Automatic Selection of Loop Scheduling Algorithms Using Reinforcement Learning

Sumithra Dhandayuthapani[1,2], Ioana Banicescu[1,2], Ricolindo L. Cariño[2], Eric Hansen[1], Jaderick P. Pabico[1,2] and Mahbubur Rashid[1,2]
[1]Dept. of Computer Science and Engineering, Mississippi State University, PO Box 9637
[2]Center for Computational Sciences–ERC, Mississippi State University, PO Box 9627
Mississippi State MS 39762
(sumithra@erc, ioana@cse, rlc@erc, hansen@cse, jade@erc, mr174@erc).msstate.edu

Abstract

This paper presents the design and implementation of a reinforcement learning agent that automatically selects appropriate loop scheduling algorithms for parallel loops embedded in time-stepping scientific applications executing on clusters. There may be a number of such loops in an application, and the loops may have different load balancing requirements. Further, loop characteristics may also change as the application progresses. Following a model-free learning approach, the learning agent assigned to a loop will select from a library the best scheduling algorithm for the loop during the lifetime of the application. The utility of the learning agent is demonstrated by its successful integration into the simulation of wave packets – an application arising from quantum mechanics. Results of statistical analysis using pairwise comparison of means on the running time of the simulation with and without the learning agent validate the effectiveness of the agent in improving the parallel performance of the simulation.

1. Introduction

The presence of loops with independent iterations is a rich source of parallelism in most scientific applications. To obtain high application performance and to take advantage of parallelism, these loops are executed on multiple processors. During parallel execution, factors such as the frequent communication among processes, the load imbalance among processors, the overhead incurred during communication, and the synchronizations during computations, can cause performance degradation. Among these, load imbalance is the most dominant factor, being caused by the interactive effects of irregularities in problem, algorithmic, and systemic characteristics. Problem irregularities are mainly brought about by non-uniform distribution of application data among processors, while algorithmic irregularities are due to different conditional execution paths within the loop. The systemic irregularities are due to unpredictable memory access times, cache misses, and interrupts. To address the load imbalance problem, various load assignment and scheduling algorithms have been developed.

Selecting an effective and efficient scheduling algorithm from the currently available ones to achieve load balancing for a specific scientific application executing in an unpredictable environment is a difficult task. The difficulty is due to the complex nature of application characteristics, which may change during runtime, combined with the dynamic nature and unpredictability of the computational environment. Load balancing may be necessary in several parts of an application, and each part may require different scheduling algorithms. Furthermore, the load balancing characteristics of each part may change as the application progresses. A scheduling algorithm that is selected offline, which performs well early in the application's lifetime, may later become inappropriate. In this scenario, a dynamically changing set of scheduling algorithms is needed for the various parts of the application. The selection of scheduling algorithms for this dynamic set is a more difficult task. An intelligent entity is needed to select the best scheduling algorithm that achieves load balancing in each part of an application during runtime.

This paper describes a reinforcement learning (RL) system that was designed to dynamically select loop scheduling algorithms for parallel loops in time-stepping scientific applications. An application that utilizes the RL system is assumed to have a number of parallel loops that are executed via dynamic loop scheduling. The interface between the RL system and the application is very simple: the application supplies to the RL system a loop identifier, the scheduling method that was used with the loop, and the

0-7803-9043-1/05/$20.00 ©2005 IEEE

loop execution time; the RL system returns the scheduling method to be used by the application next time the identified loop is executed. The RL system, which is implemented in C, was initially tested with a Fortran 90 application that simulates wave packet dynamics, on a Linux cluster. Timing experiments indicate that, statistically, the performance of the application with the RL system is significantly better than the performance of the application using any of the loop scheduling algorithms. For example, on two or four processors, the parallel time of the application with learning is nearly half the parallel time of the application without learning.

