics and configurations, chemical reactions, balances between different modes of heat transfer, topography, and fire/atmosphere interactions. The computational load associated with "burning" cells is much larger than that for "unburned" cells that will lead to load imbalance conditions if that difference in computational requirement is not taken into consideration when the application computational load assigned to the processors at runtime. Consequently, static scheduling techniques are very inefficient to optimize the execution of applications that continuously change their temporal and spatial characteristics. The wildfire spread behavior is an example of this class of adaptive distributed applications.

Optimizing the performance of parallel applications through load balancing algorithms is well studied in the literature and they can be classified as either static or

dynamic algorithms. The compile-time static approaches [8][9][10][11] assign work to processors before the computation starts and can be efficient if we know how the computations will progress a priori. On the other hand, if the workload cannot be estimated beforehand, dynamic load balancing strategies have to be used [12] [13][14][15]. For example, the diffusion-based methods [14][15] divide the processor pool into small and overlapping neighborhoods. The underloaded processor requests work levels from its neighbors and then determine how much work to request from each neighbor. Because of the overlapping neighborhood, the work will eventually diffuse throughout the system to achieve global load balancing. However, the local schemes are inadequate for heterogeneous and adaptive distributed applications because they lack a global view of the current state of the application. Some global schemes [16][17][18] predict future performance based on past information or based on some prediction tools, such as Network Weather Service (NWS)[19]. In [18], the authors use the predicted CPU information provided by NWS to guide scheduling decisions. Dome [17] remaps the computation based on the time each processor spends on computing during the last computational phase. Other optimization techniques are based on application-level scheduling [20][21][22]. AppLeS in [20][21] assumes the application performance model is static and provided by users and GHS system [22] assumes the total computation load of applications is a constant.

Some researchers [23][24][25][26][27][28] have explored the load balancing issues for adaptive applications. The applications in [23][24][25] are adaptive mesh refinement either on unstructured or structured grid and load balancing is achieved by repartitioning the computations among processors after each refinement phase. [26] assumes the adaptation are infrequent and the load remains relatively stable between adaptations. [27][28] introduce a load balancing framework for asynchronous adaptive applications. However the wildfire simulation represents such applications which are loosely synchronous and constantly adapting, and requires more adaptive and efficient runtime optimization techniques. In this paper, we present an Autonomic Runtime Manager (ARM) that continuously monitors and analyzes the current state of the application as well as the computing and networking resources and then makes the appropriate planning and scheduling actions. The ARM control and management activities are overlapped with the application execution to minimize the overhead incurred using the ARM runtime optimization algorithm.

The reminder of this paper is organized as follows: Section 2 gives an overview of the ARM system and a detailed analysis of the wildfire simulation. Results from the experimental evaluation of the ARM runtime optimization are presented in Section 3. We compare the performance of the wildfire simulation with and without the ARM system by using different partitioning approaches. A conclusion and outline of future research directions are presented in Section 4.

2. Autonomic Runtime Manager (ARM): An Overview

The Autonomic Runtime Manager (ARM) is responsible for controlling and managing the execution environment for large-scale distributed adaptive applications at runtime. Once the application is running, ARM will optimize the application execution to improve performance dynamically. The ARM main modules include (see Figure 1): 1) Online Monitoring and Analysis and 2) Autonomic Planning and Scheduling. Online monitoring and analysis module interfaces with different kinds of sensors that collect information about the fire propagation and environmental data as well as the information about the states of the underlying resources. These current state conditions are then used by the ARM to steer the simulation into the direction that maximizes the performance or any other desired property (e.g., accuracy). Based on the objectives of the analysis (e.g., accurate vs. approximate but fast simulations), the planning engine will use the resource capability models as well as the performance models associated with the computations, and the knowledge repository to select the appropriate models and partitions for each region (empirical-based, physics-based) and then decompose the computational workloads into schedulable Computational Units (*CUs*). Based on the availability of computing resources and their access policies, the scheduler will dynamically schedule the *CUs* on available Resource Units (*RUs*) that can be clusters of high performance workstations, massive parallel computers, and/or distributed/shared memory multiprocessor systems.

It is to be noted here that the ARM hides the underlying heterogeneity of the execution environment from the application and can interface the application to different types of execution models and different types of resources harnessing the maximum utilization of features and capabilities of the underlying environment. In this paper, we will use the wildfire simulation as a running example to explain the main operations of the ARM modules and the performance gains that can be achieved.

