Mississippi State University
Scholars Junction

Theses and Dissertations Theses and Dissertations

8-7-2004

Automatic Selection of Dynamic Loop Scheduling Algorithms for Load Balancing using Reinforcement Learning

Sumithra Dhandayuthapani

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
Dhandayuthapani, Sumithra, "Automatic Selection of Dynamic Loop Scheduling Algorithms for Load Balancing using Reinforcement Learning" (2004). *Theses and Dissertations*. 829.
https://scholarsjunction.msstate.edu/td/829

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

AUTOMATIC SELECTION OF DYNAMIC LOOP SCHEDULING ALGORITHMS FOR LOAD BALANCING USING REINFORCEMENT LEARNING

By

Sumithra Dhandayuthapani

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

August 2004

Copyright by

Sumithra Dhandayuthapani

2004

AUTOMATIC SELECTION OF DYNAMIC LOOP SCHEDULING ALGORITHMS

FOR LOAD BALANCING USING REINFORCEMENT LEARNING

By

Sumithra Dhandayuthapani

Approved:

Ioana Banicescu Associate Professor of Computer Science and Engineering (Major Professor)	Thomas Philip Professor of Computer Science and Engineering (Committee Member)
Eric Hansen Associate Professor of Computer Science and Engineering (Committee Member)	Edward B. Allen Assistant Professor of Computer Science and Engineering (Graduate Coordinator)

A. Wayne Bennett
Dean of the Bagley College of Engineering

Name: Sumithra Dhandayuthapani

Date of Degree: August 7, 2004

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Ioana Banicescu

Title of Study: AUTOMATIC SELECTION OF DYNAMIC LOOP SCHEDULING ALGORITHMS FOR LOAD BALANCING USING REINFORCEMENT LEARNING

Pages in Study: 76

Candidate for Degree of Master of Science

Scientific applications are large, complex, irregular, computationally intensive and typically contain data parallel loops. The presence of loops with independent iterations, makes parallel computing as a natural choice for solving these applications. The computational requirements of these iterations vary due to variations in problem, algorithmic and systemic characteristics, leading to performance degradation during parallel execution. Considerable amount of research has been dedicated to the development of dynamic scheduling techniques based on probabilistic analysis to address these predictable and unpredictable factors that lead to severe load imbalance. The mathematical foundations of these scheduling algorithms have been previously developed and published in the literature. These techniques have successfully been integrated into scientific applications as well as into runtime systems. Recently, efforts have also been directed to integrate these

techniques into dynamic load balancing libraries for scientific applications. The optimal scheduling algorithm to load balance a specific scientific application in a dynamic computing environment is very difficult to determine without an exhaustive testing of all the scheduling techniques. This is a time consuming process, and therefore, there is a need for developing an automatic mechanism for the selection of dynamic scheduling algorithms. In recent years, extensive work has been dedicated to the development of reinforcement learning and some of its techniques have addressed load balancing problems. However, the techniques do not cover a number of aspects that affect the performance of scientific applications. First, these previously developed techniques address the load balancing problem only at a coarse granularity level (for example, job scheduling), and the reinforcement learning techniques used for load balancing are based on learning from trained datasets which are obtained prior to the execution of the application. Moreover, scientific applications contain parameters whose variations are so irregular that the use of training sets would not be able to accurately capture the entire spectrum of possible characteristics. Finally, algorithm selection using reinforcement learning has only been used for simple sequential problems. This thesis addresses these limitations and provides a novel integrated approach for automating the selection of dynamic scheduling algorithms at a finer granularity level to improve the performance of scientific applications using reinforcement learning.

This integrated approach is experimentally tested on a scientific application that involves a large number of time steps - the Quantum Trajectory Method (QTM). A quali-

tative and quantitative analysis of the effectiveness of this novel approach is presented to underscore the significance of its use in improving the performance of large scale scientific applications.

DEDICATION

To Vinodhini, Sasi Rekha, Saraswathi and Dhandayuthapani.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major professor Dr. Ioana Banicescu, for giving me the opportunity to work on this exciting area of research. She was a constant force of motivation and encouragement and gave the vision and direction to complete this thesis. I would like to thank my committee members Dr. Eric Hansen and Dr. Thomas Philip. I would like to thank Dr. Ricolindo Carino for his valuable suggestions and technical advice. I would also like to thank the Engineering Research Center for providing the infrastructure and facilities for doing this research. I would like to acknowledge the National Science Foundation for its support for my Master Thesis through the following grants: NSF #0082979, NSF #0132618 and NSF #9984465.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE xii

CHAPTER

I. INTRODUCTION 1

1.1 Motivation 2
1.2 Research Problem Statement (Hypothesis) 3
1.3 Contributions 3
1.4 Organization 4

II. BACKGROUND AND RELATED WORK 5

2.1 Scheduling and Load Balancing 5
2.2 Load Balancing Algorithms 6
2.2.1 Non-adaptive Scheduling Algorithms 7
2.2.2 Adaptive Scheduling Algorithms 8
2.2.3 Automatic Selection of Scheduling Algorithm 9
2.3 Machine Learning 10
2.4 Machine Learning Basics 10
2.5 Supervised Learning 11
2.5.1 Inductive Learning 11
2.5.2 Learning Through Networks 12
2.6 Reinforcement Learning 13
2.7 Components of Reinforcement Learning 14
2.8 General Concepts of Reinforcement Learning 16

CHAPTER Page

2.8.1 Markov Decision Process 16
2.8.2 Value and Policy 16
2.9 Learning Optimal Policy 18
2.9.1 Model Free Methods 18
2.9.2 Model Based Methods 19
2.10 Summary 20

III. DESIGN AND IMPLEMENTATION 21

3.1 Overview 21
3.1.1 Design Goals 23
3.2 Design Layers 24
3.2.1 Application Layer 25
3.2.2 Scheduling and Load Balancing Layer 25
3.2.3 Reinforcement Learning Layer 27
3.2.3.1 Learning Algorithms 30
3.3 Case Study 32
3.3.1 Problem Mapping 34
3.3.2 Algorithm Selection Mechanism 34
3.3.3 State - Action Transition 35
3.4 Implementation Details 36
3.5 Summary 37

IV. EXPERIMENTAL RESULTS AND ANALYSIS 38

4.1 Experimental Setup 38
4.2 Overview of Experiments 39
4.3 Performance Analysis 41
4.3.1 Performance Metrics 41
4.3.1.1 Parallel Execution Time 41
4.3.1.2 Cost 41
4.3.2 Cost Analysis with and without Learning 42
4.3.3 Behavior of Q Learning and SARSA Learning for the QTM . . . 47
4.3.4 Scheduling Methods Selected for the Computational Loops . . . 47
4.3.5 Effect of Varying the Learning Rate on the Performance of QTM 54
4.3.6 Effect of Varying the Discount Rate on the Performance of QTM 60
4.3.7 Effect of Varying Episodes and Number of Processors 64
4.4 Summary 68

CHAPTER Page

V. CONCLUSIONS AND FUTURE WORK 72

5.1 Future Work 73

REFERENCES 75

LIST OF TABLES

TABLE Page

2.1 Value Iteration 17

2.2 Policy Iteration 18

3.1 Scheduling Strategy 28

3.2 SARSA Learning 29

3.3 Q Learning 30

3.4 Wave-packet Simulation algorithm using QTM 31

4.1 Linux: Scheduling methods selected for 2 processors and varying episodes . 69

4.2 Linux: Scheduling methods selected for 4 processors and varying episodes . . 69

4.3 Linux: Scheduling methods selected for 8 processors and varying episodes . . 69

4.4 Solaris: Scheduling methods selected for 2 processors and varying episodes . 70

4.5 Solaris: Scheduling methods selected for 4 processors and varying episodes . 70

4.6 Solaris: Scheduling methods selected for 8 processors and varying episodes . 70

LIST OF FIGURES

FIGURE		Page
2.1	Actor Environment Interaction	14
3.1	Generalized Time-Stepping Scientific Application with Parallel Loops	22
3.2	General Design	23
3.3	Overview	24
3.4	Replicated work queue	27
3.5	Wave packet Simulation of free particles using Quantum Trajectory Method	33
3.6	Interaction between learner, scheduler and the runtime environment	36
3.7	State-Action Transition	37
4.1	Experimental Setup	40
4.2	Linux : Cost comparison of Learning vs No Learning for 10000 episodes	43
4.3	Linux : Cost comparison of Learning vs No Learning for 5000 episodes	44
4.4	Linux : Cost comparison of Learning vs No Learning for 500 episodes	44
4.5	Linux : Cost comparison of Learning vs No Learning for 200 episodes	45
4.6	Ultra : Cost comparison of Learning vs No Learning for 200 episodes	45
4.7	Ultra : Cost comparison of Learning vs No Learning for 2000 episodes	46
4.8	Ultra : Cost comparison of Learning vs No Learning for 5000 episodes	46
4.9	Ultra : Cost comparison of Learning vs No Learning for 10000 episodes	47

FIGURE Page

4.10 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001 49

4.11 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001 50

4.12 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001 50

4.13 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001 51

4.14 Linux: Algorithms selected by Q Learning - Wave-Packet Size :501 51

4.15 Linux:Algorithms selected by Q Learning - Wave-Packet Size :501 52

4.16 Linux: Algorithms selected by Q Learning - Wave-Packet Size :501 52

4.17 Linux:Algorithms selected by Q Learning - Wave-Packet Size :1501 53

4.18 Linux: Algorithms selected by Q Learning - Wave-Packet Size :1501 53

4.19 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1501 54

FIGURE Page

4.20 Solaris:Algorithms selected by SARSA Learning - Wave-Packet Size :501 . . 55

4.21 Solaris: Algorithms selected by SARSA Learning - Wave-Packet Size :501 . . 55

4.22 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size : 501 . 56

4.23 Solaris:Algorithms selected by SARSA Learning - Wave-Packet Size :1001 . 56

4.24 Solaris: Algorithms selected by SARSA Learning - Wave-Packet Size :1001 . 57

4.25 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size : 1001 . 57

4.26 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size : 1001 . 58

4.27 Solaris:Algorithms selected by SARSA Learning - Wave-Packet Size :1501 . 58

4.28 Solaris: Algorithms selected by SARSA Learning - Wave-Packet Size :1501 . 59

4.29 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size :1501 . 59

4.30 Solaris: Variation of cost with learning rate for 500 episodes 60

4.31 Solaris: Variation of cost with learning rate for 1000 episodes 61

4.32 Solaris: Variation of cost with learning rate for 5000 episodes 61

4.33 Solaris: Variation of cost with learning rate for 10000 episodes 62

4.34 Linux: Variation of cost with learning rate for 500 episodes 62

4.35 Linux: Variation of cost with learning rate for 5000 episodes 63

4.36 Linux: Variation of cost with learning rate for 10000 episodes 63

4.37 Solaris: Variation of cost with discount rate for 500 episodes 65

4.38 Solaris: Variation of cost with learning rate for 1000 episodes 65

4.39 Solaris: Variation of cost withdiscount rate for 5000 episodes 66

4.40 Solaris: Variation of cost withdiscount rate for 10000 episodes 66

4.41 Linux: Variation of cost with discount rate for 500 episodes 67

4.42 Linux :Variation of cost with discount rate for 5000 episodes 67

4.43 Linux :Variation of cost with discount rate for 10000 episodes 68

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

RL Reinforcement Learning

MDP Markov Decision Process

QTM Quantum Trajectory Method

RTDP Real Time Dynamic Programming

TD Temporal Difference

DP Dynamic Programming

MC Monte Carlo

GSS Guided Self Scheduling

WF Weighted Factoring

AWF Adaptive Weighted Factoring

AF Adaptive Factoring

CHAPTER I

INTRODUCTION

The presence of loops with independent iterations is a rich source of parallelism in scientific applications. To obtain high performance solutions and to take advantage of the parallelism, these applications are executed on multiple processors. During the parallel execution, various factors can lead to performance degradation, such as the inter-processor communication, the load imbalance among the processors, the overhead incurred during communication, and the synchronization process during the computations. Among these factors, load imbalance is the predominant factor that affects the performance of the application. Load imbalance is caused mainly due to the variances in problem characteristics, algorithmic characteristics, and systemic characteristics. The imbalance in the problem characteristics is due to the non-uniform distribution of data, while the imbalance in the algorithmic characteristics is due to different conditional execution paths and boundary phenomena. Systemic characteristics cause unpredictable data access time, cache misses and interrupts adding to load imbalance. To address load imbalance problems, various scheduling algorithms have been developed. The earlier techniques mainly address the problem characteristics and to a certain extent they address predictable systemic charac-

teristics. The scheduling algorithms proposed in recent years have addressed all the factors and have accounted for complex factors that lead to load imbalance.

1.1 Motivation

Currently, it is difficult to find an optimal scheduling algorithm to achieve load balancing for a specific scientific application which is executed in an unpredictable environment. This is due to the complex nature of the application which changes during runtime and due to the dynamic nature and unpredictability of the computational environment. Thus, an intelligent system is needed to find the best scheduling algorithm to achieve load balancing during runtime. In the recent years, extensive research work has been dedicated in artificial intelligence to integrate machine learning strategies for load balancing problems. Although machine learning techniques have been used from artificial intelligence area for load balancing problems, these techniques have addressed problems at a coarser granularity and not at finer one within an application. Moreover, the problems addressed require a model with input/output pair which gives all possible environmental conditions. This is very difficult to achieve in practice when we deal with dynamic systems. Automatic algorithm selection techniques have been developed in recent years, but they have been tested sequentially and only on simple algorithms (i.e., sorting algorihtms). They have not yet been tested on parallel systems. To overcome these problems and to make best use of the existing state of the art techniques in both scheduling and artificial intelligence, this thesis proposes an integrated approach for improving the performance of scientific applications.

1.2 Research Problem Statement (Hypothesis)

The research problem is stated as follows:

1. The implementation of a generic framework for the automatic selection of the best suited scheduling algorithm for load balancing scientific applications in parallel distributed environments using reinforcement learning is possible.
2. The performance of a scientific application running in distributed environments using this integrated framework will be improved with respect to the conventional approach of manually specifying a scheduling algorithm for the application.

1.3 Contributions

The main contribution of this thesis is the application of reinforcement learning techniques to automatically determine the optimal scheduling algorithms for parallel loops within a time-stepping scientific application, without prior knowledge of the computational requirements of the parallel loops and the runtime environment. To the best of our knowledge, this is the first time such an integrated design is being developed.

