Self-optimization of Large Scale Wildfire Simulations

Jingmei Yang[1], Huoping Chen[1], Salim Hariri[1], and Manish Parashar[2]

[1] University of Arizona,
{jm_yang, hpchen, hariri}@ece.arizona.edu
[2] Rutgers, The State University of New Jersey,
parashar@caip.rutgers.edu

Abstract. The development of efficient parallel algorithms for large scale wildfire simulations is a challenging research problem because the factors that determine wildfire behavior are complex. These factors make static parallel algorithms inefficient, especially when large number of processors is used because we cannot predict accurately the propagation of the fire and its computational requirements at runtime. In this paper, we propose an Autonomic Runtime Manager (ARM) to dynamically exploit the physics properties of the fire simulation and use them as the basis of our self-optimization algorithm. At each step of the wildfire simulation, the ARM decomposes the computational domain into several natural regions (e.g., burning, unburned, burned) where each region has the same temporal and special characteristics. The number of burning, unburned and burned cells determines the current state of the fire simulation and can then be used to accurately predict the computational power required for each region. By regularly monitoring and analyzing the state of the simulation, and using that to drive the runtime optimization, we can achieve significant performance gains because we can efficiently balance the computational load on each processor. Our experimental results show that the performance of the fire simulation has been improved by 45% when compared with a static portioning algorithm.

1 Introduction

For over fifty years, attempts have been made to understand and predict the behavior of wildfires. However, the factors that determine wildfire behavior are complex and the computational loads associated with regions in the domain vary greatly both in time and space. Load balancing and efficient parallel execution of these simulations on large numbers of processors present significant challenges.

Optimizing the performance of parallel applications through load balancing is well studied and can be classified as either static or dynamic. The static approaches [3][4] assign work to processors before the computation starts and can be efficient if we know how the computations will progress a priori. If the workload cannot be estimated beforehand, dynamic load balancing strategies have to be used [5][6][7][8]. Some global schemes [9][10] predict future performance based on past information or based on some prediction tools such as Network Weather Service (NWS)[11]. Other optimization techniques are based on application-level scheduling [12][13]. AppLeS

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 615 – 622, 2005.

[12] assumes the application performance model is static and provided by users and GHS system[13] assumes the applications computation load is a constant.

There are a few techniques that assume adaptive applications [14][15][16]. However, the wildfire simulation is a continuously changing application and requires adaptive and efficient runtime optimization techniques. In this paper, we present an Autonomic Runtime Manager (ARM) that continuously monitoring the computing requirements of the application, analyzing the current state of the application as well as the computing and networking resources and then making the appropriate planning and scheduling actions at runtime. The ARM control and management activities are overlapped with the application execution to minimize the overhead incurred.

The reminder of this paper is organized as follows: Section 2 gives a brief overview of the ARM system and a detailed analysis of the wildfire simulation. Results from the experimental evaluation of the ARM system are presented in Section 3. A conclusion and outline of future research directions are presented in Section 4.

2 Autonomic Runtime Manager (ARM) Architecture

The Autonomic Runtime Manager(ARM) is responsible for controlling and managing the execution for large-scale applications at runtime. The ARM main modules include (Fig. 1): 1) Online Monitoring and Analysis Module and 2) Autonomic Planning and Scheduling Module. The online monitoring and analysis module monitors the state of the application and underlying system and determines whether the online planning engine should be invoked. The planning and scheduling engine uses the resource capability models as well as performance models associated with the computations, and the knowledge repository to select the appropriate models and partitions for each region and then decompose the computational workloads into schedulable Computational Units (*CUs*). In this paper, we will use the wildfire simulation as a running example to explain the main operations of the ARM modules.

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 length x breadth. For each cell, there are eight major wind directions as shown in Fig. 2. 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. The direction and the value of the maximum fire spread rate within the burning cell can be computed using Rothermel's fire spread model [2]. When the simulation time advances to the ignition times of neighbors, the neighbor cells will ignite. 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.