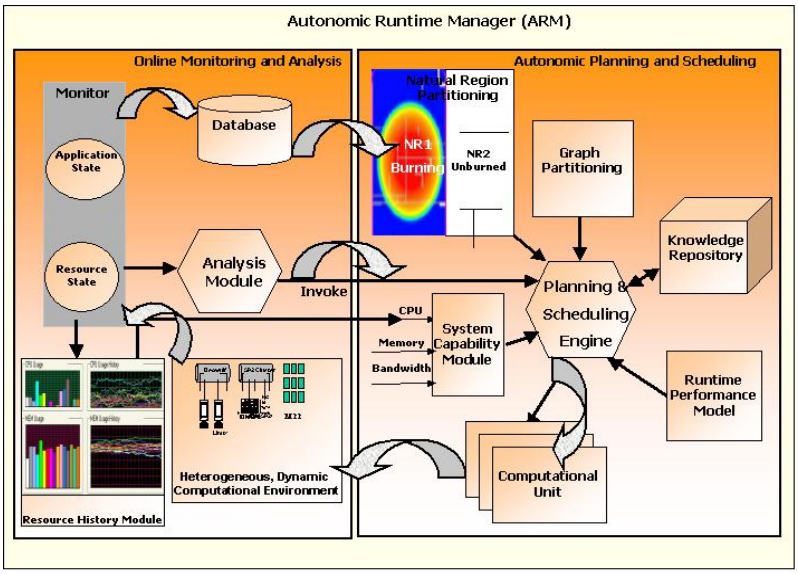


Figure 1. **Autonomic Runtime Manager (ARM) Architecture**

2.1. An Illustrative Example - Wildfire Simulation

In the wildfire simulation model, the entire area is represented as a 2-D cell-space composed of cells of dimensions l x b (l: length, b: breadth). For each cell, there are eight major wind directions N, NE, NW, S, SE, SW, E, W as shown in Figure 2. The weather and vegetation conditions are assumed to be uniform within a cell, but may vary in the entire cell space. A cell interacts with its neighbors along all the eight directions.

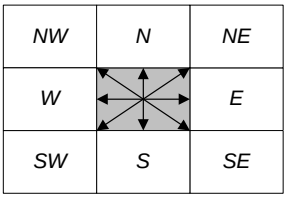


Figure 2. **Fire direction after ignition**

When a cell is ignited, its state will change from "unburned" to "burning". During its "burning" phase, the fire will propagate to its eight neighbors along the eight directions. The direction and the value of the maximum fire spread rate within the burning cell can be computed using Rothermel's fire spread model [3], which takes into account the wind speed and direction, the vegetation type, the fuel moisture and terrain type, such as slope and aspect, in calculating the fire spread rate. The fire behavior in eight directions including the spread rate, the time for fire to spread to eight neighbors, and the flame length could be determined using the elliptical growth model developed by Anderson [4]. When the simulation time advances to the ignition times of neighbors, the neighbor cells will ignite and their states will change from "unburned" to "burning". In a similar way, the fire would propagate to the neighbors of these cells. With different terrain, vegetation and weather conditions, the fire propagation could form very different spread patterns within the entire region.

The wildfire simulation model used in this paper is based on fireLib[1], which is a C function library for predicting the spread rate and intensity of free-burning wildfires. It is derived directly from the BEHAVE [2] fire behavior algorithms for predicting fire spread in two dimensions, but is optimized for highly iterative applications such as cell- or wave-based fire growth simulation. We parallelized the sequential version of the fire simulation using MPI[5]. This parallelized fire simulation divides the entire cell space among multiple processors such that each processor works on its own portion and exchanges the necessary data with each others after each simulation time step. The parallel wildfire simulation is a loosely synchronous iterative application. Each processor performs the computation on part of the whole space, maintains the ignition map which is the ignition times of all cells, and proceeds to the next ignition cell as the simulation advances.

Because of the synchronization between iterations, the execution time during one iteration is effectively determined by the execution time of the slowest, or most heavily loaded processor. Consequently, the application performance will be severely degraded if the computational loads on all the processors are not well balanced at runtime. In the following sections, we present

how the ARM system can effectively maximize the performance of the wildfire simulation at runtime.