The specific contributions of this thesis are as follows:

1. The integrated design for automatic selection of scheduling algorithms for parallel scientific application has been designed and successfully implemented for distributed memory architectures.
2. Empirical proof shows that the strategy works and achieves cost reduction and efficient utilization of the computational resources when compared to the conventional approach by fixing a scheduling algorithm for an application.
3. The strategy can be used in several places within an application.
4. The implementation is portable across different operating systems and scalable for increasing problem size with the increase in the number of processors.
5. The strategy is flexible for incorporating new scheduling algorithms and reinforcement learning techniques thereby accommodating software development.

6. A qualitative and quantitative analysis of the experimental results for different operating systems, problem sizes, number of processors, learning algorithms and learning parameters are discussed to justify and validate the design and development of this integrated design approach.

1.4 Organization

The thesis is organized as follows: A review of state-of-the-art techniques in load balancing and reinforcement learning is presented in Chapter 2. The design of the automatic selection framework and the implementation details are discussed in Chapter 3. The experimental results with a detailed performance analysis from both qualitative and quantitative perspectives are presented in Chapter 4, to validate the hypothesis. The conclusion and future work are presented in Chapter 5.

CHAPTER II

BACKGROUND AND RELATED WORK

This chapter gives an overview of the techniques from the perspective of both scheduling algorithms for load balancing and machine learning.

2.1 Scheduling and Load Balancing

The problem of load imbalance in scientific applications on distributed systems has been addressed by scheduling algorithms. Extensive research has been done in developing these algorithms and integrating them with runtime systems and implementing them in applications. There has been tremendous advancement in the development of scheduling algorithms address load imbalance problems with increasing levels of complexity over the years. The scheduling algorithms include static scheduling, self scheduling (SS), Guided Self Scheduling (GSS), Fixed Size Chunking (FSC), Factoring, Weighted Factoring (WF), Fractiling, Adaptive Weighted Factoring (AWF) and Adaptive Factoring (AF) [10, 12, 15, 6, 2, 9, 5, 3, 4].

The static scheduling models the execution times of the loop iterates as being equal, which is very unrealistic for scientific applications that execute in dynamic environment. The static techniques do not address load imbalance during runtime and they have theo-

retical constraints imposed on them. In contrast, dynamic scheduling algorithms make a realistic assumption of the execution time of the loop iterated as independent random variables with variability among them. The recent advances in research on scheduling parallel loop iterates concentrate on dynamic load balancing. The new dynamic load balancing strategies relax the theoretical constraints and are based on probabilistic analysis. These techniques schedule iterates in chunks with decreasing sizes. The chunk sizes are dynamically chosen at runtime such that they have a high probability of completing before the optimal time. In addition, the techniques accommodate changes present in the heterogeneous system and handle most of the performance degradation factors caused by predictable phenomena such as irregular data and unpredictable phenomena like data access latency and operating system interference. The scheduling techniques have been incorporated into load balancing libraries and they were extensively tested on wide variety of applications in computational fluid dynamics, quantum physics, automatic quadrature routines, N-body simulations in both distributed memory and shared memory environments [7, 1].

2.2 Load Balancing Algorithms

Several scheduling algorithms have been proposed and incorporated into scientific applications that address a wide range of load imbalance problems from the simplest to the most complex level. With a given problem size N, and number of processors P, the scheduling algorithms determine the workload to be computed by each processor as number of iterations or chunks. The load balancing algorithms are broadly classified as non-adaptive

and adaptive algorithms. The non-adaptive algorithms determine the chunk size based on the assumptions or informations that are available before the execution of the loops. The adaptive algorithms determine the information of the workloads during the execution or from the previous execution of the loop iterates.

2.2.1 Non-adaptive Scheduling Algorithms

The non-adaptive algorithms include static scheduling (STATIC), self scheduling (SS), fixed size chunking (FSC), guided self scheduling (GSS), factoring (FAC) and Weighted Factoring (WF). In STATIC, the chunks are of equal size (N/P). This technique achieves good load balancing when the iterates have constant execution time on homogeneous processors are with uniform load distribution. SS consist of unit chunk size and it is suitable only when the communication overhead is negligible, because of the large message transfers. FSC computes the optimal chunk size based on the constant overhead, mean and standard deviation of the iterate execution times which are assumed to be known prior to the loop execution. This technique performs well when the loops are executed on homogeneous processors with equal loads. GSS addresses the problem of uneven starting time of the processors and is applicable to constant length and variable length iterate executions. The chunks are allocated in decreasing size, where larger chunks are allocated initially, so that the smaller chunks can smoothen the unevenness of the larger initial chunks. GSS computes the chunk size as (R/P) where R is the number of remaining unscheduled iterates. In FAC, the iterates are scheduled in batches, where the size of a batch is a fixed ratio

of the unscheduled iterates and the batch is divided into P chunks. The ratio is dependent on the mean and standard deviation of the iterate execution times and determines the chunk sizes such that the resulting chunks have a high probability of finishing before the optimal time. In WF, the variation in processor speed is taken into consideration for determining the chunk size. The FAC batch of iterates are allocated to processors in proportion to their relative speeds.

2.2.2 Adaptive Scheduling Algorithms

The adaptive scheduling algorithms determine the chunk size dynamically during the run-time. Adaptive weighted factoring (AWF) evolved from factoring and weighted factoring and is applicable to time-stepping applications. Here, the weights are based on the rate of execution of iterates by processors (in contrast to the relative speeds in WF). The AWF technique uses equal processor weights in the initial computation and adapts the weight after every time step. The execution time of iterates done by a processor are recorded during a time-step, and the data from the processors are used to compute the weights for the next time-step [5]. A variation of AWF is AWF-B where the weights of the processors are computed using statistics from the earlier chunks within a time-step instead of the chunks from the previous time-step. This modification thus enables the application of AWF to non time-stepping applications. Another variation of AWF, which is AWF-C determines the chunk sizes based on the processor weights and the number of remaining iterates not using batches. Adaptive Factoring (AF) dynamically estimates the mean and standard deviation

of the iterate execution times during runtime, and thus relaxes the limitation imposed by factoring. First estimates are obtained from the small sized chunks of the initial batch. The chunk size of the succeeding chunks are computed based on the mean and standard deviation of the iterate execution times of the most recent chunk executed by each processor [4].

2.2.3 Automatic Selection of Scheduling Algorithm

The adaptive and non-adaptive algorithms have been incorporated into scientific applications and their performance have been analyzed by the use of load balancing tools [7, 1]. However, it is difficult to find the optimal scheduling algorithm dynamically for time-stepping scientific applications with large number of time-steps in an unpredictable environment, given several scheduling algorithms. This limitation raises a compelling need for designing an automatic agent which can use machine learning techniques to determine the optimal scheduling algorithm, dynamically by interacting with the environment and the scheduling algorithms.

The following sections give a brief discussion of the machine learning techniques and sets the stage for integrating scheduling and machine learning techniques to improve the performance of scientific applications.

2.3 Machine Learning

This section gives an introduction about machine learning and the various techniques in reinforcement learning which can be used for solving load balancing problems in scientific applications on distributed systems. Extensive research has been done in the machine learning area, and many of its techniques are being used for solving dynamic problems.

2.4 Machine Learning Basics

Machine learning deals with the study of algorithms that evolve with learning which produces intelligent programs through experience. The basic concept is the interaction between an intelligent system called agent and the environment in which the agent operates. An agent is associated with certain level of intelligence to choose a specific action in a defined environment. When the agent has to interact with an incomplete knowledge of the environment, the agent is able to choose the best action, by learning and obtaining knowledge from the environment in which it is placed, only through experience. This forms a basis for automation and forms the area of research called machine learning in artificial intelligence. Machine learning can be generally classified into "supervised learning", "unsupervised learning" and "reinforcement learning". In $supervised learning$ techniques, the learning agent is told what to do while in the case of $unsupervised learning$ for instance, the agent may learn to reduce the problem size but may not learn the correct outputs. The agent helps in finding the relationships and extracting regularities from the learning process, rather than finding the correct solution. In between these two extremes

is $reinforcementlearning$, where, the agent receives a feed back from performing an action, which guides it in finding the correct solution (like in the unsupervised learning) but it is not told what to perform next (like in the supervised learning).

2.5 Supervised Learning

Supervised learning can further be classified into inductive learning and learning from inductive and belief networks. However, these latter learning methods require a supervisor with certain previous associated knowledge for deciding the actions to take next. This basically involves a trained model and the agent learns from this built model.

2.5.1 Inductive Learning

In this method, learning is through induction from the examples obtained from the input-output pairs. There are two components of the learning agent in this model: the performance element, and the learning element. The performance element decides the action that needs to be taken, and the learning element makes improvements to the performance element. The categories under this learning techniques are: learning through decision trees (which are widely used in deterministic functions), the hypothesis and the version space hypothesis (which are based on logical theories), and computational learning theory (which analyzes the sample complexity and computational complexity of inductive learning) [17].

2.5.2 Learning Through Networks

The neural network consists of nodes connected through links. The link between two nodes is associated with a weight. The external connection to the environment is obtained through the input and the output nodes. The behavior of the environment is controlled by the adjustment of weights in the network with a given input to the system. The parallel distributed structure, and the ability of neural networks to learn and generalize, is used for solving large and complex unsolvable problems . However, in practice neural networks cannot provide the solution by working individually [8]. Back propagation and Bayesian learning are the important categories under this technique. The back propagation method consists of inputs with multi-layers of neurons. When the inputs are sent to the back propagation network, the neurons compute the weight and passes it to the output layer through one or more input layers. In the learning process, the error is propagated backward to the input layers. This method has widely been used in many applications and it converges locally to an optimal solution. Bayesian learning is used to learn representations of probabilistic functions, especially in the belief networks. This method views the problem of constructing hypotheses from data as a subproblem of the more fundamental problem, that of making decisions. Initially, the probability of each hypothesis is estimated, given the data. Predictions are made from the hypotheses, using the posterior probabilities of the hypotheses to weight the predictions [17].

2.6 Reinforcement Learning

Reinforcement Learning (RL) is an active area of research in Artificial Intelligence (AI) because of its generality and widespread applicability in both accessible and inaccessible environments. This area of machine learning concentrates on a goal-oriented approach for solving learning problems by interacting with the complex and uncertain environments. It covers the core issues in AI like learning, planning and decision-making. Reinforcement learning involves an agent, which learns the behavior of a dynamic environment through trial and error. In supervised learning, a learning technique used in neural networks, speech recognition and pattern recognition, the knowledge used is gained from a supervisor. In this learning system, we require a sample of the input-output pairs for the learning problem and the learner is instructed what actions to perform. It is often impossible to obtain the input-output pairs for real-time applications and an exhaustive representation of every possible situation cannot be obtained. In the case of reinforcement learning, the agent is given an immediate "reward" and a "state" for taking an action. 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, and does not require input-output pairs. Another important difference between supervised learning and RL is that the system is evaluated concurrently with learning. The trial and error learning mechanism and the concept of reward makes the reinforcement learning distinct from other learning techniques. A challenging problem and a key aspect in RL is the tradeoff between exploration and exploitation. To exploit is to use the best experienced action, and to explore, the agent

has to try new actions to discover better action selections for the future. Different strategies have been designed to address this trade-off problem, including ideas from optimal control theory and stochastic approximation[18].

2.7 Components of Reinforcement Learning

The components of the RL system consist of distinct states, actions, rewards, a policy, the environment and in some cases a model of the environment. Figure 2.1 shows the basic components of the RL system.

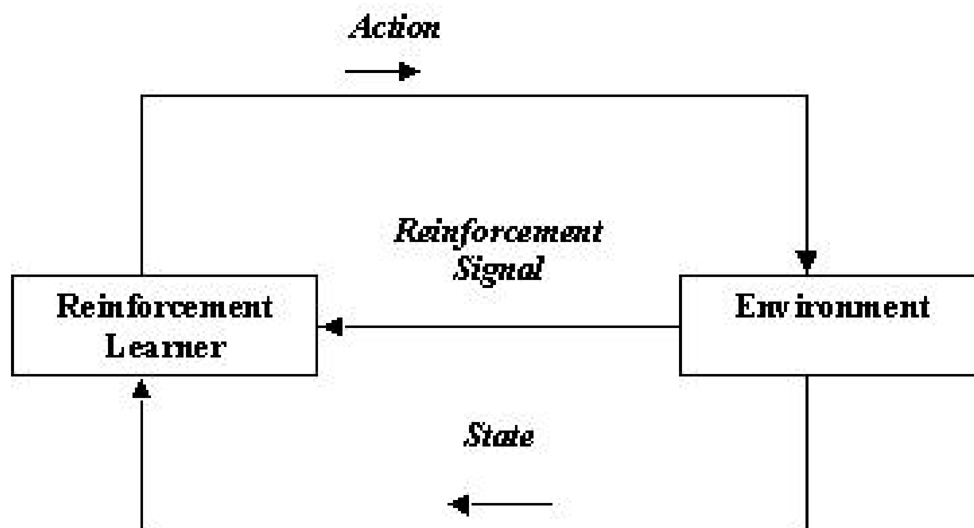

Figure 2.1 Actor Environment Interaction

The reward is a scalar reinforcement signal, which communicates the changes in the environment to the reinforcement learning system and represents the degree to which a state is desirable or not. They are used to guide the learning process and to specify the wide range of planning goals. The objective of the reward function is to maximize the benefits achieved in the long run. The environment is the basic element of the reinforcement learning system with which the agent interacts through trial and error method. The policy is the mapping function between the states of the environment and actions that are taken when the agent is in that state [11]. Thus, it determines the behavior of the learning agent at a specific instance of time. The policy is the foundation of the RL system, and depending on the problem that needs to be solved, it can be simple or have complex functionalities associated with it. In reinforcement learning, the agent interacts with the environment through the action it takes when it is in a particular state. The general scenario is as follows. The agent receives input(s) when it is in a current state, and takes an action. It chooses an action from a state to produce the output, and changes the state to produce a new state. The value of the transition that produced the change in a state is given as a reinforcement signal. The agent learns the system behavior to maximize the reward obtained on a trial and error basis [16].

2.8 General Concepts of Reinforcement Learning

2.8.1 Markov Decision Process

In reinforcement learning problems, the learning agent can learn from both immediate and delayed rewards. During this process of delayed reinforcement, the agent may not be able to achieve optimal results in the immediate steps, but will learn to achieve optimal results in the long run. The states of the environment and the rewards in reinforcement learning are well defined by the Markov Decision Process (MDP). Basically the MDP consists of a set of states, actions, rewards and transition functions. The transition function probabilistically specifies the next state of the environment as a function of the current state and action. The reinforcement model is said to be Markov if the state transitions are independent of the previous environment states or actions [11]. The Markov property is important to reinforcement learning because the decisions taken by the agent are considered to be functions of the current state, and help in predicting the next state and expected reward from the current state of the environment.