Our wildfire simulation model is based on fireLib [1], which is a C function library for predicting the spread rate and intensity of free-burning wildfires. We parallelized the sequential fire simulation using MPI. This parallelized fire simulation divides the entire cell space among multiple processors such that each processor works on its own

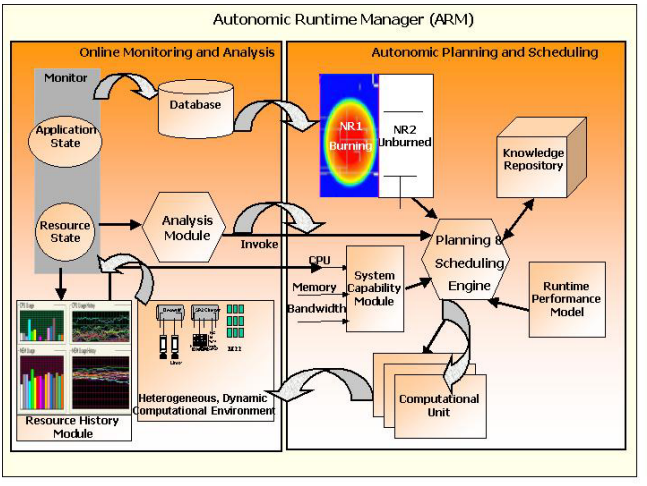

Fig. 1. Autonomic Runtime Manager (ARM) architecture

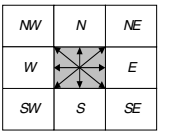

Fig. 2. Fire direction after ignition

portion and exchanges the necessary data with each other's after each simulation time step. At each time step, each processor computes and maintains the ignition maps of the 8 neighbors of the current ignited cell. Then the ignition map changes are exchanged between processors.

In our current implementation, a coordinator processor gathers the ignition map changes from each worker processor and then broadcasts them to all processors. Since there are only a few cells whose ignition times are changed at each time step, we believe the communication overhead with the coordinator is low. Thus the estimated execution time at time t for processor P_i can be defined as follows:

$$T_i(t) = T_{comp}(P_i,t) + T_{comm}(P_i,t) \tag{1}$$

where $T_{comp}(P_i,t)$ and $T_{comm}(P_i,t)$ are the computation and communication time at step t for processor P_i, respectively.

The application computational workload (ACW) of the simulation is defined as:

$$ACW(t) = N_B(t)T_B + N_U(t)T_U \tag{2}$$

where $N_B(t)$ and $N_U(t)$ are the number of burning and unburned cells at time t; T_B and T_U are the estimated computation times of each burning and unburned cell. $T_B > T_U$ because burning cells are more computation intensive than unburned cells, which contribute significantly to the imbalance conditions at runtime. Let α_i be the fraction of the workload assigned to processor P_i, it will be given a workload of $\alpha_i \times ACW(t)$. Therefore, the expected computation time for processor P_i can be defined as follows:

$$T_{comp}(P_i,t) = \alpha_i(N_B(t)T_B + N_U(t)T_U) = N_B(P_i,t)T_B + N_U(P_i,t)T_U \tag{3}$$

where $N_B(P_i, t)$ and $N_U(P_i, t)$ are the number of burning cells and unburned cells assigned to processor P_i at time step t.

The communication cost $T_{comm}(P_i,t)$ includes the time required for data gathering, synchronization and broadcasting, which can be defined as follows:

$$T_{comm}(P_i,t) = T_{gather}(P_i,t) + T_{sync}(P_i,t) + T_{bcast}(t) \tag{4}$$

Data gathering operation can be started once the computation is finished. The data gathering time of processor P_i at time step t is given by:

$$T_{gather}(P_i,t) = mT_{Byte}N_c(P_i,t) \tag{5}$$

where m is the message size in bytes sent by one cell, $N_C(P_i,t)$ is the number of cells assigned to processor P_i whose ignition time are changed during the time step t, and T_{Byte} is the data transmission time per byte. It is important to notice that broadcast operation can only start after the coordinator processor receives the data from all processors. Consequently, the data broadcasting time can be defined as:

$$T_{bcast}(t) = mT_{Byte}\sum_{i=0}^{P-1} N_c(P_i,t) \tag{6}$$

Then, the estimated execution time of the wildfire simulation on processor i can be computed as:

$$T_{total_i} = \sum_{t=1}^{N_t} T_i(t) \tag{7}$$

where N_t is the number of time steps performed by the wildfire simulation.