2.2. Online Monitoring and Analysis

The online monitoring module monitors the current state of the fire simulation in terms of the number and the locations of burning cells and unburned cells. By profiling the application behavior at runtime, the computation time spent on each iteration can be obtained. In addition, the online monitoring module monitors the states of the resources involving in the execution of the fire simulation, such as the CPU load, available memory, network load etc. The runtime state information is stored in a database. The online analysis module analyzes the runtime information and the load imbalance of the wildfire simulation and then determines whether or not the current allocation of workloads need to be changed.

We use a metric that we refer to as the *Imbalance Ratio (IR)* to quantify the load imbalance that can be computed as:

$$IR(t) = \frac{Max_{i=0}^{P-1}(T_{comp}(p_i,t)) - Min_{i=0}^{P-1}(T_{comp}(p_i,t))}{Min_{i=0}^{P-1}(T_{comp}(p_i,t))} \times 100\% \quad (1)$$

Where $T_{comp}(p_i, t)$ is the computation time at time step t on processor p_i, and, P denotes the number of processors. So $Max_{i=1}^{P-1}T_{comp}(p_i,t)$ represents the computation time spent on the most heavily-loaded processor and $Min_{i=1}^{P-1}T_{comp}(p_i,t)$ is the computation time spent on the most lightly-loaded processor. We use a predefined threshold $IR_{threshold}$ to measure how severe the load imbalance value is. If IR exceeds the specified threshold, the imbalance conditions are considered severe and repartitioning is required. Then the autonomic planning and scheduling module will be invoked to carry the appropriate actions to repartition the simulation workload.

The selection of the threshold $IR_{threshold}$ can significantly impact the effectiveness of the repartitioning algorithm. If the threshold chosen is too low, too many load repartitions will be triggered and the high overhead produced outweigh the expected performance gains. On the other hand, when the threshold is high, the load imbalance cannot be detected quickly and consequently the performance improvement will be reduced. In the experimental results subsection, we show how we can experimentally determine the appropriate threshold value.

2.3. Autonomic Planning and Scheduling

The autonomic planning and scheduling module partitions the whole fire simulation domain into several sub-domains based on the state of the application and the current loads on the underlying processors. To reduce the rescheduling overhead, we use a dedicated processor to run the autonomic planning and scheduling engine and overlap that with the worker processors executing the workload of the distributed wildfire simulation. Once the new partition assignments are finalized, a message is sent to all the worker processors to read the new assignments after they are done with the current computations. Consequently, the ARM runtime optimization activities are completely overlapped with the application execution and the overhead is less than 4% as will be discussed later.

2.3.1 Partitioning Strategy

We have developed and experimented with two partitioning strategies in our ARM prototype.

Natural Region Partitioning Approach:

This method uses the runtime information associated with the application current state to partition the fire simulation domain into several Natural Regions (*NRs*), where each region has the same temporal and spatial characteristics (e.g., burning (*NR1*), and unburned regions (*NR2*)) and assign each region to processors according to their capabilities.

Let *ACW(t)* be the predicted Application Computational Workload at time *t* in terms of the number of cells in the "burning" region $N_B(t)$ and "unburned" region $N_U(t)$. *ACW(t)* can be defined as follows:

$$ACW(t) = N_B(t) + N_U(t) \quad (2)$$

We use Processor Load Ratio (*PLR*) metric to quantify the computing capacity for each processor such that

$$\sum_{i=0}^{P-1} PLR(p_i,t) = 1 \quad (3)$$

Let $L(P_i,t)$ be the system load on processor P_i at time t which can be measured by using the CPU queue length. If $L(P_i,t) <= 1$, then there is only one process running on processor P_i and the estimated execution time of one burning cell on processor P_i at time t is given by:

$$T_B(p_i,t) = T_B \quad (4)$$

where T_B is the estimated computation time of one burning cell on a dedicated processor. If $L(P_i,t) > 1$, there are other applications running on processor P_i. The expected computation time of one burning cell will be longer due to multiprogramming and can be estimated as follows:

$$T_B(p_i,t) = L(p_i,t) * T_B \quad (5)$$

Consequently, the average execution time for a burning cell at time t is computed as:

$$T_{B_{avg}}(t) = \frac{\sum_{i=0}^{P-1} T_B(p_i,t)}{P} \quad (6)$$

To balance the load on each processor, we define an adjustment factor, Processor Allocation Factor (*PAF*), which is inversely proportional to the processor execution

time with respect to the average execution time. The *PAF* for processor p_i can be computed as:

$$PAF(p_i,t) = \frac{T_{B_{avg}}(t)}{T_B(p_i,t)} \quad (7)$$

By normalizing the *PAF*, we could obtain the *PLR* for processor p_i as follows:

$$PLR(p_i,t) = \frac{PAF(p_i,t)}{\sum_{i=0}^{P-1} PAF(p_i,t)} \quad (8)$$

Therefore the Processor Computational Load (*PCL*) to be assigned to processor p_i is given as:

$$PCL(p_i,t) = PLR(p_i,t) \times ACW(t) \quad (9)$$

Then the corresponding workload of each natural region will be assigned to processors according to their *PCL*s.

Graph Partitioning Approach

The wildfire simulation domain can be represented as an undirected weighted graph *G(V, E)* of *V* vertices and *E* edges. The cells of the domain are the vertices of the graph and an edge exists between two graph vertices if these cells are neighbors. Each graph vertex has a weight associated with it, which indicates the workload of the corresponding cell. Burning cells have a larger weight than the unburned cells because of the difference in their computational complexities. Each edge of the graph also has a weight that models the interprocessor communication. Thus a graph partitioning for the fire simulation domain yields the assignment of cells to processors.

There are several graph partitioning tools that can be used [24][29]. The objective of the graph partitioning algorithm is to find a reasonable load balance that minimizes the edgecut and the interprocessor communication; where edgecut is defined as the total weight of edges that cross the partitions. In our ARM prototype, we have implemented the ParMetis[29] graph partitioning tool. The ParMetis is a widely used graph partitioning program that is developed at the University of Minnesota. We use *ParMETIS_V3_PartKway()* routine to initially partition the fire simulation domain into sub-domains and use *ParMETIS_V3_AdaptiveRepart()* routine to repartition the domain when the distribution results in load imbalance ratio larger than the predetermine threshold. *ParMETIS_V3_AdaptiveRepart* routine makes use of a Unified Repartitioning Algorithm [30] for adaptive repartitioning that combines the best characteristics of remapping and diffusion-based repartitioning schemes. Repartitioning is performed on the graph using the weights of the vertices and edges that are computed by the online monitoring and analysis module to reflect the current state of the fire simulation in terms of the locations of burning cells and unburned cells.

2.3.2 Predictive Model

As discussed in section 2.1, the wildfire simulation maintains an ignition map to store the ignition time of each cell. As the fire simulation proceed, it will calculate the time that fire spreads from the current burning cell to its eight neighbors and update the ignition times of those neighbors accordingly. Therefore, at any given time step, based on the ignition map of the domain, we can predict what are the next *N* cells that will be ignited.

The ARM system implemented sensors to collect the ignition time changes of cells at run time and stores them into the ARM database. The autonomic planning and scheduling module will compare the ignition times of cells with the current time and obtain the next *N* cells that will burn. Thus, the predicted application computational workload $ACW_{pred}(t)$ can be computed as follows:

$$ACW_{pred}(t) = N_B(t) + N_{B\,pred}(t) + (N_U(t) - N_{B\,pred}(t)) \quad (10)$$

where $N_B(t)$ and $N_U(t)$ are the number of burning cells and unburned cells at time step *t* as defined in Equation (2). $N_{Bpred}(t)$ denotes the predicted next *N* burning cells at time step *t*. The automatic planning and scheduling module will use the predicted application workload to partition the fire simulation domain and assign the corresponding workload to available processors.

In the next section, we evaluate the performance of these two partitioning techniques for different problem sizes and different number of processors.

3. Performance Evaluation

The experiments were performed on two problem sizes for the fire simulation. The first problem size is a 256*256 cell space with 65536 cells. The second problem has a 512*512 cell domain with 262144 cells. To introduce a heterogeneous fire patterns, the fire is started in the southwest region of the domain and then propagates northeast along the wind direction until it reaches the edge of the domain. In order to make the evaluation for different problem sizes accurate, we maintain the same ratio of burning cells to 17%; that is the total number of burning cells when the simulation terminates is about 17% of the total cells for both problem sizes.