2.8.2 Value and Policy

All reinforcement learning problems are based on "value functions" in determining the returns that can be expected by being in a particular state. Two important concepts involved here are the "policy" and the "value". A $policy$ is a mapping from a state to an action. It determines the selection of an action from a state by an agent. Given a policy, the expected

return can be obtained from a given state. The $value$ is the expected return that can be achieved by being in a particular state when following a given policy. The $optimal value$ function assigned to each state is the largest return achievable by a given policy [18]. The policy whose value functions are optimal is the $optimal policy$.

Table 2.1 Value Iteration

```
Initialize V(s) arbitrarily
loop until policy good enough
  loop for s ∈ S
    Q(s,a) = R(s,a) + γ Σ_{s'∈S} T(s,a,s')V(s')
    V(s) = max_a Q(s,a)
  end loop
end loop
```

We can find the optimal policy through an optimal value function through value iteration. The policy iteration finds the optimal policy directly without the optimal value function. In this case, the policy obtains the value function as the expected infinite discount reward by being in a particular state, executing a given policy. The policy is modified if any improvement is possible. When a stage is reached where no further improvements could be made, the policy converges as an optimal policy.

Table 2.2 Policy Iteration

Choose an arbitrary policy π
loop
  compute the value function of policy π:
    solve the linear equations
      $V_\pi(s) = R(s, \pi(s)) + \gamma \Sigma_{s' \in S} T(s, \pi(s), s') V_\pi(s')$
  improve the policy at each state:
    $\pi'(s) = argmax_a(R(s, a) + \gamma \Sigma'_s \in ST(s, a, s') V_\pi(s'))$
  $\pi = \pi'$
until no further improvement is possible

2.9 Learning Optimal Policy

There are two designs for learning optimal policy by the learning agent. One is through a model-free learning and the other is through a model-based learning. The model-based method uses a model and the utility function from that model. The model-free approach uses a action-value function (Q) and does not require a model for the learning process.

2.9.1 Model Free Methods

The Temporal Difference (TD) is a model free approach of reinforcement learning. It is based on a combination of Monte-Carlo (MC) and Dynamic Programming. The TD updates the estimates from the learned estimates like Dynamic Programming and learns without a model like Monte Carlo method. The TD is the most promising method of learning without a model of the environment and the probability distributions associated with the next states (unlike in dynamic programming). Unlike in Monte Carlo method, which waits

for the returns at the end of the episode, the TD gets the estimates at one time step. The TD methods are classified as "on-policy" and "off policy". The$StateActionRewardStateAction(SARSA)$ and $ActorCriticMethods$ are on-policy methods, and $Qlearning$ and $Rlearning$ are off-policy methods. The actor critic method learns a policy iteration through an actor which is the policy structure to select the actions. The critic criticizes the actions taken by the "actor" using the reinforcement signal it receives from the environment, and finds the error associated with each selection. A positive error maximizes the benefit of choosing a particular action, and a negative error minimizes the benefit of choosing that particular action in the future. The SARSA learning algorithm 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 was proposed by Watkins [20, 19], is the most widely used form of reinforcement learning, and is the most effective method for delayed reinforcement. The Q function directly approximates to Q*(the optimal value) independent of the policy being followed. In Q learning, the agent chooses the action with the maximum Q value from a particular state.

2.9.2 Model Based Methods

The model based methods are Dyna, Prioritized Sweeping and Real time dynamic programming (RTDP). In Dyna, a model is built from experience and the policy is modified

from the experience gained from the model. This model suffers from a computational complexity than the model free Q learning method.

2.10 Summary

This chapter presented the background of machine learning and some discussion of the reinforcement learning techniques. A variety of algorithms for automating the selection of the optimal scheduling algorithm for the purpose of load balancing time stepping scientific applications for the learning process were reviewed. We set the stage for choosing the appropriate reinforcement learning technique. The best choice for the problem is to find a model free approach, as building the model for real time applications is a time consuming process. It is not possible to train the model for all possible conditions, unlike in the case of a specific reinforcement learning framework for solving scheduling problems as mentioned in [21, 14]. In addition, because scheduling occurs in a dynamic environment, the training set might not match the particular execution sequence. This chapter sets the stage for an integrated approach to improve the performance of scientific applications by an effective selection of the optimal scheduling algorithm using reinforcement learning techniques. The next chapter describes the design details and its integration into a scientific application.

CHAPTER III

DESIGN AND IMPLEMENTATION

This chapter presents the design and implementation details of a generic framework for automatic selection of dynamic scheduling algorithms using reinforcement learning in parallel and distributed environments. This chapter is organized into three sections. The first section provides the foundations of the design layout for an integrated framework focusing on the automatic selection of scheduling algorithms for large scale scientific applications using reinforcement learning strategies. The second section deals with the extension of the generic design by embedding it within a scientific application(a computationally intensive quantum physics problem) followed by a description of the interaction among the design components. The third section provides information about implementation details.

3.1 Overview

The main source of parallelism in scientific applications is the presence of loops with large number of iterations, which could be executed independently. The effect of variances in iterate execution times due to problem, systemic and algorithmic characteristics lead to load imbalance among the processors, and consequently degrade the performance of the application. A number of loop scheduling algorithms have been proposed and implemented

to address these problems. These algorithms address varying levels of complexity. For a specific parallel loop which is highly irregular finding the optimal scheduling algorithm is difficult. One has to test all the scheduling algorithms in order to find the best strategy to balance the parallel loops. The current implementation of the scheduling algorithms, which are applied for solving complex computational problems, does not explore the possibilities of choosing at runtime among all the dynamic scheduling algorithms, the optimal one, for the particular computational environment the application finds itself at, and the specific time it is executed. This thesis addresses this selection problem by providing a generic design that automatically determines at runtime the optimal scheduling algorithm using reinforcement learning techniques (Refer Figure 3.1).

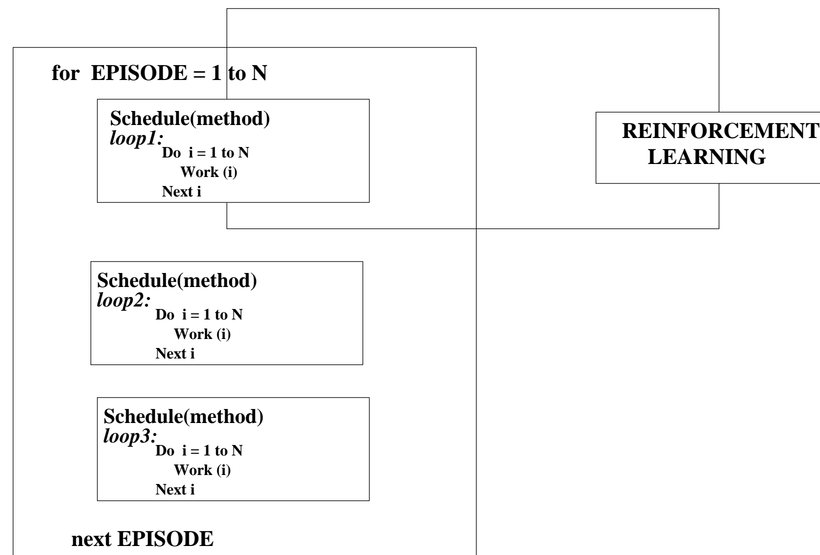

Figure 3.1 Generalized Time-Stepping Scientific Application with Parallel Loops

One of the advantages of using this approach is that no a priori information is needed about the problem characteristics or about the runtime environment before the execution of the application (Refer Figure 3.2).

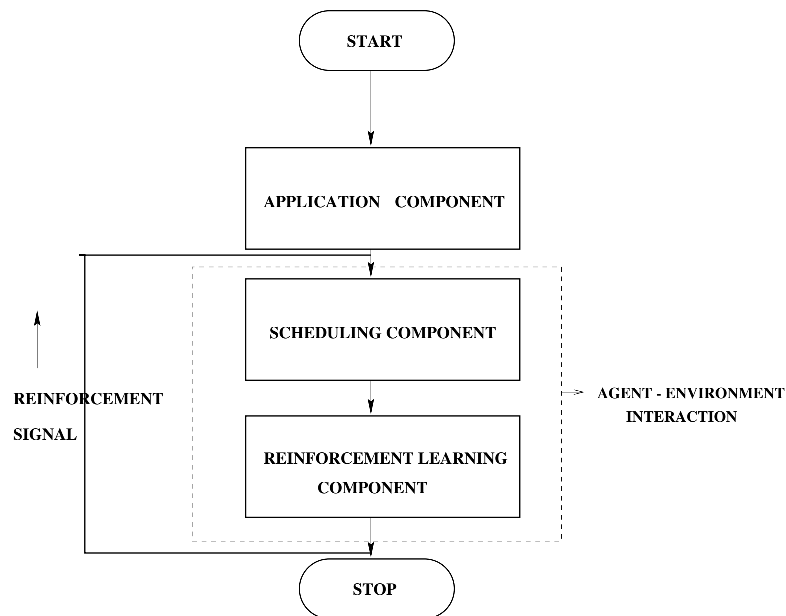

Figure 3.2 General Design

3.1.1 Design Goals

The following are the design goals for the selection framework:

1. Automatic selection of an optimal scheduling algorithm in an unpredictable runtime environment.
2. Provision of a generic design framework for addressing application, architectural and computational environmental irregularities.
3. Portability across different computational platforms.
4. Cost minimization and efficient utilization of computational resources.

5. Provision for incorporating new loop scheduling algorithms and machine learning techniques for future enhancements to solve a wide variety of problems.

3.2 Design Layers

The generic framework consists of three different layers (Refer Figure 3.3).

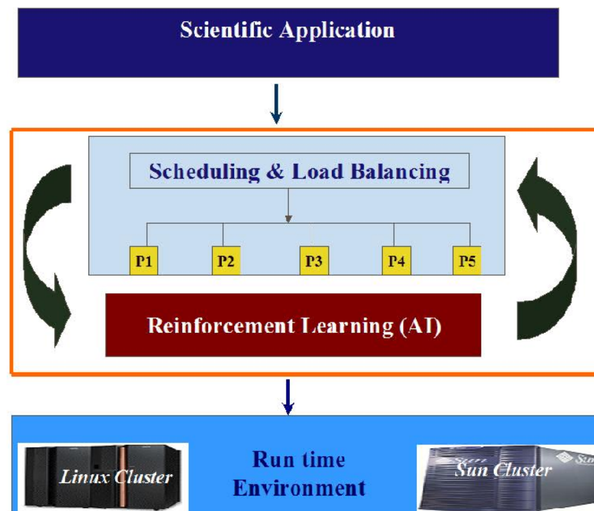

Figure 3.3 Overview

The first layer is the application layer which contains the information and computations specific to the application. The second layer is the scheduling layer which consists of a composition of scheduling algorithms which address the load balancing problem in the parallel loop of the application. The third layer is the learning layer which consists of

algorithms and techniques for the automatic selection of the scheduling algorithm during runtime depending on the performance characteristics of the loop under consideration.

The different layers coordinate to improve the overall performance of the scientific application. The functionality of each layer is modularized such that optimization and software development in one layer can take place independent of other layers.

3.2.1 Application Layer

This layer contains a time-stepping scientific application which contain loops with large number of iterations where scheduling techniques can be applied. The performance of the application is affected by predictable and unpredictable factors. The data distribution may be irregular, or may change dynamically during runtime. The problem characteristics can also change for different architectures, workloads and processors. These factors are very difficult to determine accurately. There may be several loops which have different execution patterns and require different algorithms that address the load imbalance problem efficiently. This layer contains such applications to which scheduling strategies are applied to address the load imbalance problem dynamically at runtime. The selection of the appropriate scheduling algorithm is provided by reinforcement learning techniques.

3.2.2 Scheduling and Load Balancing Layer

This layer provides scheduling routines for performance improvement of loops in complex applications, that are subjected to load imbalance due to problem, algorithmic or systemic

characteristics. The scheduling algorithms are based on master-slave strategy with data replicated on all processors (Refer Table 3.1). The description of the load balancing algorithms incorporated in this layer is described in Chapter 2. During the scheduling process, initially the data is distributed to all the participating processors and each processor is assigned a specific portion of the iterations. In this scheduling process, one processor acts as "master processor" or the "scheduler". All the other processors act as "slave processors" and each processor works on its assigned portion. When there is imbalance in the work loads, a processor works on portions assigned to other processors. After completing the execution of all the loop iterates, the computed results are sent to all the processors to have synchronized data that may be required for further computations (Refer Figure 3.4). The scheduler coordinates with the slave processors in assigning loop iterates, receiving computed results and in sending termination signals to the processors when there is no pending work and the processors have completed the assigned work to them. The scheduler broadcasts the results of the computation to all the processors after the termination of the loop execution. The interaction between the scheduler and the slave processors is facilitated by transfer of messages for specific functions. The WRK-MSG message is sent from the scheduler to all the slave processors to specify the number of iterations and the starting location of data on which each processor is scheduled to work. The RES-MSG message is sent from a processor to the scheduler after the completion of the assigned iterations. REQ-MSG message is sent from the slave processor to the scheduler requesting for work. Finally the END-MSG message is sent from the scheduler to all the processors

signaling the end of computation. The scheduling process interleaves computation with communication by working on a fraction of work and then probing for incoming messages.

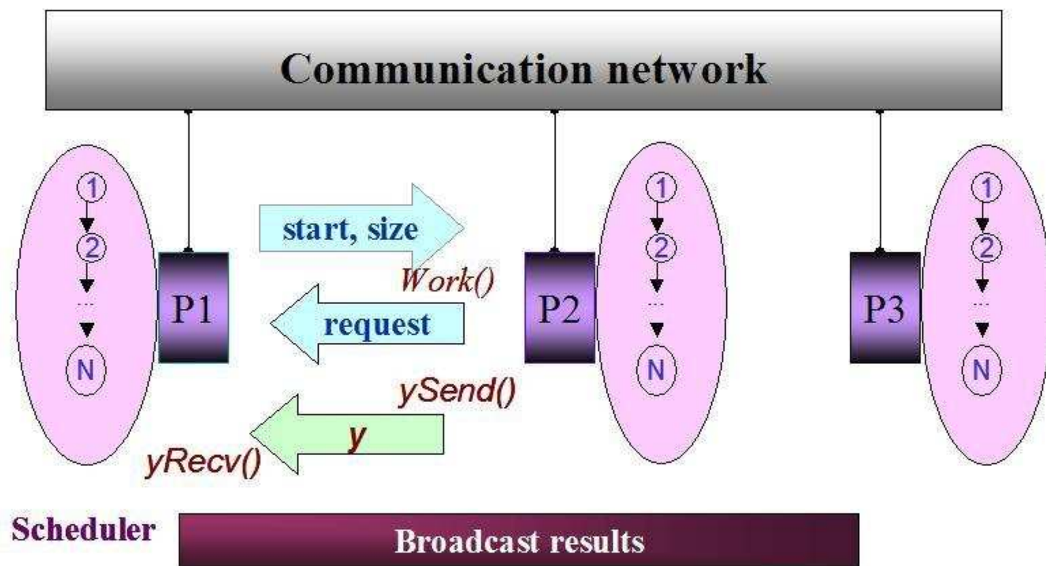

Figure 3.4 Replicated work queue

3.2.3 Reinforcement Learning Layer

This layer provides the interface between the runtime environment and the scheduling layer providing control and optimization to the scheduling layer. This layer addresses the problem of "adaptive learning" of an autonomous agent through its interaction with the dynamic environment by using a combination of exploration and exploitation techniques. The reinforcement layer monitors the performance of the loop scheduling algorithms and specifies the optimal scheduling algorithm by interacting with the runtime environment.