An outline of the remainder of this paper is as follows. Section 2 reviews the loop scheduling approach to dynamic load balancing in a scientific application. A review of the general principles of RL and the specific learning techniques SARSA and QLEARN are also given in this section. Section 3 describes a RL system designed to be integrated into time-stepping scientific applications with parallel loops. The RL system monitors the execution times of the loops, and utilizes these times as inputs to the learning technique, in order the determine the scheduling algorithms for the loops in the succeeding time steps. The representative application that was employed to test the RL system is the simulation of wave packet dynamics using the quantum trajectory method. Section 4 presents the experimental setup to compare the performance of this application using the RL system and its performance without using the system. Results of the statistical analysis on the collected performance data are also discussed in this section. Section 5 gives a summary and lists future research directions.

2. Load Balancing and Reinforcement Learning

The main contribution of this paper is the application of reinforcement learning to select loop scheduling algorithms for improving performance via load balancing, of time-stepping scientific applications that contain computationally intensive parallel loops. This section provides a summary of the selection problem, and gives an overview of reinforcement learning, the proposed solution.

2.1. Load Balancing Via Dynamic Loop Scheduling

Many scientific applications are computationally intensive because they contain loops with large numbers of independent iterations. These loops are easily amenable to parallelization, however, the loop iteration execution times could vary due to conditional statements or variable amount of computation required by different iterations. Even in applications where there is no algorithmic variation in iteration lengths, iteration execution times may vary due to differences in effective processor speeds. The cumulative effect of variances in loop iteration execution times could ultimately lead to processor load imbalance and therefore to severe performance degradation of parallel applications.

For effectively load balancing scientific applications, algorithms for scheduling parallel loop iterations with variable running times have been extensively studied [17, 25, 34, 15, 21] and have resulted in the development and implementation of dynamic loop scheduling algorithms based on probabilistic analysis [15, 2, 14, 4, 3, 5, 6, 9, 10]. A fundamental trade-off in scheduling parallel loop iterations is that of balancing processor loads while minimizing scheduling overhead. Although a number of loop scheduling algorithms are available, selecting the algorithm that is appropriate for an application involves guesswork. Employing a sophisticated algorithm when a simpler algorithm is sufficient would entail higher overhead. The application may also have a number of parallel loops, each loop requiring a different scheduling algorithm for minimum completion time. Furthermore, if the application involves time-stepping, then the loop characteristics may change as the application progresses. A scheduling algorithm that is appropriate in earlier stages may not achieve the highest performance in later stages. Thus, runtime selection of the scheduling algorithm is justified.

2.2. Reinforcement Learning Algorithms

In recent years, intelligent systems have been developed and implemented in several real world problem domains such as game playing [19, 31, 32, 33] and robotics [27, 20, 22]. A more specific example is program selection [18]. These systems achieved their perceived *intelligence* through learning via the use of a computational technique called reinforcement learning.

Reinforcement Learning (RL) is an active area of research in Artificial Intelligence (AI), specifically in machine learning, because of its generality and widespread applicability in different environments. RL uses a goal-oriented approach for solving problems and by interacting with a complex and uncertain environment. It solves problems via learning, planning and decision-making. RL is to be distinguished from *Supervised Learning* (SL) which is another type of approach to machine learning. In SL, knowledge is gained from a supervisor or teacher that uses a sample of input-output pairs while the learner is instructed what actions to perform. In SL, learning is obtained offline, like the manner that supervised neural networks solve real-world problems such as speech recognition and pattern recognition. However, it is often impossible to obtain the input-output pairs for real-time applications because an exhaustive representation of all possible situations cannot be obtained. RL, on the other hand, involves an agent that learns

the behavior of a dynamic environment through trial and error. The agent is given an immediate *reward* for taking an action and observes the effect of that action on the *state* of the environment. The agent learns the optimal path that will lead to the goal by learning through the experience gained about the states, actions, and rewards in an uncertain environment. The learning process in RL does not require input-output pairs, and is done online such that the problem is concurrently solved while learning [30].