We begin with an examination of the effects of the imbalance ratio threshold on application performance. We ran the fire simulation with a problem size of 65536 on 16 processors and varied the $IR_{threshold}$ values to determine the best value that minimizes the execution time.

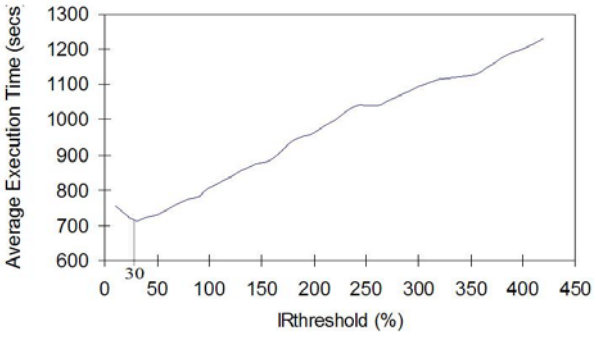


Figure 3. **The sensitivity of the fire simulation to the $IR_{threshold}$ value**

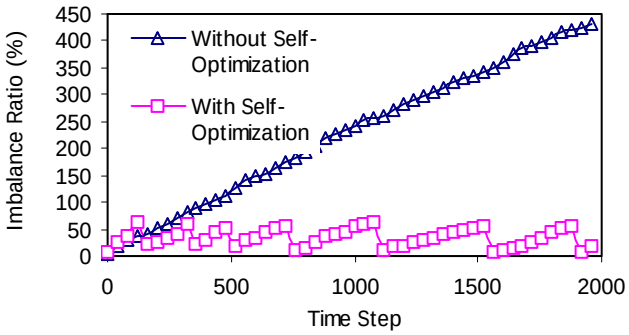


Figure 4. **Imbalance ratios for 2000 time steps of the fire simulation for a problem size = 65536, number of processors = 16, $IR_{threshold}$ = 50%**

Table 1.
Workload Distribution Quality for Problem Size with 65536 cells

Runtime Optimization Approach	8 Processors		16 Processors	
	Max/Min Computation	Std Dev	Max/Min Computation	Std Dev
Static Scheduling	1841/259 sec	626	921/129	316
Natural Region Approach	940/ 693 sec	86	464/299	53
Graph Partitioning Approach	923/633 sec	104	463/285	59

Table 2.
Workload Distribution Quality for Problem Size with 262144 cells

Runtime Optimization Approach	16 Processors		32 Processors	
	Max/Min Computation	Std Dev	Max/Min Computation	Std Dev
Static Scheduling	16339/2042 sec	5289	8950/1031 sec	2097
Natural Region Approach	8375/6603 sec	382	3662/2340 sec	467
Graph Partitioning Approach	8441/5163 sec	807	3744/2038 sec	570

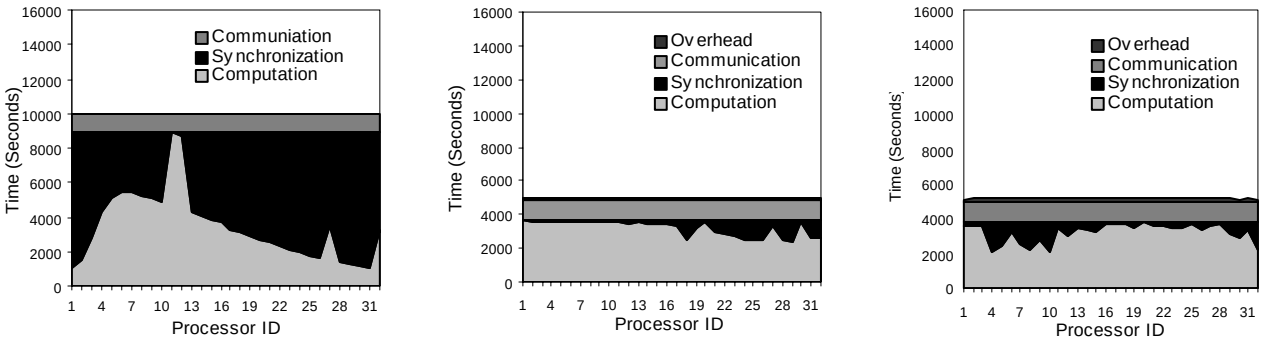
The results of this experiment are shown in Figure 3. The execution times are taken as the average for three runs. We observed that the best execution time, 713 seconds, was achieved when the $IR_{threshold}$ is equal to 30%. We need to do more research to determine the relationship between the number of processors and problem size to the threshold value. In the experimental results, we assume that once the load imbalance ratio becomes above 30%, the repartitioning is triggered.