Table 3.1 Scheduling Strategy

Algorithm 1: Loop scheduling strategy

```
Require: Replicate loop data on all processors
if scheduler then
  Initialize scheduling variables
end if
Send REQ-MSG to scheduler
while not terminated do
 while a message was received do
   if message is WRK-MSG then
       Set range for pending work
  else if message is END-MSG then
       Set termination flag
  else if message is REQ-MSG then
    if there is unscheduled work then
       Compute workload
       Respond with END-MSG
   else if sender is not the scheduler then
       Respond with END-MSG
   end if
  else if message is RES-MSG then
       Receive results
  end if
 end while
 if there is pending work then
    Execute a fraction of the pending work
    if work is nearly finished and request not yet sent then
        Send REQ-MSG to scheduler
    end if
    if work is finished then
       Send RES-MSG to scheduler
    end if
 end if
end while
Broadcast results from scheduler
```

The components involved in this layer are the states, actions, rewards and the environment. The "states" are the "loop scheduling algorithms" from the scheduling layer and the $actions$ are the "decisions to select the optimal algorithm". The "rewards" are the "evaluative feedback" which reflects the performance of the scheduling algorithms. The application is provided with a set of scheduling algorithms and the reinforcement learning layer finds the best scheduling algorithm through its interaction with the scheduling layer and the assessment of algorithm performance. This leads to the selection of applying a particular algorithm in an unpredictable environmental condition, selection that otherwise cannot be optmially made.

Table 3.2 SARSA Learning

Initialize Q(s,a) arbitrarily
repeat for each episode
  initialize s
  choose a from s using poicy derived from Q
  repeat for each episode
    take action a, observe r, s'
    choose a' from s' using policy derived from Q
    $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a))]$
    a=a'
    s=s'
  until s is terminal

Table 3.3 Q Learning

Initialize Q(s,a) arbitrarily
repeat for each episode
  initialize s
  repeat for each episode
    choose a' from s' using policy derived from Q
    take action a, observe r, s'
    $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a))]$
    s=s'
until s is terminal

3.2.3.1 Learning Algorithms

The reinforcement learning algorithms used for this integrated approach are Q and $SARSA$ (Refer Table 3.2 and 3.3). These algorithms are based on the “temporal difference methods”. They are not dependent on any particular model for which input-output pairs are needed. The collection of input-output pairs is difficult to achieve for all possible combinations of processors, problem sizes and environmental conditions. Moreover, the pattern provided by the model might not exactly match a particular environmental condition which is subjected to variations due to predictable or unpredictable conditions. Thus, the “model free” approach is suitable for the selection process. This approach overcomes these limitations and provides computational advantages over the “model based” methods.

Table 3.4 Wave-packet Simulation algorithm using QTM

Algorithm 2 : Simulation of quantum wavepacket dynamics using QTM
Initialize time, positions, velocities and probability densities of pseudo-particles
for each time step in turn do
for pseudo-particle i = 1 to N do **[Loop1]**
compute classic potential and classic force
end for
for pseudo-particle i = 1 to N do **[Loop2]**
call MWLS(i, positions, densities)
compute quantum potential Q[i]
end for
for pseudo-particle i = 1 to N do **[Loop3]**
call MWLS(i, positions, quantum potentials)
compute quantum force f[i]
end for
for pseudo-particle i = 1 to N do **[Loop4]**
call MWLS(i, positions, velocities)
compute derivative of velocity dv[i]
end for
output time, positions, densities, velocities, classic force
output quantum potential, quantum force, derivative of velocity
for pseudo-particle i = 1 to N do **[Loop5]**
update densities
update positions and velocities
end for
end for

3.3 Case Study

The integrated design is tested on a computationally intensive scientific application - the wavepacket simulation using the Quantum Trajectory Method (QTM). This is a time stepping application with parallel loops. In QTM, a set of pseudo-particles is used to represent a physical particle. Each pseudo-particle executes a quantum trajectory governed by the Lagrangian QFD-EOM and the quantum potential. The pseudo-particles form a wave packet, which collectively represents the physical particle. The Moving Least Square algorithm is used for curve fitting, and the resulting curve is differentiated to obtain the required derivatives. The parallelization process involved the distribution of the computational loops over the pseudo-particles (Refer Figure 3.5). The distributed memory version of the QTM was selected for experimentation in this thesis. There are three major loops where the core of the computation is concentrated. The loops are used to calculate the "quantum force", the "quantum potential" and the "divergence of velocity". In the previous studies, one specific scheduling algorithm is suited for all the three loops with no concern regarding their individual computational requirements. The algorithm is specified statically during the initiation of the computation process and does not change throughout the application. To cater to the specific characteristics of these different computational loops, reinforcement learning techniques are employed in selecting the optimal scheduling algorithm for individual loops.

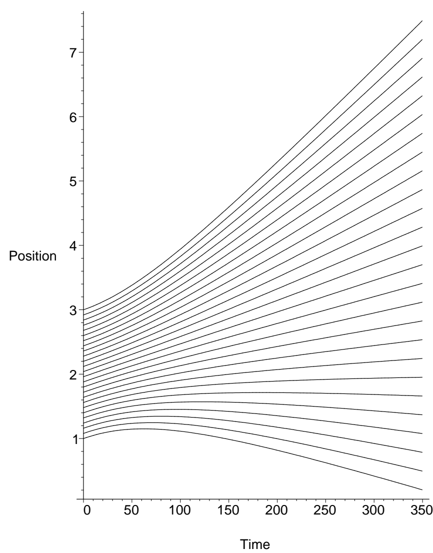

Figure 3.5 Wave packet Simulation of free particles using Quantum Trajectory Method

3.3.1 Problem Mapping

The load imbalance in the computation of quantum trajectories in the wavepacket simulation problem running in a distributed environment has been effectively addressed by novel loop scheduling algorithms. The QTM problem is mapped to the reinforcement learning problem as follows. The scheduling algorithms for achieving load balancing are modeled as different states which are selected dynamically depending on the variability of the environment. The actions determine the particular choice of an algorithm from one state to another. The transition from one state to another takes place as shown in Figure 3.7. The arcs represent the transition from one state to another. The environment is the dynamic system of clusters on which the application is executed. The reinforcement signal or reward is modeled using the parallel cost which is the product of the number of processors and the loop execution time, which is given as feedback to the reinforcement learner. In each loop, each component (the quantum potential, the quantum force and the divergence of velocity), has an individual reinforcement learner which can select the best algorithm suited to that loop, thereby reducing the overall computation time when compared to the traditional approach of exhaustive execution. The total number of time-steps in the application is modeled as episodes.

3.3.2 Algorithm Selection Mechanism

The process of selecting the best scheduling algorithm depending on the dynamic environmental conditions can be considered as an MDP, hence reinforcement learning techniques

are suitable for solving the problem. The scheduling algorithms are equivalent in terms of the problem they solve. These algorithms address the dynamic nature of the runtime environment at different complexity levels. The algorithm selection problem can be modeled as a MDP with states, actions, parallel costs, transition functions and a "value minimization function" or the "objective function". The states are represented by the various scheduling algorithms, that also include information about the execution of the particular algorithm during the previous time-steps. The actions constitute the selection of a particular scheduling algorithm given the performance of all the algorithms and the previous algorithm that were executed " a priori". The state action transitions are provided from the state action transition matrix that is updated after every execution of the chosen algorithm, which is used in the action selection process during the next time-step (iteration) or the episode. The Q and $SARSA$ reinforcement techniques are used for obtaining the optimal value selection and in finding the optimal policy. The "cost" for the current iteration is used in the decision of selecting and executing the scheduling algorithm during the next time-step. The goal of the selection process is to reduce the overall computational time by making efficient use of the parallel processors (Refer Figure 3.6).

3.3.3 State - Action Transition

The graph of the state-action transition is described as follows. After receiving a feedback, the scheduling algorithm can remain in the same state or move to a different state. In the scenario of the problem being considered, the states are defined by various dynamic

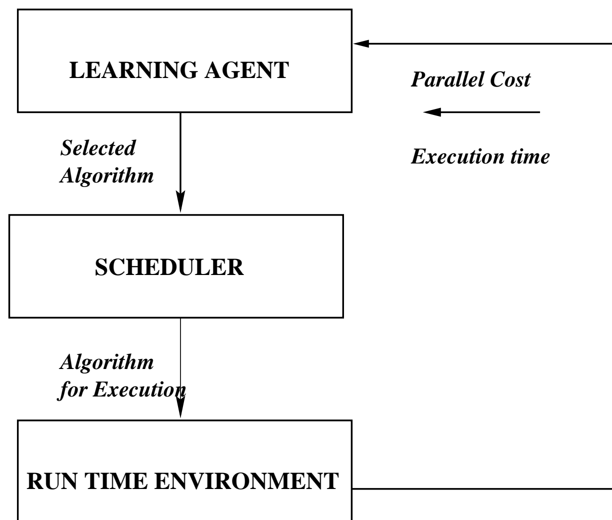

Figure 3.6 Interaction between learner, scheduler and the runtime environment

scheduling algorithms. The feedback value and Q values, which are obtained after the execution of a particular algorithm, are used for learning the best scheduling algorithm during the next time step. Here, we consider that an algorithm is defined by a state and that the reward is defined by the execution time obtained from the runtime environment. The action is defined by the decision to execute or not to execute an algorithm (Refer Figure 3.7). The multiple state transitions is exhibited by the change in state during runtime.

3.4 Implementation Details

The QTM application is coded in Fortran 90. The loop scheduling algorithms contain utility functions in Fortran 90 and uses the MPI message passing paradigm for the execution of the application in a distributed memory environment. The reinforcement-learning component is coded in C and is called from the Fortran application. The architecture is

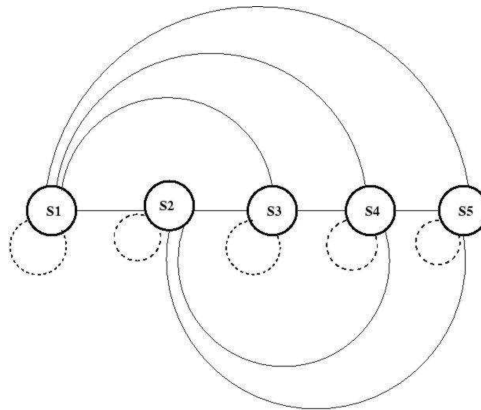

Figure 3.7 State-Action Transition

generic for any operating system on which it is built upon. The design is independent of the underlying architecture and chooses the appropriate functions during the execution.

3.5 Summary

This chapter describes the design details and the various components of the design with their functions. It also discusses the extension of the design by applying it to a time-stepping scientific application which requires the integrated approach to solve the problem. The verification and validation of this novel approach is discussed through experiments and analysis in the next chapter.

CHAPTER IV

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents the experiments and analysis of the results to verify and validate the hypothesis. The main objective of this thesis, to provide an automatic selection mechanism for finding the optimal scheduling algorithm in a dynamic environment without prior execution, was realized by a successful implementation of the proposed integrated design. The performance of the execution of a scientific application on different computational platforms with varying number of processors and different problem sizes proves the portability and the scalability claimed by this approach. The computational advantage and the generic design of this approach is achieved and is validated through exhaustive experiments. The experimental setup and a detailed qualitative and quantitative comparison is discussed in the following sections.

4.1 Experimental Setup

The hardware and software configuration for the experiments is discussed in this section. The experiments were conducted on two operating systems, SUN Solaris (UltraMSPARC cluster) and Linux (Empire cluster) at the Engineering Research Center (ERC) at the Mississippi State University. The UltraMSPARC is a 64 processor cluster with 16 nodes,

where each node contains four ultraSPARC II processors and 32 GB combined RAM capacity. The Empire cluster consists of IBM x330 Linux machines. The cluster consists of 1 GHz and 1.266 GHz Pentium III processors with a cumulative RAM capacity of 607.5 GB. The clusters are interconnected using ATM or Myrinet networks characterized by low latency and high bandwidth.

4.2 Overview of Experiments

The experiments to verify and validate the integrated approach are divided into various categories (Refer Figure 4.1). The main categories are based on the operating systems on which they are executed, namely, $Solaris$ and $Linux$. The next sub-categories of experiments are based on the type of learning methods employed as $No\ Learning$, $Q\ Learning$ and $SARSA\ Learning$. In $NoLearning$, the scheduling technique is specified as an input to the application. In the case of Q and $SARSA$ learning, the scheduling method is provided dynamically at runtime. The third level of experiments are based on the learning parameters. In this classification, the learning rate is varied from 0.001 to 0.9, and the discount rate is varied from 0.1 to 0.9. For each combination of the learning rate and discount rate, and for the $No\ Learning$ category, experiments were conducted for varying number of processors, episodes and wave-packet sizes. The number of processors used were 2, 4, 8, 16 and 32. The number of episodes used were 200, 500, 1000, 2000, 5000 and 10000. Three different wave-packet sizes were used, 501, 1001 and 1501 pseudo-particles.The experimental analysis for the various combinations is discussed in the following sections.