An RL system consists of an agent and its environment (Figure 1). The environment can be in any *state* s from a distinct set of states. The change in the state of the environment is driven by a transition function T, which is dependent on the action a performed by the agent and the current state of the environment. The agent selects an action a from a distinct and finite set of actions based on the *reward* r it received and its observation of the current state s of the environment. The reward is a scalar reinforcement signal from a set R, which communicates the changes in the environment to the learner and represents the degree to which a state is desirable or not. The reward is used to guide the learning process and to specify the wide range of planning goals. The reward maximizes the benefits achieved in the long run. At a specific time instance, the learner is guided in the selection of actions by a *policy* B. A policy maps the states of the environment and actions that are taken when the agent is in a particular state [16]. The general learning scenario in a RL system is as follows. The agent receives an input i from the set I when it is in a current state, and takes an action. It chooses an action from a state guided by a policy to produce an output. The action taken by the agent changes the state to produce a new state. The value of the transition that produced the change in a state is given as a reward. The reward reinforces the signal that a chosen action maximizes an agent's goal function, given a state of the environment. The agent learns the system behavior to maximize the reward obtained on a trial and error basis [26].

There are two approaches for learning in RL: model-based and model-free. The model-based learning approach uses a model M and the utility function U_M from M, while model-free learning approach uses an action-value function Q and does not require a model for the learning process. Examples of model-based learning are Dyna [28, 29], prioritized sweeping [23], Queue-Dyna [24], and Real-Time Dynamic Programming [7]. However, the model-based method of learning is not applicable for the automatic selection of algorithms problem because M is unknown. The model-free learning approaches were used in this endeavor instead. An example of a model-free approach is Temporal Difference (TD), which is based on the combination of Monte-Carlo (MC) and Dynamic Programming (DP). TD updates the estimates from the learned estimates like DP and learns without a model like MC. The TD class of learning algorithms include SARSA and QLEARN [35, 36]. SARSA learns the transitions from a state-action pair to another state-action pair, and finds the policy by using a greedy approach. It needs to know the next action that will be taken in order to update the value function. Q learning, on the other hand, uses a delayed reinforcement and chooses an action that maximizes the policy via a Q function. The Q function directly approximates the optimal value $Q*$ independent of the policy being followed.

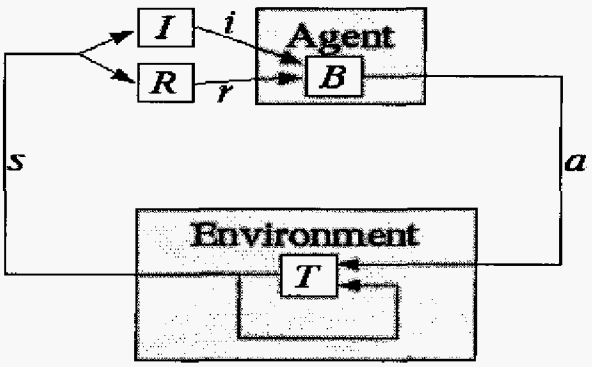

Figure 1. Components of a reinforcement learning system

3. RL in a Time-stepping Application

A time-stepping scientific application which requires dynamic load balancing provides an excellent environment for RL. Irregularities in the application and in the underlying computing system may evolve unpredictably, hence a load balancing algorithm that performs well in earlier time steps may prove to be inefficient in later time steps. However, a large number of time-steps provides ample opportunities to learn. A RL agent following the model-free learning approach becomes very useful for automatically selecting the appropriate load balancing algorithm during the lifetime of the application. This section describes a framework for the integration of RL into a class of scientific applications to improve their performance.