It is important to notice that the online monitoring and analysis and autonomic planning and scheduling activities are all carried out on a separate processor and completely overlapped with the worker processor computations.

Figure 4 shows how the imbalance ratios increase linearly as the simulation progresses using static partitioning algorithm and compare that with our runtime optimization algorithm. For example, at 2000 time steps of the simulation, the imbalance ratio for static scheduling is about 450% while it is around 25% in our approach. In fact, using our approach, the imbalance ratio is kept bound within a small range.

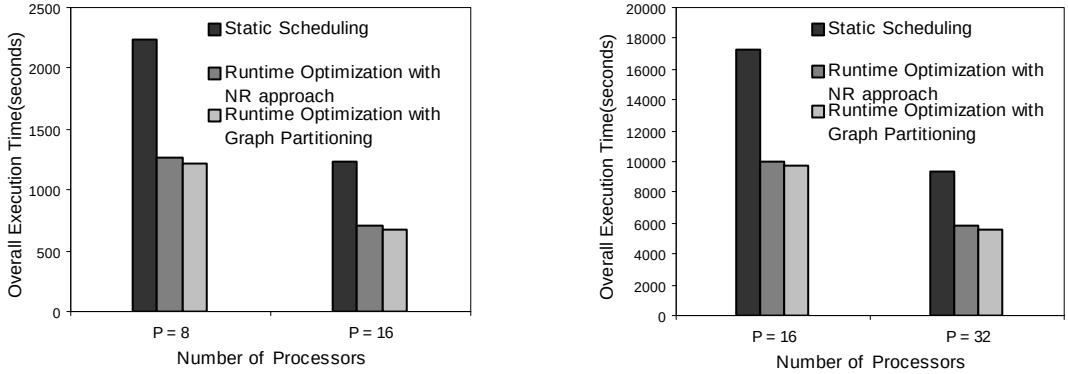
We now evaluate the ARM's performance in optimizing the execution time of the wildfire simulation using the following three metrics: 1) the quality of the workload distribution for all processors; 2) the overall execution time of the wildfire simulation; and 3) the overhead incurred by the ARM system.

First we evaluate the quality of the workload distribution. In Figure 5a, we see a processor-by-processor breakdown of the wildfire simulation's performance on 32 processors with a problem size of 262144 cells. Most of the computation is clustered within the range from processor 5 through 15. Figure 5b and 5c



(a) (b) (c)

Figure 5. **Breakdown of the Execution time for different optimization techniques applied to an application with 22144 cells running on 32 processors. (a) Static scheduling. (b) Runtime optimization with natural region partitioning approach. (c) Runtime optimization with graph partitioning approach.**



(a) (b)

Figure 6. **Overall performance of different optimization approaches on different processor configurations. (a) Problem size 65536 cells. (b) Problem size 262144 cells.**

provide the results for runtime optimization using the natural region partitioning approach and the graph partitioning approach, respectively. As we can see, for both approaches, the computational load is distributed evenly across the processors. The computation time has a standard deviation of roughly 467 for natural region partitioning approach, and 570 for the graph partitioning approach, compared with 2097 for the static scheduling approach. Table 1, 2 summarizes the standard deviations of the computation time for two problem sizes on different processor configurations.

Our second metric is the overall execution time of the wildfire simulation. Figure 6 demonstrates the overall execution time with different runtime optimization approaches as well as the static scheduling algorithm on two problem sizes and different processor configurations. For problem size 65536 cells on 8 processors, the ARM system with graph partitioning approach provides an improvement of 45 percent over static scheduling, 3 percent over runtime optimization with the natural region partitioning approach. On 16 processors, these numbers are 45 percent and 6 percent. For problem size 262144 cells on 16 processors, the numbers are 44 percent and 3 percent, while, on 32 processors, they are 41 percent and 5 percent.