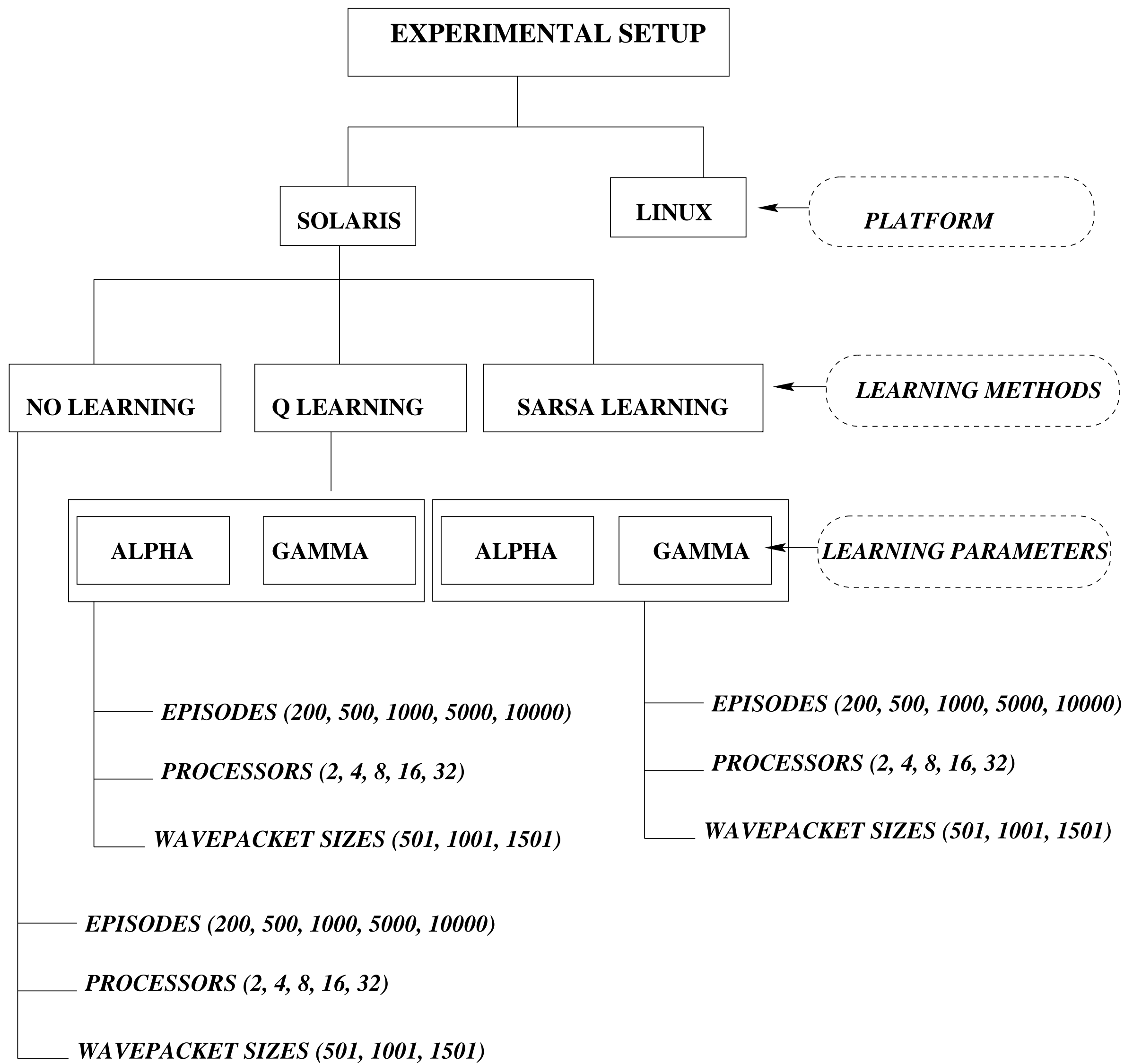

Figure 4.1 Experimental Setup

4.3 Performance Analysis

A comparative analysis is performed from both qualitative and quantitative perspectives, for using reinforcement learning techniques to automate the selection process of finding the best scheduling algorithm to achieve load balancing of scientific applications (with large number of time steps) running in the distributed environment. The scientific application used for the entire analysis is : wavepacket simulation using QTM.

4.3.1 Performance Metrics

The performance metrics used in the evaluation of the generic framework are:

- Parallel Execution Time
- Cost

4.3.1.1 Parallel Execution Time

The parallel execution time, T_p is a function of the specific scheduling algorithm, number of processors P and the computational environment. In the experiments, T_p represents the time required to complete a given computational task [13].

4.3.1.2 Cost

The cost is also used as a performance metric to assess the performance of scientific applications. It indicates the resource utilization in performing the computation. This metric is a direct indicator of the scalability of the scheduling algorithms and is the product of the number of processors and the parallel execution time.

$$\text{Cost} = P * T_p$$

4.3.2 Cost Analysis with and without Learning

A two-fold analysis is performed to validate the integrated design. Firstly, the application of the new approach with reinforcement learning is tested against the implementation without learning on two different operating systems, Linux and Solaris. Secondly, the performance of the reinforcement learning techniques employed in the integrated approach, namely Q and $SARSA$ learning are compared against each other. In figure(4.2), with 200 episodes, the cost variation for varying processors is given for different scheduling algorithms employed individually along with the integrated approach. It can be seen from the graph that the integrated approach performs better than the non-adaptive techniques such as SS, FSC, GSS and FAC, and even against the advanced adaptive techniques such as AF, AWF-B and AWF-C. The cost improvement is consistent for increasing number of processors and for different operating systems. There was 42.33 %, 50.90% and 48.57% cost improvement over the non-adaptive techniques and the average improvement of 29.71% ,15.71% and 20.49% over the adaptive techniques, for 4, 8 and 16 processors, respectively. This validates the hypothesis of the thesis that the integrated approach provides better cost improvement when compared with the approach without learning. Figures 4.3, 4.4 and 4.5 show the cost comparison for 5000, 500 and 200 episodes respectively on the Linux operating system. Refer figures 4.6, 4.7, 4.8 and 4.9 which show cost comparison on the Solaris operating system. Similar results were obtained for 1001 and 1501 wave packet sizes.

From these graphs we find that there is improvement in cost when the number of episodes increases, that allows the system to learn better from more experiences. The results obtained from these comparisons highlight the achievement of the goal of this research work, that of attaining performance improvements using reinforcement learning techniques. By outperforming the approach without learning, the integrated learning approach achieves the design goal of cost minimization and efficient utilization of resources. It also proves that it is “generic”, since it can successfully execute on different computational platforms on different clusters of workstations.