Figure 2 illustrates a high-level structure for the class of time-stepping applications with computationally intensive parallel loops. An application in this class evolves over N time steps. Within a single time step, L parallel loops possibly with different lengths and nonuniform iteration times are executed. Typically, there will be other statements between the loops, and the results of the computations in earlier loops will be needed by other loops. Since the loops are parallel, dynamic loop scheduling will be utilized for

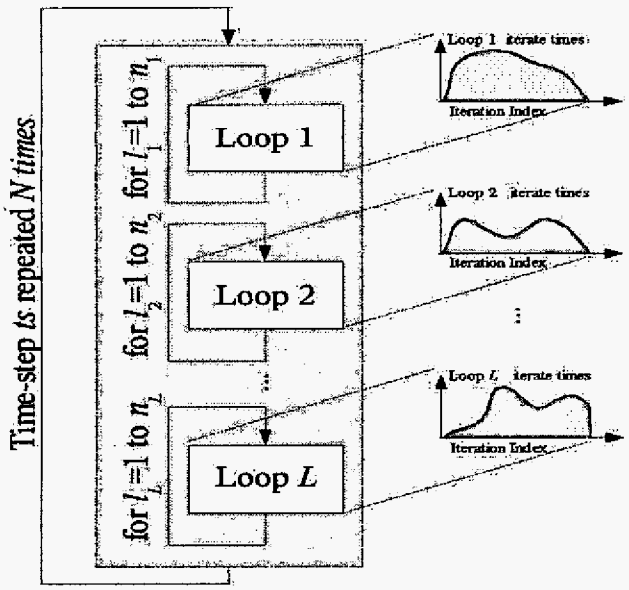

Figure 2. High-level structure of a time-stepping application with parallel loops

load balancing in order to minimize the loop completion times. General-purpose loop scheduling routines have already been developed and can be integrated into the application [8, 1].

A problem that arises when using dynamic scheduling to execute a parallel loop is the selection of the appropriate loop scheduling algorithm. A common practice is to undertake trial runs of the application, employing all the available scheduling algorithms, and choosing the algorithm which gives the highest performance for the production runs of the application. This may be feasible if the application has a single parallel loop whose characteristics do not significantly change with the time steps, and if the parallel execution environment is dedicated. The difficulty of the selection problem becomes complicated when there are several loops to schedule, each having unique load balancing characteristics that vary with the time steps. The dynamic nature of the parallel environment, such as unpredictable network latencies or operating system interference, further compounds the selection problem. These difficulties justify the need for an intelligent agent to select the best scheduling algorithm for a parallel loop during runtime.

Figure 3 illustrates the design of a proposed RL system for selecting an algorithm for scheduling a single parallel loop. The design is derived by adding the loop scheduling context to the environment of the generic RL system in Figure 1. The parallel loop, whose iterations may have irregular execution times, is to be invoked repeatedly in an application. The goal of the system is to minimize the total time spent by the application in the loop. This translates

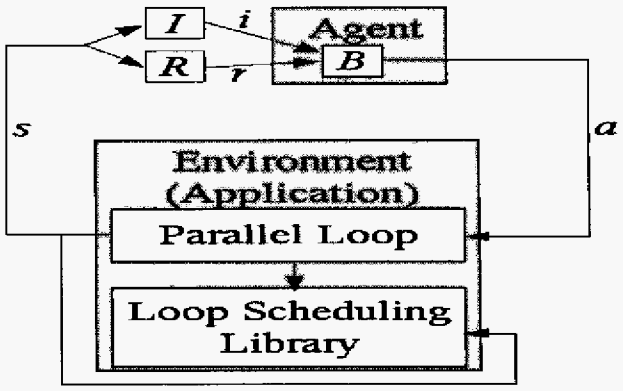

Figure 3. Design of a RL system for loop scheduling algorithm selection

to minimizing the completion time of each loop invocation, which is accomplished through dynamic load balancing via loop scheduling. During the first few invocations of the loop when the agent does not have information about the behavior of environment, the agent simply specifies each algorithm in the library in a round-robin fashion. After this initial learning period, the agent applies the adaptive learning policy B on the accumulated information to compute an action a, that is, to select a scheduling algorithm from a library. The environment transitions to another state s with the change in the scheduling algorithm to be used by the application. The loop completion time with the selected algorithm determines a performance level, which is the basis of the reward r for the action taken by the agent. The application communicates information i about the changed state and the reward r to the agent for continuous learning by the policy B. If the agent takes action only after a specified number of loop invocations, the application simply reuses the algorithm associated with the current state s, denoted by the loop back arrow from the environment to the library.