As we can see in Tables 1 and 2, the natural region partitioning approach is more successful than the graph partitioning approach in terms of data distribution quality with a smaller standard deviation for the processor computation times. However, as shown in Figure 6, the overall execution time with the natural region partitioning approach is slightly larger than that of the graph partitioning approach. The reason is that graph partitioning approach partitions the graph in a way that both balances the workload and minimizes the communication time between sub-domains, which eventually reduces the overall execution time.

Table 3.
Overhead imposed by ARM system for problem size 65536 cells

Number of Processors	ARM overhead		
	Data Collecting Time	Reading New Partition Time	Prcnt.
8	1.8 sec	8.9 sec	0.8%
16	1.9 sec	15.5 sec	2.4%

Table 4.
Overhead imposed by ARM system for problem size 262144 cells

Number of Processors	ARM overhead		
	Data Collecting Time	Reading New Partition Time	Prcnt.
16	21.6 sec	70 sec	0.9%
32	23.1 sec	165.9 sec	3.4%

Finally we show that the overhead incurred by the ARM system is small and does not have a negative impact on the application performance. In our implementation, one processor is dedicated to run the two ARM modules: the online monitoring and analysis and the autonomic planning and scheduling modules. That means the ARM computations are completely overlapped with the computations of the distributed fire simulation. Consequently, the only overhead incurred in our approach is the time spent by the fire simulation to send its current state information to ARM sensors and the time spent in reading the new assigned simulation loads to the worker processors. To quantify the overhead on the whole application performance, we conducted experiments to measure the overhead introduced by our algorithm. Tables 3 and 4 summarize the overheads caused by the ARM system for two problem sizes running on different processor configurations. In all cases, the overhead cost is less than 4% of the overall execution time.

4. Conclusions and Future Work

In this paper, we described an architecture for an autonomic runtime manager that maximizes the parallel execution of large scale applications at runtime by continuously monitoring and analyzing the state of the computations and the underlying resources, and efficiently exploit the physics of the application. In our approach, the physics of the problem (e.g., how fire propagates and how it is impacted by wind speed and direction, fuel, moisture, etc.) and its current state are the main criterion used in our runtime optimization algorithm. The Autonomic Runtime Manager (ARM) main modules are the Online Monitoring and Analysis, and Autonomic Planning and Scheduling modules. The execution of the ARM modules is overlapped with the distributed application being self-optimized to reduce the overhead. We show that the overhead of our ARM system is less than 4%. We have also evaluated the ARM performance on a large wildfire simulation for different problem sizes. The experimental results show that using the ARM runtime optimization, the performance of the wildfire simulation can be improved by up to 45% when compared to the static parallel partitioning algorithm. We also evaluated different partitioning methods such as the natural region partitioning approach and the graph partitioning approach.

References

[1] <http://www.fire.org>

[2] P. L. Andrews, “BEHAVE: Fire Behavior Prediction and Fuel Modeling System - BURN Subsystem”, *Part 1. General Technical Report INT-194.* Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Research Station; 1986. 130 p.

[3] R. C. Rothermel, “A Mathematical Model for Predicting Fire Spread in Wildland Fuels”, *Research Paper INT-115.* Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1972. 40 p.

[4] H. E. Anderson, “Predicting Wind-Driven Wildland Fire Size and Shape”, *Research Paper INT-305. Ogden,* UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1983. 26 p.

[5] M. Snir, S. Otto, S. Huss-Lederman, and D. Walker, “*MPI the Complete Reference*”, MIT Press, 1996

[6] S. Hariri, L. Xue, H. Chen etc., "AUTONOMIA: an autonomic computing environment", *Conference Proc. of the 2003 IEEE IPCCC*

[7] S. Hariri, B. Khargharia, H. Chen, Y. Zhang, B. Kim, H. Liu and M. Parashar; "The Autonomic Programming Paradigm", Submitted to *IEEE computer* 2004.