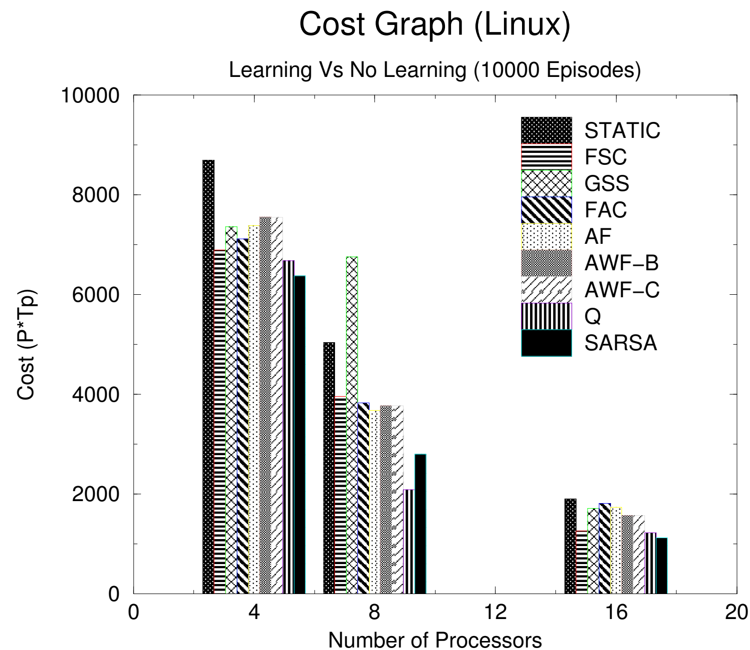

Figure 4.2 Linux : Cost comparison of Learning vs No Learning for 10000 episodes

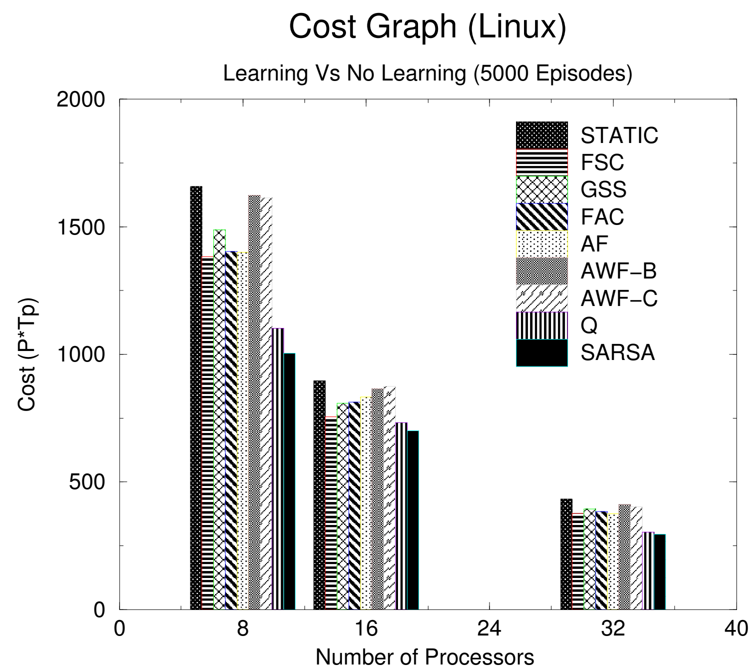

Figure 4.3 Linux : Cost comparison of Learning vs No Learning for 5000 episodes

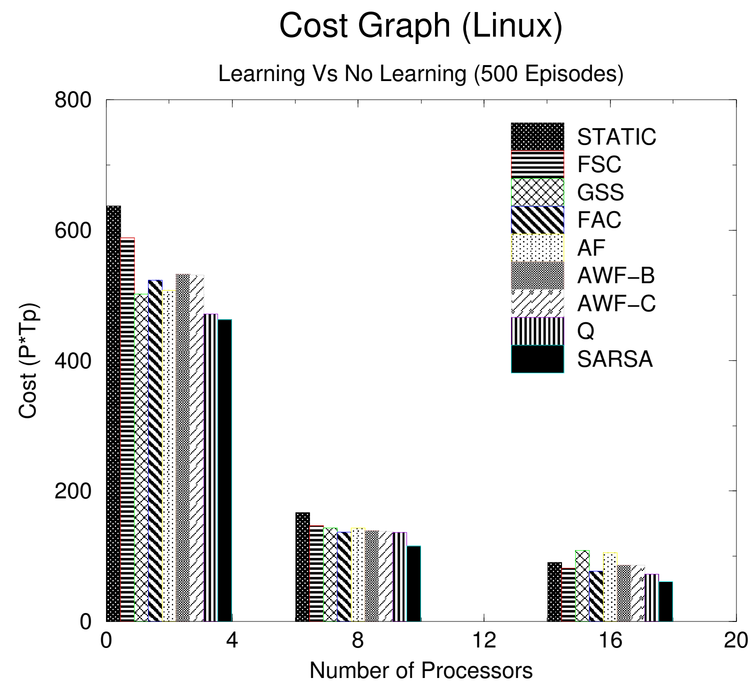

Figure 4.4 Linux : Cost comparison of Learning vs No Learning for 500 episodes

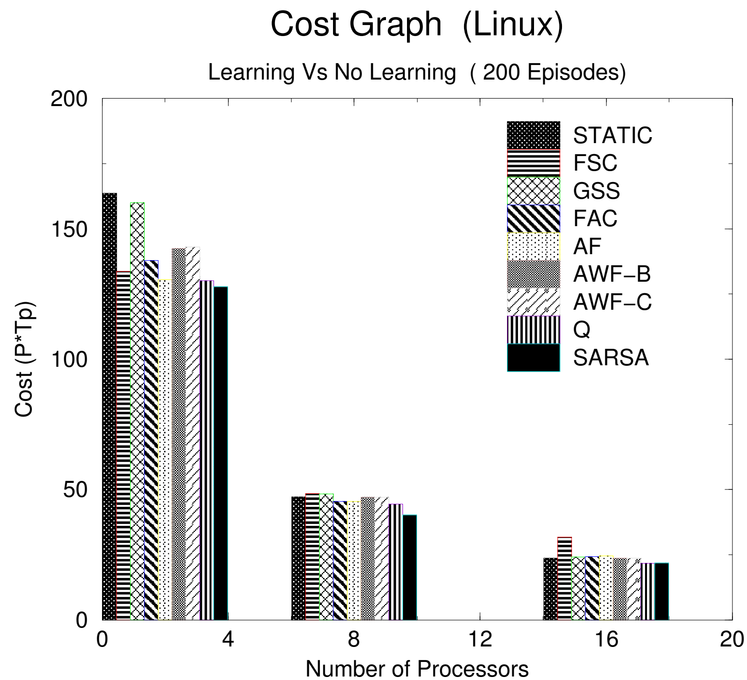

Figure 4.5 Linux : Cost comparison of Learning vs No Learning for 200 episodes

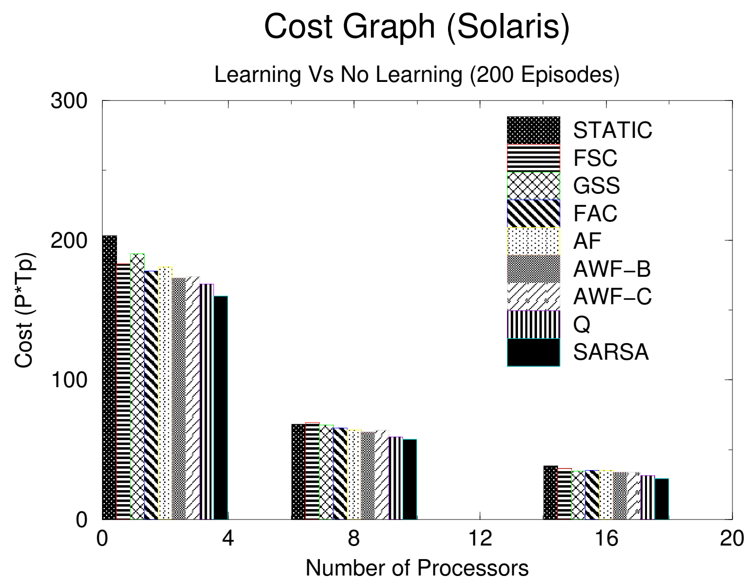

Figure 4.6 Ultra : Cost comparison of Learning vs No Learning for 200 episodes

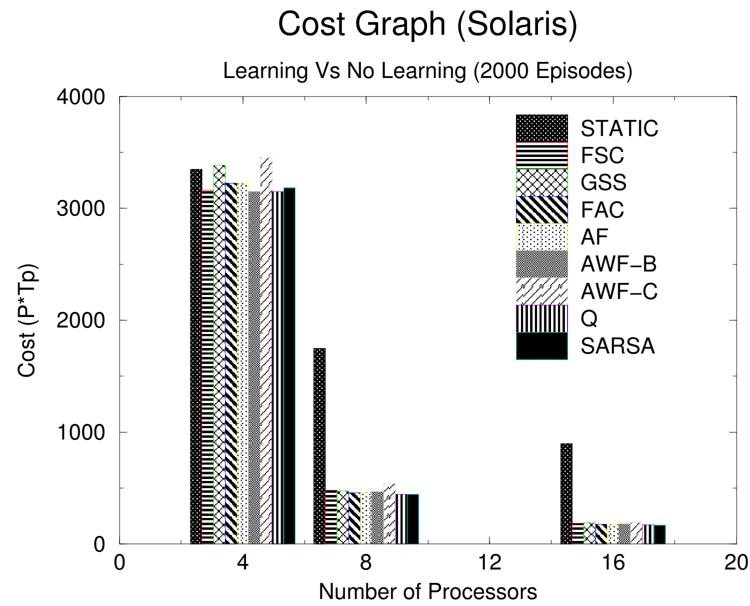

Figure 4.7 Ultra : Cost comparison of Learning vs No Learning for 2000 episodes

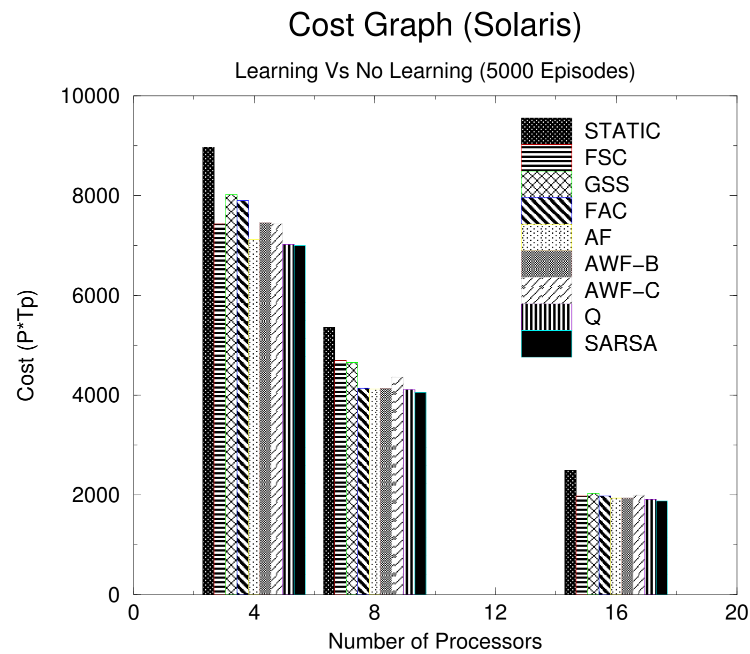

Figure 4.8 Ultra : Cost comparison of Learning vs No Learning for 5000 episodes

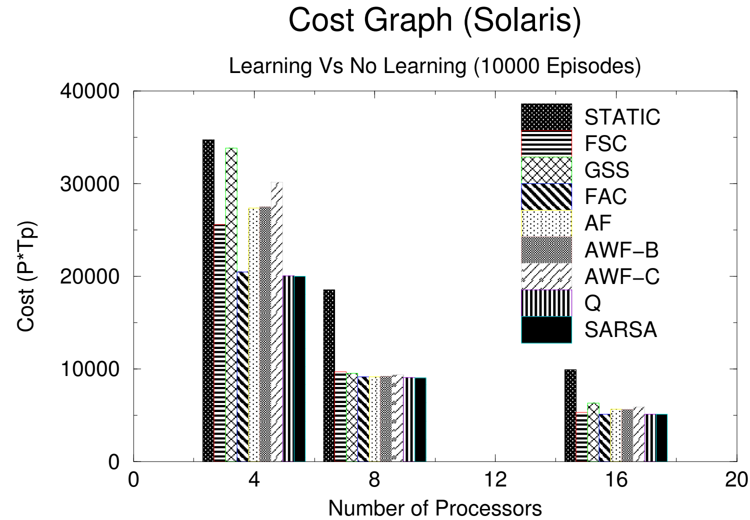

Figure 4.9 Ultra : Cost comparison of Learning vs No Learning for 10000 episodes

4.3.3 Behavior of Q Learning and SARSA Learning for the QTM

Quantitative analysis of the experimental results is viewed from three different perspectives: (i) from the the application perspective when the integrated technique is applied, by analyzing different parallel loops, wave-packet sizes, number of episodes and the number of processors; (ii) from the scheduling algorithms perspective, by analyzing the selection of the loop scheduling algorithms for a specific parallel loop of computation; and (iii) from the reinforcement learning perspective by varying the learning rate and the discount rate and measuring the effect these parameters have on the performance of the application.

4.3.4 Scheduling Methods Selected for the Computational Loops

The application of the reinforcement learning algorithms for load balancing complex scientific application at a finer granular level within an application was achieved by employ-

ing reinforcement learning techniques specific to a parallel loop, which dynamically meets the requirements imposed by the computations. This is illustrated by the graphs following this section. In these graphs, the performance of three different computationally intensive parallel loops are analyzed by employing the reinforcement learning techniques. Figure 4.16 shows the selection of scheduling algorithms for the three parallel loops in a time-stepping scientific application for 1000 episodes, for a wave packet size of 1001, on 2 processors with a learning rate of 0.3 and a discount rate of 0.1, by employing Q learning on Linux machines. The loop scheduling algorithms employed are static scheduling (Static), Fixed Size Scheduling (FSC), Guided Self Scheduling (GSS), Factoring (FAC), Adaptive Factoring (AF), Adaptive Weighted Factoring - Batch (AWF-B), and Adaptive Weighted Factoring - Chunk (AWF-C). The graphs shows that each parallel loop predominantly selects a specific loop scheduling algorithm dynamically. We see that FSC was selected for 60 $\%$ of the episodes and FAC was selected for 26 $\%$ of the episodes, for the computation of the quantum potential. Similarly, AF and AWF-C were selected for 60 $\%$ of the episodes and for 26 $\%$ of the episodes, for the quantum force, and AF was selected for 76 $\%$ of the episodes for the parallel loop which computes the divergence of velocity. Initially, all the algorithms are executed for equal number of times (in this case, till 200 episodes). After the initial exploration, the algorithm which performs best when compared to the previous episodes is executed for the particular environmental condition. The optimal algorithm selected for the three computational loops, quantum potential, quantum force and divergence of velocity is shown in the graphs. The graph shows that each

computational loop selects a different scheduling algorithm based on the complexity of a particular parallel loop of computation. This justifies the need for using individual reinforcement learners for each of the loops, such that optimal selection is achieved local to the loop. This will contribute to optimization of the entire application.

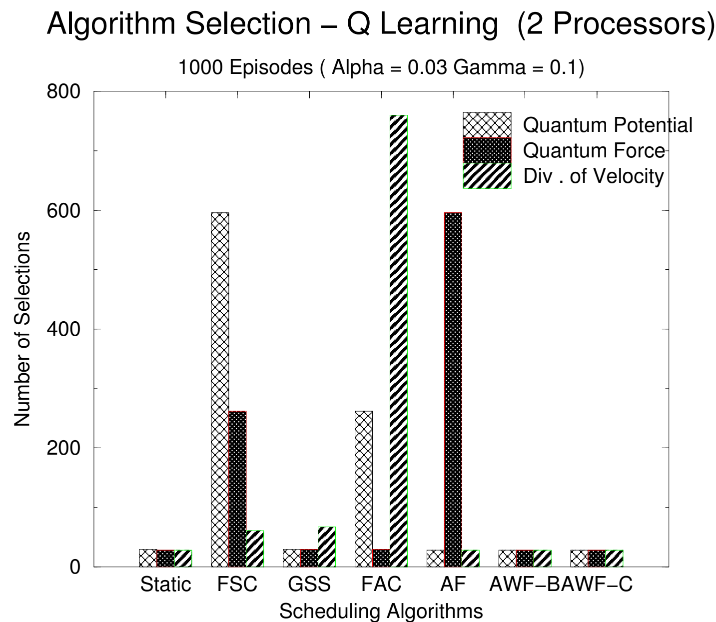

Figure 4.10 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001

The scheduling algorithm selected for 500 episodes with $SARSA\ Learning$ for a wave-packet size of 501 particles on Solaris are shown in Figure 4.20, Figure 4.24, and Figure 4.22 for 2, 4 and 8 processors respectively . In Figure 4.23, Figure 4.24, Figure 4.25 and Figure 4.26 the algorithm selection for wave-packet size of 1001 particles is illustrated. Figure 4.27, Figure 4.28 and Figure 4.29 show the algorithm selected for wave-packet