The components of the proposed system are independent: the application can execute the loop (possibly inefficiently) without the loop scheduling library; the library can be linked to any scientific application with a parallel loop; and the RL agent can be used by other applications. However, the integration of the components enables performance improvement in the application. The flexible design allows upgrades to be incorporated into any component without affecting the operation of the others, such as additional scheduling algorithms for the library, new learning policies into the agent, and more accurate formulas into the application.

Although the RL system proposed above is designed for a single parallel loop, the setup can be replicated within a single application with several parallel loops. Figure 4 il-

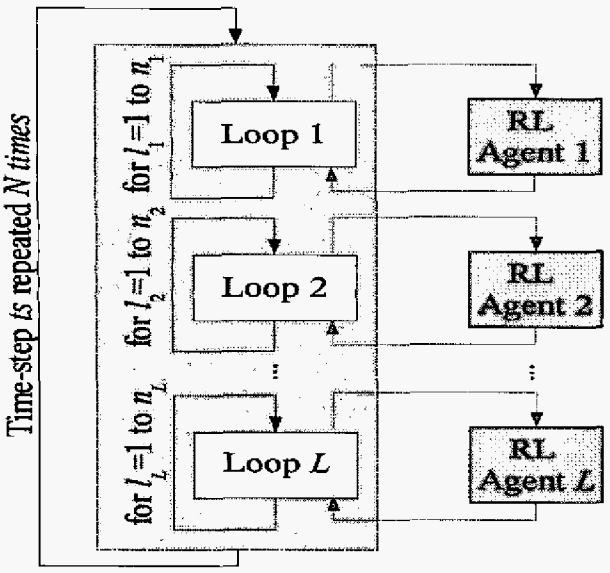

Figure 4. Integration of RL into time-stepping application with parallel loops

lustrates the integration of RL into the application class described at the beginning of this section.

4. Experimentation and Discussion

As a test case, the RL system described in the previous section was integrated into the simulation of wave packet dynamics using the quantum trajectory method (QTM) [11, 12]. Wave packet dynamics is used to model various phenomena in physics. Based on the hydrodynamic formulation of quantum mechanics, QTM represents the a wave packet by an unstructured set of pseudo particles whose trajectories are coupled by the quantum potential. The bulk of the computations for the time evolution of the wave packet occurs in the computationally intensive parallel loops for the quantum potential, the quantum force, and the divergence of velocity. An iteration in each loop invokes a moving least squares algorithm for the computation of certain derivatives. Each loop is executed using load balancing via dynamic loop scheduling because the iterations perform different amounts of computations. Furthermore, the amount of computations vary with the evolution of the wave packet. The wave packet simulation is implemented in Fortran 90 while the RL agent is implemented in C [13]. The interface between the RL agent and the simulation code required only two additional statements for each of the three parallel loops: before a loop starts, the agent is called to compute the index of the scheduling algorithm, and after the loop is finished, the agent is supplied with the algorithm index and the loop completion time.

The timing experiments were performed on a general purpose Linux cluster of 1038 Pentium III (1.0 and 1.266 GHz) processors with the Red Hat Linux operating system. Two processors reside in one node, 32 nodes are connected via fast Ethernet switch to comprise a rack, and the racks are connected via Gigabit Ethernet uplinks. On this cluster, the queuing system (PBS) attempts to assign homogeneous compute nodes to a job, but this is not guaranteed. Network traffic volume is also not predictable because the cluster is general purpose. Other jobs were running along with the simulations.