[8] P.E. Crandall, and M. J. Quinn, “Block Data Decomposition for Data-parallel Programming on a Heterogeneous Workstation Network”, *2nd IEEE HPDC*, pp. 42-49, 1993

[9] Y. F. Hu, and R. J. Blake, “Load Balancing for Unstructured Mesh Applications”, *Parallel and Distributed Computing Practices*, vol. 2, no. 3, 1999

[10] S. Ichikawa, and S. Yamashita, “Static Load Balancing of Parallel PDE Solver for Distributed Computing Environment”, *Proc. 13th Int’l Conf. Parallel and Distributed Computing Systems*, pp. 399-405, 2000

[11] M. Cierniak, M. J. Zaki, and W. Li, “Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations”, Computer *Journal*, vol. 40, no. 6, pp. 256-372, 1997

[12] M. Willebeek-LeMair, and A.P. Reeves, “Strategies for Dynamic Load Balancing on Highly Parallel Computers”, *IEEE Trans. Parallel and Distributed Systems*, vol.4, no. 9, pp. 979-993, Sept. 1993.

[13] F. C. H. Lin, and R. M. Keller, “The Gradient Model Load Balancing Method”, *IEEE Trans. on Software Engineering*, vol. 13, no. 1, pp. 32-38, Jan. 1987

[14] G. Cybenko, “Dynamic Load Balancing for Distributed Memory Multiprocessors”, *Journal of Parallel and Distributed Computing*, vol. 7, no.2, pp. 279-301, 1989

[15] G. Horton, “A Multi-Level Diffusion Method for Dynamic Load Balancing”, *Parallel Computing*, vol.19, pp. 209-229, 1993

[16] N. Nedeljkovic, and M. J. Quinn, “Data-Parallel Programming on a Network of Heterogeneous Workstations”, *1st IEEE High Performance Distributed Computing Conference*, pp. 152-160, Sep. 1992

[17] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, “ Dome: Parallel Programming in a Heterogeneous Multi-User Environment,” *Proc. 10th Int’l Parallel Processing Symp.*, pp. 218-224, 1996.

[18] C. Liu, L. Yang, I. Foster, and D. Angulo, “Design and Evaluation of a Resource Selection Framework for Grid Applications”, *11th IEEE High Performance Distributed Computing Conference*. Edinburgh. Scotland, 2002

[19] R. Wolski, N. Spring, and J. Hayes, “The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing”, *Journal of Future Generation Computing Systems*, pp. 757-768, 1998

[20] F. Berman, R. Wolski, S. Figueria, J. Schopf, and G. Shao, “Application-Level Scheduling on Distributed Heterogeneous Networks”, *Supercomputing’96*, 1996

[21] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, “Adaptive Computing on the Grid Using AppLeS”, *IEEE Trans. on Parallel and Distributed Systems*, vol 14, no 4, pp 369--382, April, 2003

[22] X.-H. Sun and M. Wu, "Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling," *Proc. of 2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April, 2003.

[23] L. Oliker, and R. Biswas, “Plum: Parallel Load Balancing for Adaptive Unstructured Meshes”, *Journal of Parallel and Distributed Computing*, vol. 52, no. 2, pp. 150-177, 1998

[24] C. Walshaw, M. Cross, and M. Everett, “Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes”, *Journal of Parallel and Distributed Computing*, vol. 47, pp. 102-108, 1997

[25] Y. Zhang, J. Yang, S. Chandra, S. Hariri and M. Parashar, “Autonomic Proactive Runtime Partitioning Strategies for SAMR Applications”, *Proceedings of the NSF Next Generation Systems Program Workshop, IEEE/ACM 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, USA, 8 pages. April 2004

[26] M. A. Bhandarkar, R. K. Brunner, L. V. Kale, “Run-time Support for Adaptive Load Balancing”, *Proceedings of the 15th IPDPS 2000 Workshops on Parallel and Distributed Processing, Lecture Notes In Computer Science*; Vol. 1800**,** pp. 1152 – 1159, 2000

[27] K. Barker, N. Chrisochoides, **“**An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications**”,** *Proceedings of the ACM/IEEE SC2003 Conference*, 2003

[28] K. Barker, A. Chernikov, N. Chrisochoides, K. Pingali, **“**A Load Balancing Framework for Adaptive and Asynchronous Applications**”,** *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 183-192, Feb. 2004

[29] K. Schloegel, G. Karypis, V. Kumar, “Parallel Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes”, *Technical Report 97-014*, Univ. of Minnesota, 1997

[30] K. Schloegel, G. Karypis, V. Kumar, “A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations”, *Proceedings of the International Conference on Supercomputing, 2000*