2.2 Online Monitoring and Analysis

The online monitoring module collects the information about the wildfire simulation state, such as the number and the location of burning cells and unburned cells, and the computation time for the last time step. At the same time, it monitors the states of the underlying resources, 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 load imbalance of the wildfire simulation and then determines whether or not the current allocation of workload needs to be changed.

Figure 3 shows the breakdown of the execution time and type of activities performed by four processors. Processor P_0 has the longest computation time because it is handling a large number of burning cells. Consequently, all the other three processors have to wait until processor P_0 finishes its computation and then the data broadcasting can be started. To balance the workload, the online analysis module should quickly detect large imbalance and invoke the repartitioning operation. To quantify the imbalance, we introduce a metric, Imbalance Ratio (IR) that can be computed as:

$$IR(t) = \frac{Max_{i=0}^{P-1}(T_{comp}(P_i,t)) - Min_{i=0}^{P-1}(T_{comp}(P_i,t))}{Min_{i=0}^{P-1}(T_{comp}(P_i,t))} \times 100\% \tag{8}$$

We use a predefined threshold $IR_{threshold}$ to measure how severe the imbalance is. If $IR(t) > IR_{threshold}$, the imbalance is considered severe and repartitioning is required. Then the automatic planning and scheduling module will be invoked to carry the appropriate actions to reparation the simulation workload.

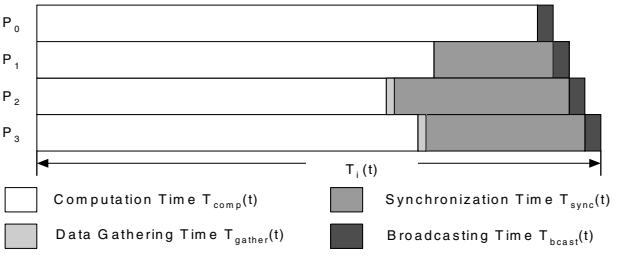

Fig. 3. The breakdown of the processor execution time at time step *t*

The selection of the threshold $IR_{threshold}$ can significantly impact the effectiveness of the self-optimization approach. If the threshold chosen is too low, too many load repartitioning will be triggered and the high overhead produced outweigh the expected performance gains. On the other hand, when the threshold is high, the imbalance conditions cannot be detected quickly. In the experimental results subsection, we show how we can experimentally choose this threshold value.

2.3 Autonomic Planning and Scheduling

The autonomic planning and scheduling module partitions the whole fire simulation domain into several natural regions (burning, unburned) based on its current state and then assigns them to processors by taking into consideration the states of the processors involved in the fire simulation execution. To reduce the rescheduling overhead, we use a dedicated processor to run the ARM self-optimizing algorithm and overlap that with the worker processors that compute their assigned workloads. Once the new partition assignments are finalized, a message is sent to all the worker processors to read the new assignments once they are done with the current computations. Consequently, the ARM self-optimization activities are completely overlapped with the application computation and the overhead is very minimum less than 4% as will be discussed later.