A free particle represented as a wave packet of 501 pseudo particles was simulated for 10,000 time steps. Simulations using other quantum mechanical models, larger wave packet sizes (1001 and 1501), more time steps (50,000), and other parallel environment were also undertaken. Results for these simulations and a more thorough analysis of the performance of the RL system will be reported in the future.

For this paper, an experiment was designed to test the hypothesis that, on a given number of processors, the application using load balancing methods selected by the RL agent during time-stepping (LEARN) will perform better than the application using a single load balancing method fixed before runtime (NOLEARN). LEARN has two levels that represent the learning methods used: SARSA and QLEARN. NOLEARN has eight levels representing the loop scheduling algorithms that are contained in the load balancing library: STATIC, FSC, GSS, FAC, AF, AWF, MODF, and EXPT. LEARN and NOLEARN are groups of load balancing techniques, m, which is a factor hypothesized to affect the performance of the application. The application was run under different number of processors, p: 2, 4, 8, 12, 16, 20, and 24 processors. Thus, the experiment is a two-factor factorial experiment with m as the first factor and p as the second factor. The numerical characteristics of the QTM preclude wave packet sizes beyond a few thousand pseudo particles; hence, the wave packet size cannot be arbitrarily increased to generate "larger" problems for execution on "more" processors.

To model the variability of the performance measurements induced by the underlying computing architecture, the application was run with five replicates. The parallel execution time T_p for each $m \times p$ factor combination was measured and averaged across the five replicates. The average T_p are graphed in Figure 5 with the mean values annotated with letters. The letters represent the statistical groupings from the Least-Squares Means (LSMEANS) method. Mean values with the same letter belong to one statistical group. The differences of the mean values in the same statistical group are not significantly different from zero according to the t statistics at $\alpha = 0.05$. Figure 5 indicates that:

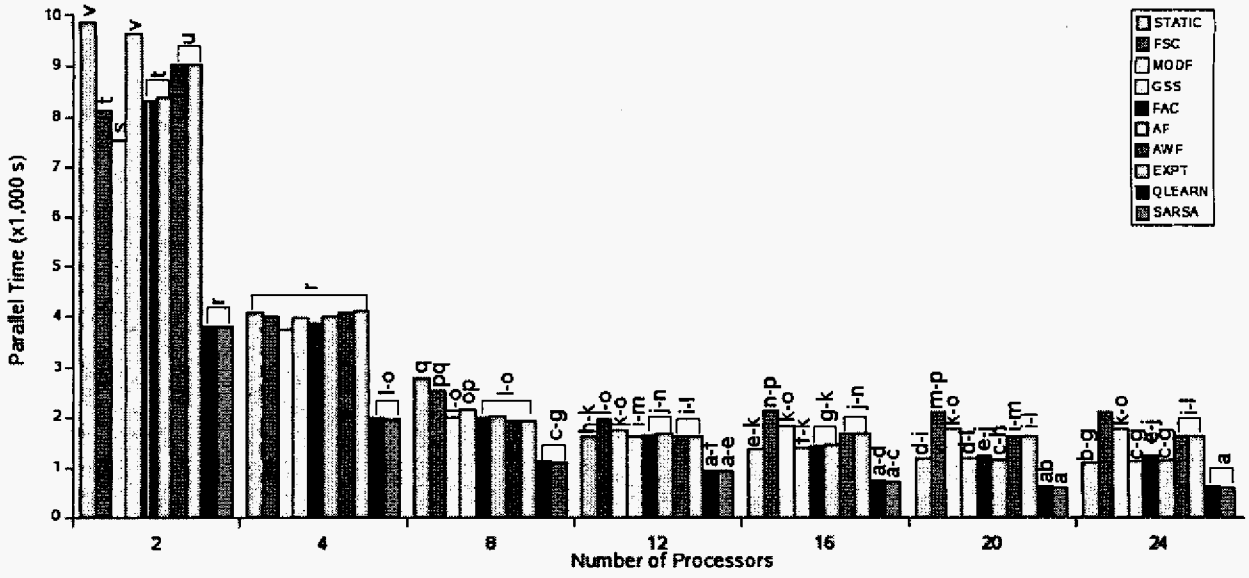

Figure 5. Mean parallel time (T_p) for wave packet simulation with 501 pseudo particles using different load balancing methods and RL techniques at increasing number of processors. Means with the same letter are not significantly different at 0.05 significance level via t statistics using LSMEANS

- For each p, the T_p of the application with either RL technique (LEARN) is significantly lower than the T_p of the application without learning (NOLEARN).
- For each p, there is no significant difference between the T_p of using SARSA or QLEARN.
- For LEARN, there is a significant drop in the T_p when p is increased from $p = 2$ to $p = 8$. However, the T_p does not significantly change as p is further increased from $p = 12$ to $p = 24$. This means that with RL, the optimum p for the application with 501 pseudo particles is at most $p = 12$.
- For NOLEARN, STATIC is worst from $p = 2$ to $p = 8$, but better than most other NOLEARN techniques for $p = 16$ to $p = 24$. The reason for this is that with a fixed problem size, the performance of a dynamic scheduling method degrades with additional processors, due to the scheduling overhead increase. STATIC is not penalized with higher scheduling overhead like the dynamic techniques when using more processors.
- The T_p for LEARN at $p = 2$ is not significantly different from the T_p for NOLEARN at $p = 4$. Similarly, the T_p for LEARN at $p = 4$ is statistically comparable to the T_p for NOLEARN at $p = 8$ with the exception of STATIC and FSC. This means that for $p \leq 8$ that were tested, the application with RL using p processors has statistically the same T_p as the application without RL using the next higher p. The T_p for LEARN at $p = 12$ is even significantly better than the T_p for NOLEARN at $p = 16$.

5. Summary and Concluding Remarks

This paper presents the design of a reinforcement learning (RL) agent that automatically selects a load balancing algorithm from a library. The RL agent was designed for the class of the large-scale time-stepping applications that have one or more computationally intensive parallel loops with nonuniform iteration execution times. These execution times may also vary with respect to the evolution of time-step. Each parallel loop is assigned an RL agent which dynamically chooses from a library a loop scheduling algorithm to minimize the loop completion time. Investigations on the performance of running a representative scientific application - simulation of wave packet dynamics, with and without the RL agent, were performed. Comparisons of the performance, in terms of parallel time T_p, of the simulation with and without the RL agent, were undertaken by pairwise comparison of means via the LSMEANS method using the t-statistic. The analysis of the results indicate that at any number of processors p the simulation performs statistically better with the RL agent than without the agent. There is no significant difference in the performance of the simulation at any number of processors when using either QLEARN or

SARSA for the RL technique. The analysis also identifies the optimal number of processors p^* for the wave packet size tested. For p smaller than this optimal p^*, the performance of the simulation with the RL agent is comparable to or even better than the performance of the simulation with a higher number of processors without learning.

These preliminary results validate the suitability of RL as a viable procedure for runtime selection of dynamic loop scheduling algorithms from a library to improve the performance of a class of large, time-stepping scientific applications with computationally intensive parallel loops. Work is under way to integrate the RL agent as an extra functionality of the loop scheduling library, so that time-stepping applications using the library to execute parallel loops automatically incorporate learning for improved performance. Work is also planned to utilize the RL agent in other time-stepping applications that involve a choice of algorithms based on a higher number of state variables.

Acknowledgments

The authors would like to thank the National Science Foundation for its support of this work through the following grants: CAREER #9984465, ITR/ACS #0085969, ITR/ACS #0081303, #0132618, and #0082979.