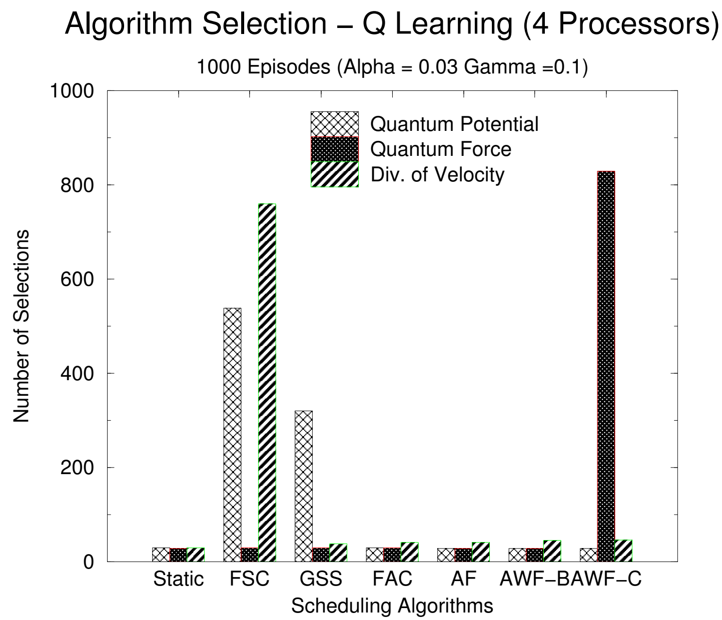

Figure 4.11 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001

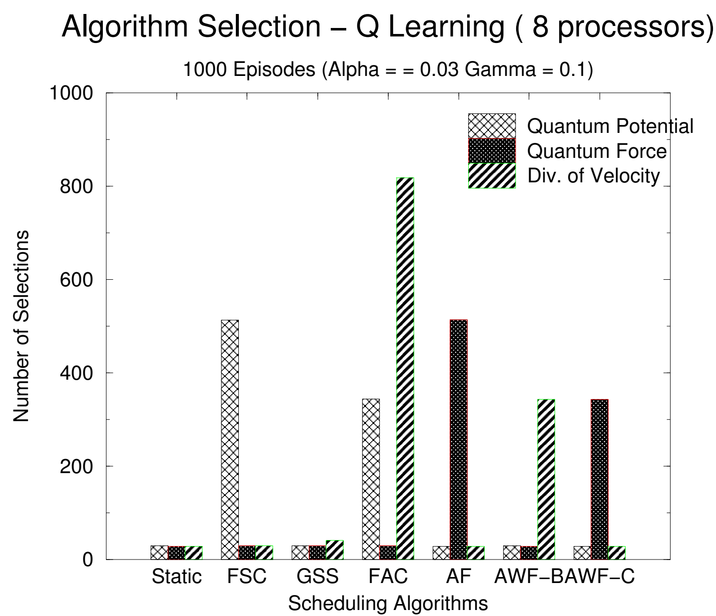

Figure 4.12 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001

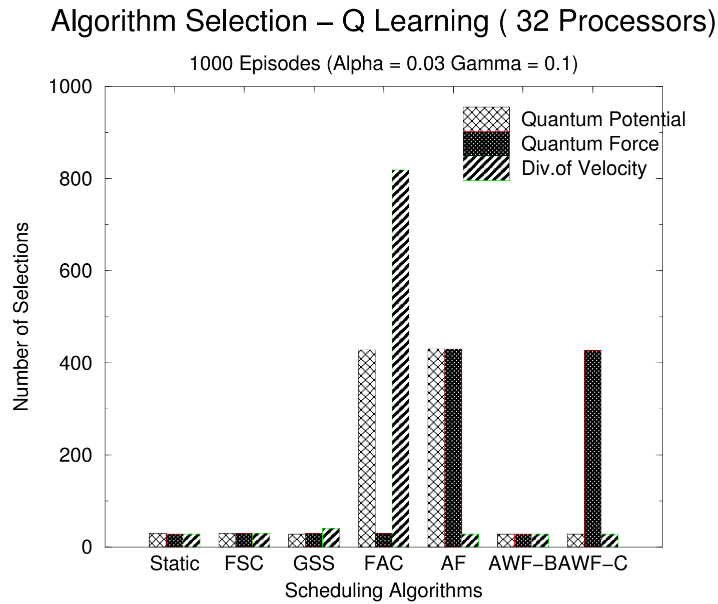

Figure 4.13 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1001

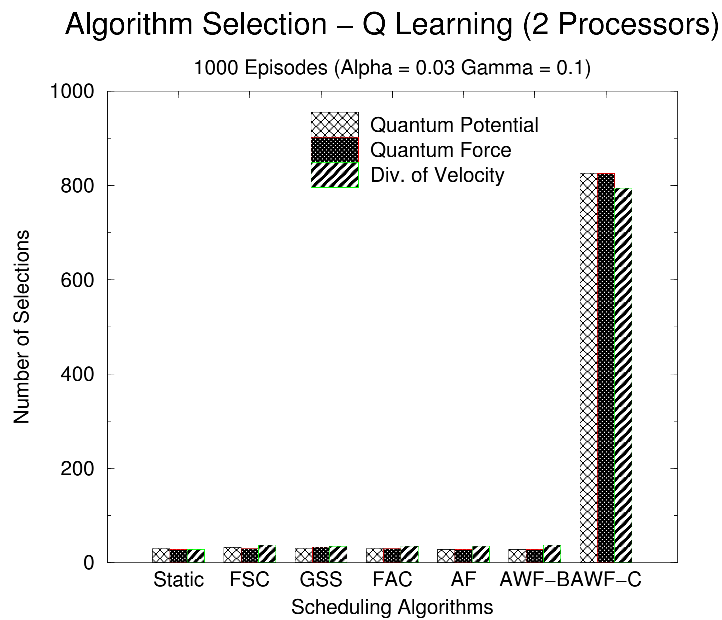

Figure 4.14 Linux: Algorithms selected by Q Learning - Wave-Packet Size :501

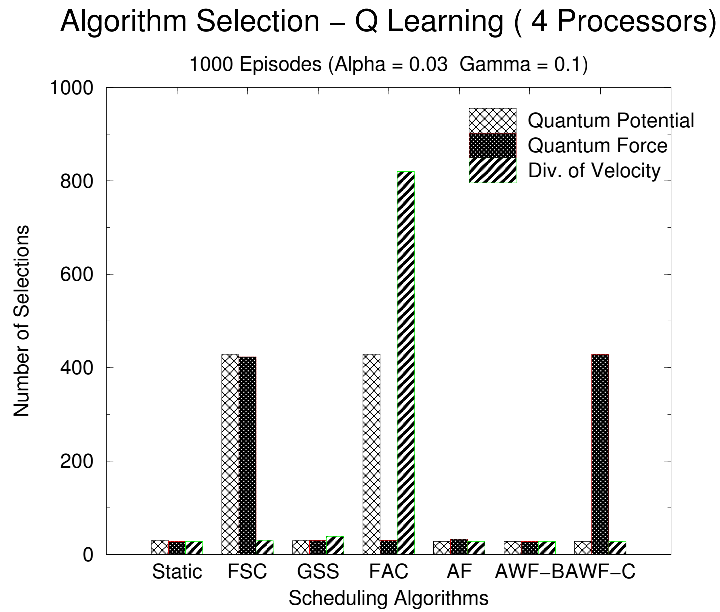

Figure 4.15 Linux:Algorithms selected by Q Learning - Wave-Packet Size :501

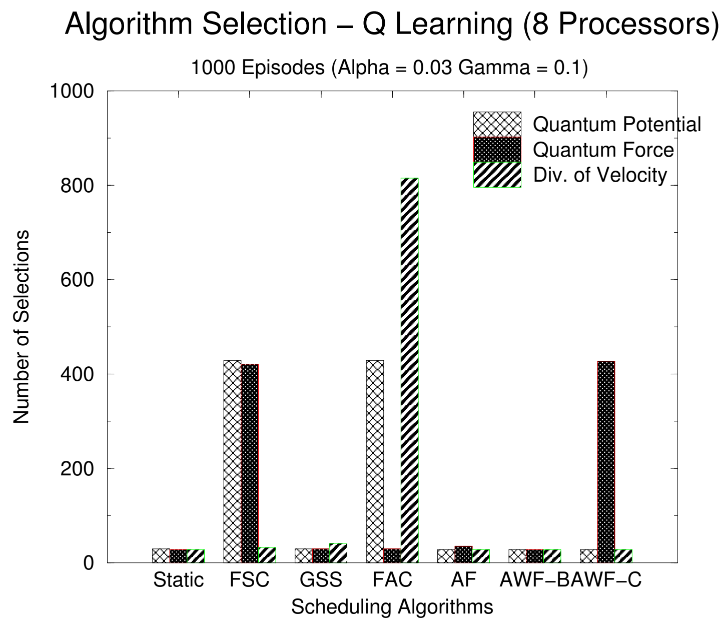

Figure 4.16 Linux: Algorithms selected by Q Learning - Wave-Packet Size :501

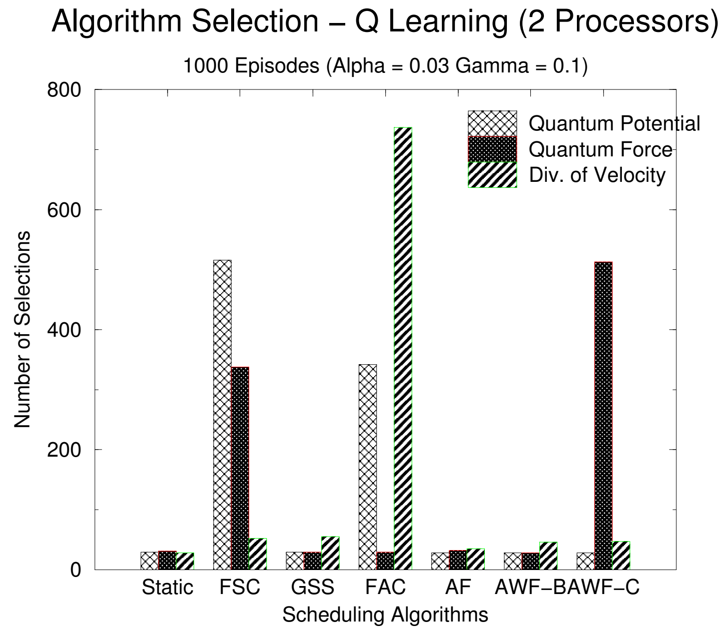

Figure 4.17 Linux:Algorithms selected by Q Learning - Wave-Packet Size :1501

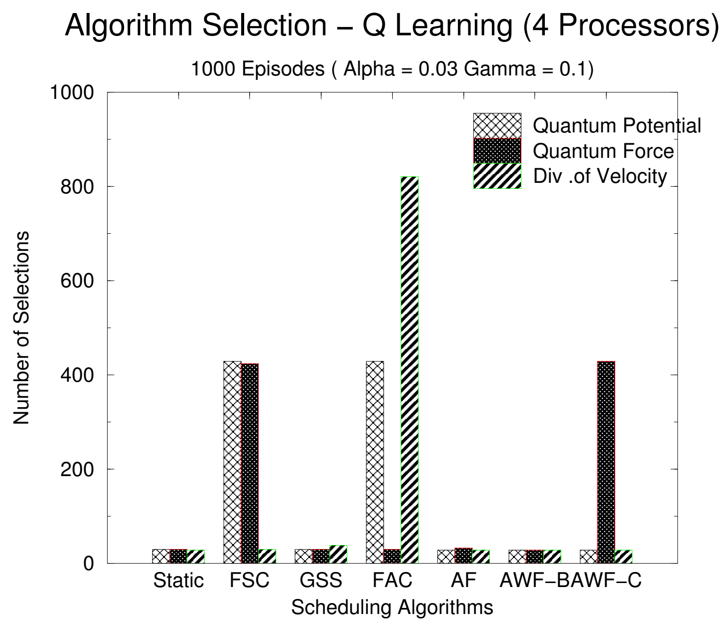

Figure 4.18 Linux: Algorithms selected by Q Learning - Wave-Packet Size :1501

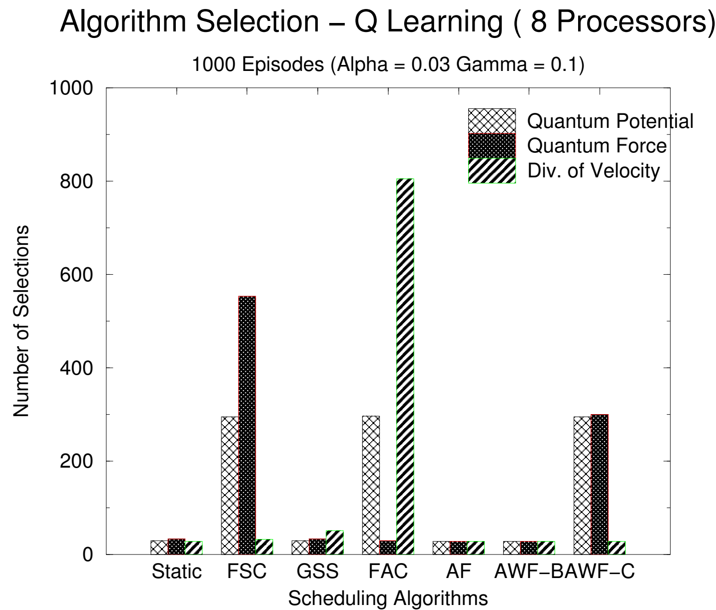

Figure 4.19 Linux: Algorithms selected by Q Learning - Wave-Packet Size : 1501

size of 1501 particles. Similar patterns were observed for different processors, operating systems, wave packet sizes, and number of episodes.

4.3.5 Effect of Varying the Learning Rate on the Performance of QTM

The effect of varying the reinforcement learning parameter, the learning rate in the performance improvement of the integrated approach is analyzed in this section. The tradeoff between exploration and exploitation, which is indicated by the "learning rate", is analyzed by varying it from low learning rate of 0.001 to a high learning rate of 0.9, for a fixed discount rate, which is the metric for choosing future reinforcements. In Figure 4.30, we see the cost variation for a constant value of the "discount rate" (Gamma) for a variable value of the "learning rate" (Alpha) ranging from 0.1 to 0.9. The graph shows the

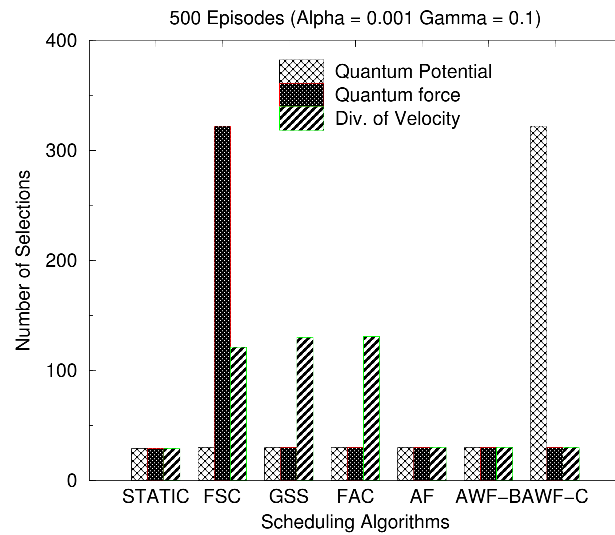

Figure 4.20 Solaris:Algorithms selected by SARSA Learning - Wave-Packet Size :501

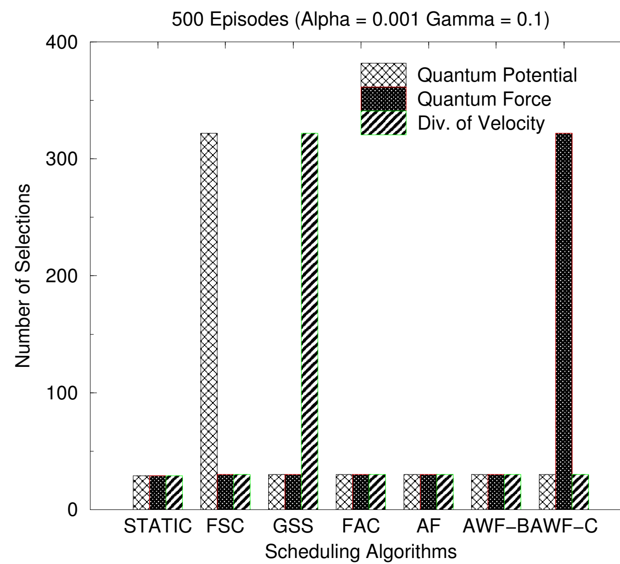

Figure 4.21 Solaris: Algorithms selected by SARSA Learning - Wave-Packet Size :501

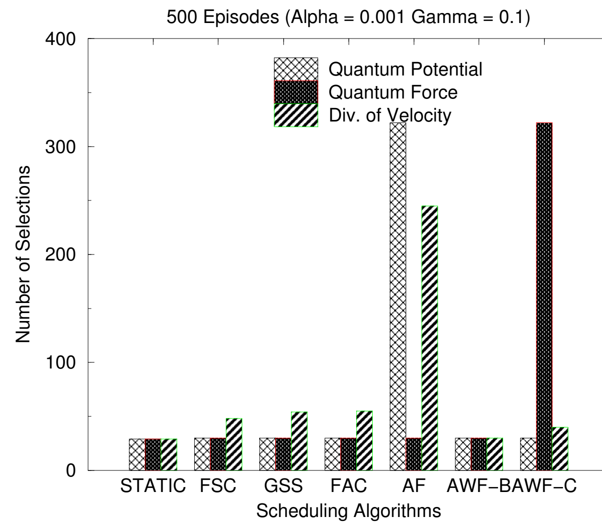

Figure 4.22 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size : 501

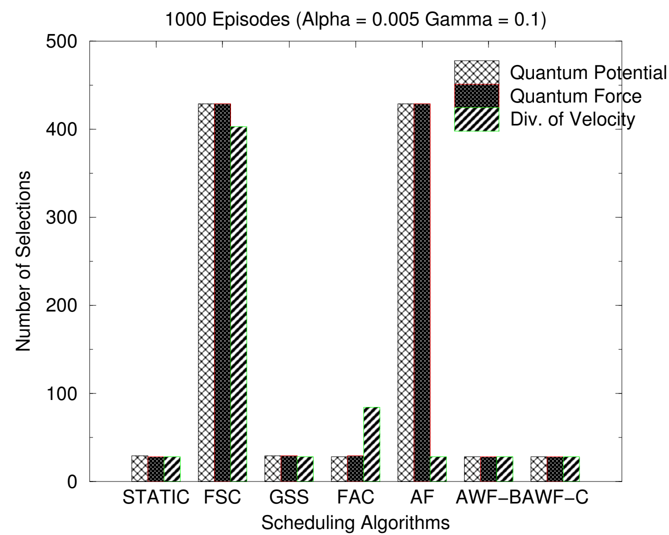

Figure 4.23 Solaris:Algorithms selected by SARSA Learning - Wave-Packet Size :1001

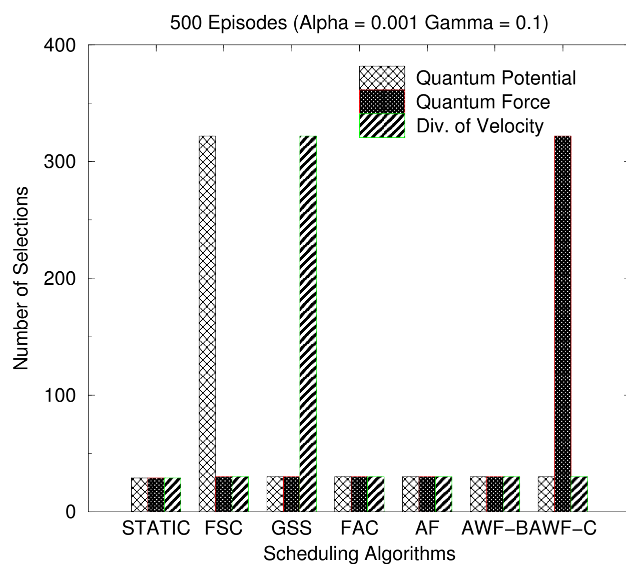

Figure 4.24 Solaris: Algorithms selected by SARSA Learning - Wave-Packet Size :1001