3 Experimental Results

The experiments were performed on two problem sizes for the fire simulation. One is a 256*256 cell space with 65536 cells. The other is 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. To make the evaluation accurate, we maintain total number of burning cells during the simulation 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. The results of this experiment are shown in Fig. 4. We observed that the best execution time, 713 seconds, was achieved when the $IR_{threshold}$

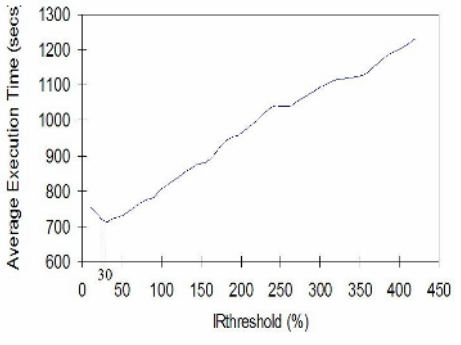

Fig. 4. The sensitivity of the fire simulation to the $IR_{threshold}$ value

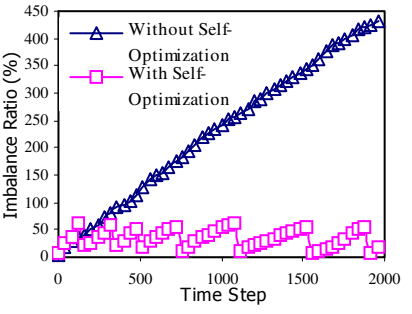

Fig. 5. Imbalance ratios for 2000 time steps of the fire simulation, problem size = 65536, number of processors = 16, $IR_{threshold}$ = 50%

is equal to 30%. Figure 5 shows how the imbalance ratio increases as the simulation progresses using static partitioning algorithm and compares that with our self-optimization algorithm. For example, at time step 2000, the imbalance ratio in the static parallel algorithm is about 450% while it is around 25% in our approach. Using our approach, the imbalance ratio is kept bound within a small range.

Figure 6 shows the computation time for each processor at time steps 1, 300 and 600 with and without the ARM self-optimization. For example, at time step 1, the computation load is well balanced among most processors for both static partitioning and self-optimization. However, as shown in Fig. 6(a), at time step 300, processor P_0 and P_1 experience longer computation times while other processors keep the same computation time as before. This is caused by having many burning cells assigned to these two processors P_0 and P_1. At time step 600, more and more cells on processor P_0 and P_1 are burning and the maximum computation time of 0.24 seconds is observed for P_1. However, if we apply the ARM self-optimization algorithm, all processors finish their computations around the same time for all the simulation time steps (see Fig. 6 (b)). For example, the maximum execution time of 0.1 seconds is observed for processor P_2 at time step 600, which is 58% reduction in execution time when compared to the 0.24 seconds observed for the static portioning algorithm.

Tables 1 and 2 summarize the comparison of the execution time of the fire simulation with and without our self-optimization algorithm. Our experimental results show that the self-optimization approach improves the performance by up to 45% for a problem size of 262144 cells on 16 processors. We expect to get even better performance as the problem size increases because it will need more simulation time and will have more burning cells than smaller problem sizes.

In our implementation, one processor is dedicated to the autonomic planning and scheduling operations while all the worker processors are running the simulation loads assigned to them. Consequently, our self-optimization algorithm will not have high overhead impact on the fire simulation performance. The only overhead incurred is the time that ARM sensors collect the runtime information and the time that worker processors read new assigned simulation loads. To quantify the overhead on the whole system, we conducted experiments to measure the overhead. Based on our

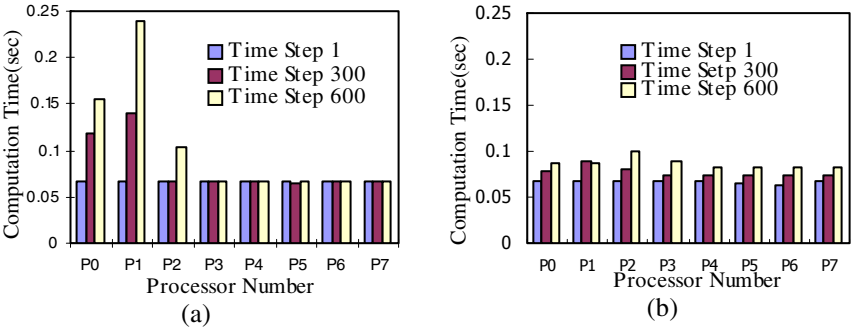

Fig. 6. Computation times of different time steps on 8 processors. Each group of adjacent bars shows the computation time of time step 1, 300 and 600, respectively. (a) Without self-optimization (b) With self-optimization