J.P. Pabico is on leave from the Institute of Computer Science, University of the Philippines Los Baños, College 4031, Laguna, Philippines.

References

[1] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. P. Pabico, and R. L. Cariño. A novel dynamic load balancing library for cluster computing. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, in association with the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, pages 346–352. IEEE Computer Society Press, 2004.

[2] I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1995.

[3] I. Banicescu and Z. Liu. Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes. In *Proceedings of the High Performance Computing Symposium (HPC 2000)*, pages 122–129, 2000.

[4] I. Banicescu and R. Lu. Experiences with fractiling in n-body simulations. In *Proceedings of High Performance Computing 98 Symposium*, pages 121–126, 1998.

[5] I. Banicescu and V. Velusamy. Load balancing highly irregular computations with the adaptive factoring. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002) - Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2002.

[6] I. Banicescu, V. Velusamy, and J. Devaprasad. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 6(3):215–226, 2003.

[7] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[8] R. L. Cariño and I. Banicescu. A load balancing tool for distributed parallel loops. In *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments*, pages 39–46. IEEE Computer Society Press, 2003.

[9] R. L. Cariño and I. Banicescu. Dynamic scheduling parallel loops with variable iterate execution times. In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium - 3rd Workshop on Parallel and Distributed Scientific and Engineering Computing With Applications (IPDPS-PDSECA 2002) CDROM*. IEEE Computer Society Press, 2002.

[10] R. L. Cariño and I. Banicescu. Load balancing parallel loops on message-passing systems. In S. Akl and T. Gonzales, editors, *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 362–367. ACTA Press, 2002.

[11] R. L. Cariño, I. Banicescu, R. K. Vadapalli, C. A. Weatherford, and J. Zhu. Parallel adaptive quantum trajectory method for wavepacket simulation. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium - 4th Workshop on Parallel and Distributed Scientific and Engineering Applications (CDROM)*. IEEE Computer Society Press, 2003.

[12] R. L. Cariño, I. Banicescu, R. K. Vadapalli, C. A. Weatherford, and J. Zhu. *Message Passing Parallel Adaptive Quantum Trajectory Method*, pages 127–139. Kluwer Academic publishers, 2004.

[13] S. Dhandayuthapani. Automatic selection of dynamic loop scheduling algorithms for load balancing using reinforcement learning. Master's thesis, Mississippi State University, 2004.

[14] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 318–328, 1996.

[15] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, Aug. 1992.

[16] L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[17] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, Oct. 1985.

[18] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*

(ICML 2000), Stanford University, Standord, CA, USA, June 29 - July 2, 2000, pages 511–518. Morgan Kaufmann, 2000.

[19] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 157–163. Morgan Kaufmann, 1994.

[20] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, 1991.

[21] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, Apr. 1994.

[22] M. J. Mataric. Reward functions for accelerated learning. In W. Cohen and H. Hirsh, editors, *Proceedings of the 11th International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[23] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[24] J. Peng and R. J. Williams. Efficient learning and planning with the Dyna framework. *Adaptive Behaviour*, 1(4):437–454, 1993.

[25] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, Dec. 1987.

[26] S. I. Reynolds. *Reinforcement Learning with Exploration*. PhD thesis, The University of Birmingham, Birmingham, United Kingdom, 2002.

[27] S. Schaal and C. G. Atkeson. Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, 14(1):57–71, 1994.

[28] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning*. Morgan Kauffmann, 1990.

[29] R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 353–357. Morgan Kauffmann, 1991.

[30] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

[31] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3–4):257–277, 1992.

[32] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[33] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[34] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Transactions on Parallel Distributed Systems*, 4(1):87–98, Jan. 1993.

[35] C. J. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.

[36] C. J. Watkins and P. Dyan. Q-Learning. *Machine Learning*, 8(3–4):279–292, 1992.