Table 1. Performance comparison for the fire simulation with and without self-optimization for different number of processors, problem size = 65536, and $IR_{threshold}$ = 30%

Number of Processors	Execution Time without static partitioning (sec)	Execution Time with Self-Optimization (sec)	Performance Improvement
8	2232.11	1265.94	43.29%
16	1238.87	713.17	42.43%

Table 2. Performance comparison for the fire simulation with and without self-optimization for different number of processors, problem Size = 262144, and $IR_{threshold}$ = 30%

Number of Processors	Execution Time without Self-Optimization (sec)	Execution Time with Self-Optimization (sec)	Performance Improvement
16	17276.02	9486.3	45.09%
32	9370.96	5558.55	40.68%

experiments, we observed that the overhead cost is less than 4% of the total execution time for both problem sizes of the fire simulation.

4 Conclusions and Future Work

In this paper, we described an Autonomic Runtime Manager that can self-optimize 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 problem being optimized. In our approach, the physics of the problem and its current state are the main criterion used to in our self-optimization algorithm. The activities of the ARM modules are overlapped with the algorithm being self-optimized to reduce the overhead. We show that the overhead of our self-optimization algorithm is less than 4%. We have also evaluated the ARM

performance on a large wildfire simulation for different problem sizes and different number of processors. The experimental results show that using the ARM self-optimization, the performance of the wildfire simulation can be improved by up to 45% when compared to the static parallel partitioning algorithm.

References

1. <http://www.fire.org>
2. Rothermel, R. C.: 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)
3. Ichikawa, S., Yamashita, S.: Static Load Balancing of Parallel PDE Solver for Distributed Computing Environment. Proc. 13th Int'l Conf. Parallel and Distributed Computing Systems (2000) 399-405
4. Cierniak, M., Zaki, M. J., Li, W.: Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations. Computer J., vol. 40, no. 6(1997) 256-372
5. Willebeek-LeMair, M., Reeves, A.P.: Strategies for Dynamic Load Balancing on Highly Parallel Computers. IEEE Trans. Parallel and Distributed Systems, vol.4, no. 9 (1993) 979-993
6. Lin, F. C. H., Kelle, R. M. r: The Gradient Model Load Balancing Method, IEEE Trans. on Software Engineering, vol. 13, no. 1 (1987) 32-38
7. Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. J. Parallel and Distributed Computing, vol. 7, no.2 (1989) 279-301
8. Horton, G.: A Multi-Level Diffusion Method for Dynamic Load Balancing. Parallel Computing, vol.19 (1993) 209-229
9. Nedeljkovic, N., Quinn, M. J.: Data-Parallel Programming on a Network of Heterogeneous Workstations. 1st IEEE HPDC (1992) 152-160
10. Arabe, J., Beguelin, A., Lowekamp, B., Seligman, E., Starkey, M., Stephan, P.: Dome: Parallel Programming in a Heterogeneous Multi-User Environment. Proc. 10th Int'l Parallel Processing Symp. (1996) 218-224
11. Wolski, R., Spring, N., Hayes, J.: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Journal of Future Generation Computing Systems (1998) 757-768
12. Berman, F., Wolski, R., Casanova, H., Cirne, Dail, W., H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D.: Adaptive Computing on the Grid Using AppLeS. IEEE Trans. on Parallel and Distributed Systems, vol. 14, no. 4(2003) 369--382
13. Sun, X.-H., Wu, M.: Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling. Proc. of 2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)(2003)
14. Oliker, L., Biswas, R.: Plum: Parallel Load Balancing for Adaptive Unstructured Meshes", J. Parallel and Distributed Computing, vol. 52, no. 2(1998) 150-177
15. Walshaw, C., Cross, M., Everett, M.: Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. J. Parallel and Distributed Computing, vol. 47(1997)102-108
16. Zhang, Y., Yang, J., Chandra, S., Hariri, S., Parashar, M.: 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 (2004)