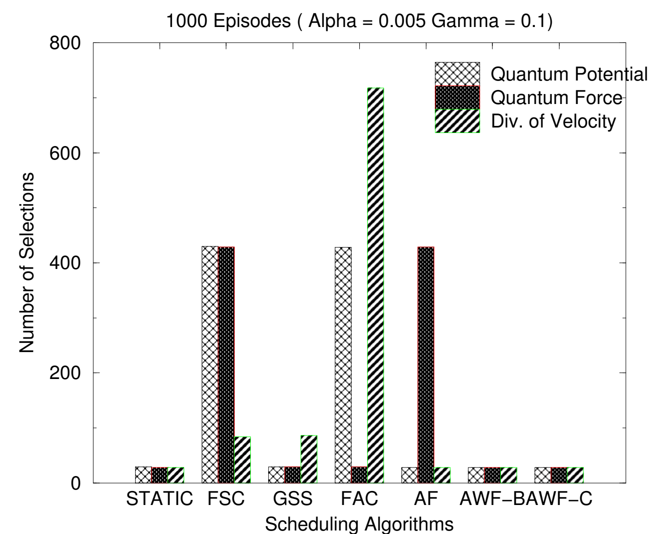

Figure 4.25 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size : 1001

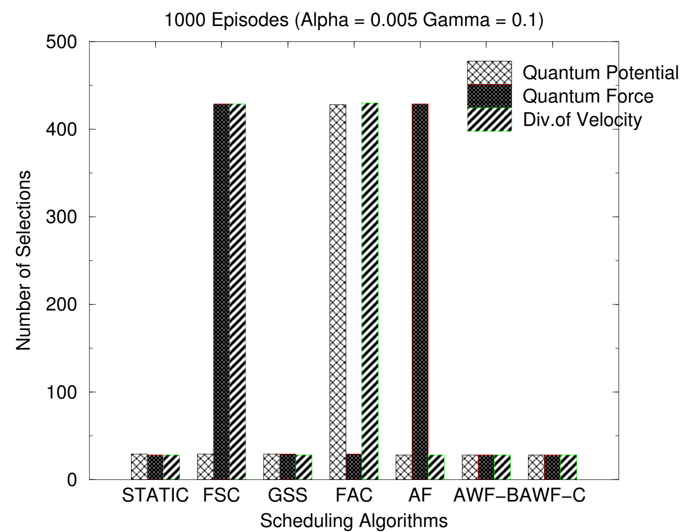

Figure 4.26 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size : 1001

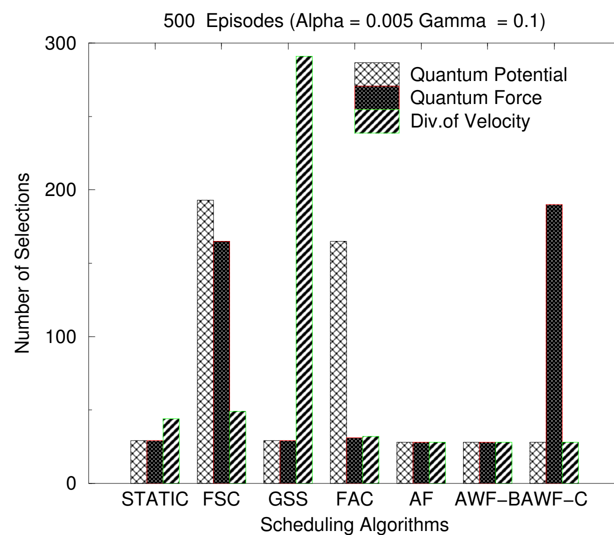

Figure 4.27 Solaris:Algorithms selected by SARSA Learning - Wave-Packet Size :1501

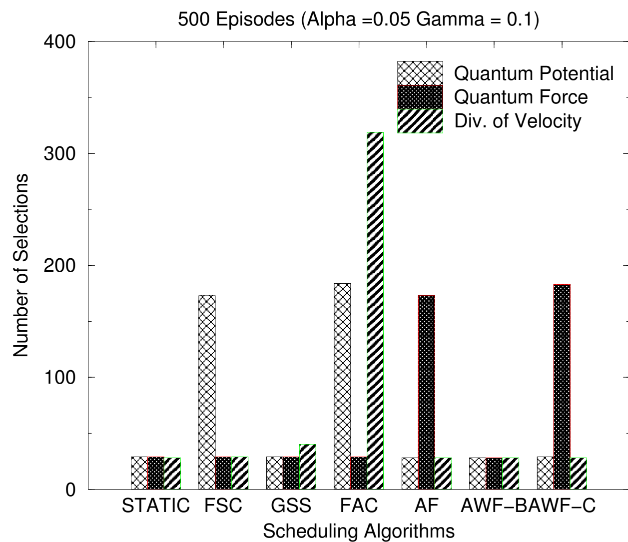

Figure 4.28 Solaris: Algorithms selected by SARSA Learning - Wave-Packet Size :1501

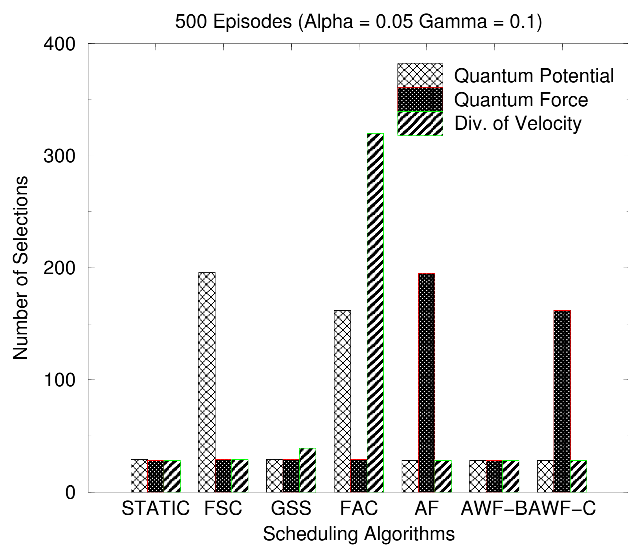

Figure 4.29 Solaris: Algorithms selected by SARSALearning - Wave-Packet Size :1501

cost variation for varying learning rate for 500 episodes and for a wave packet size of 501 particles on the Solaris operating system. There is a 14 % and a 24 % cost improvement for a lower value of alpha at 0.1 when compared to the higher value of alpha at 0.9, for 2 and 8 processors, respectively. The performance improvement was consistent when alpha value was chosen at or below 0.3 for constant value of the discount rate. Similar results were repeated by fixing different values of the discount rate for both Q and $SARSA$ learning.

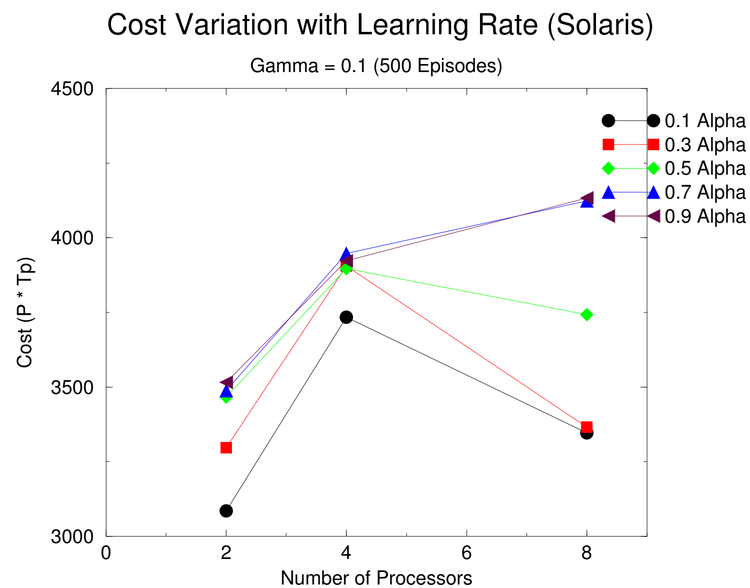

Figure 4.30 Solaris: Variation of cost with learning rate for 500 episodes

4.3.6 Effect of Varying the Discount Rate on the Performance of QTM

The cost variation of the discount rate for constant value of the learning rate is shown in Figure 4.37, Figure 4.38, Figure 4.39 and Figure 4.40 for 500, 1000, 5000 and 10000

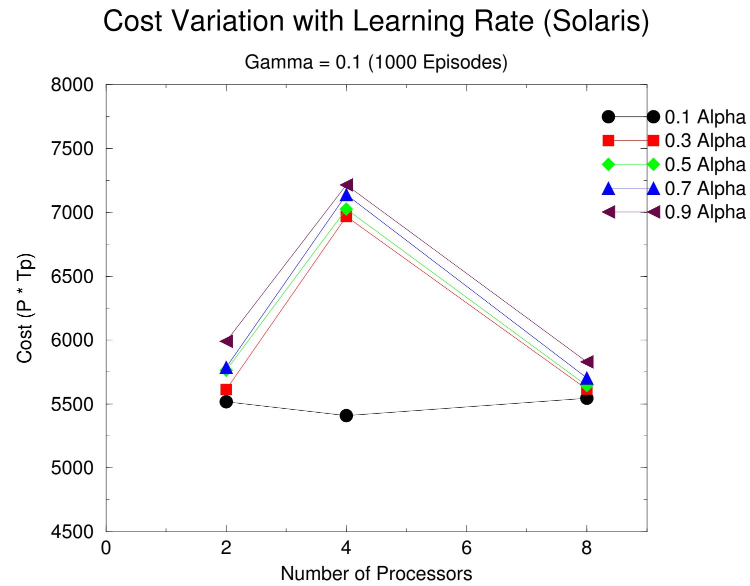

Figure 4.31 Solaris: Variation of cost with learning rate for 1000 episodes

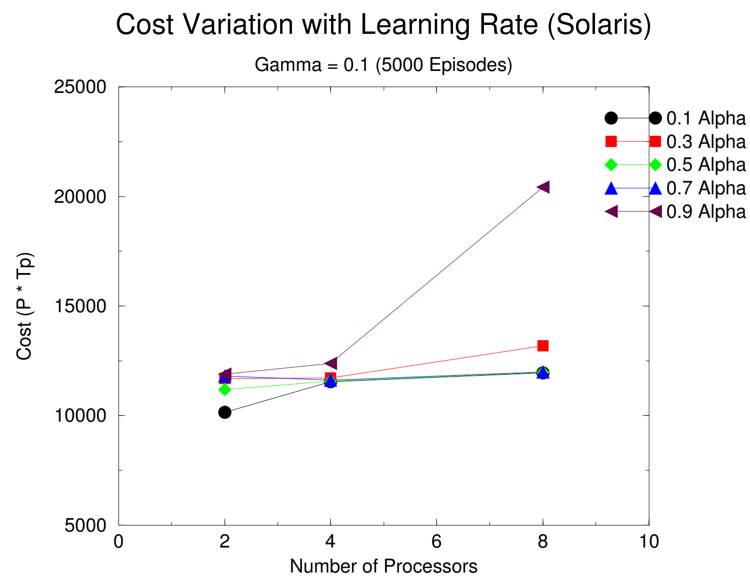

Figure 4.32 Solaris: Variation of cost with learning rate for 5000 episodes

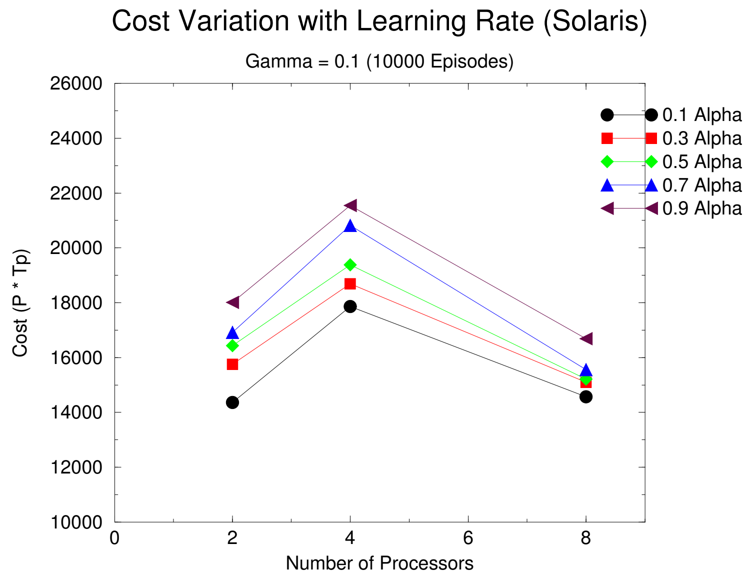

Figure 4.33 Solaris: Variation of cost with learning rate for 10000 episodes

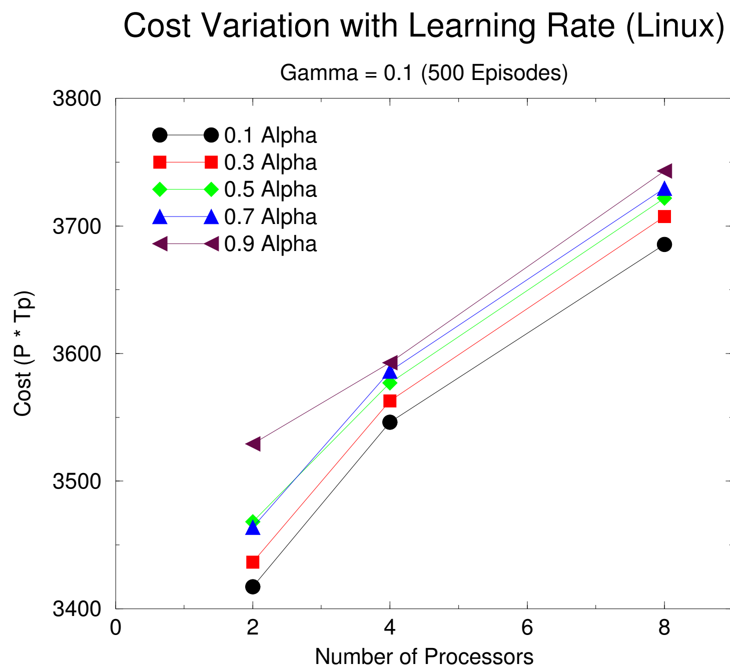

Figure 4.34 Linux: Variation of cost with learning rate for 500 episodes

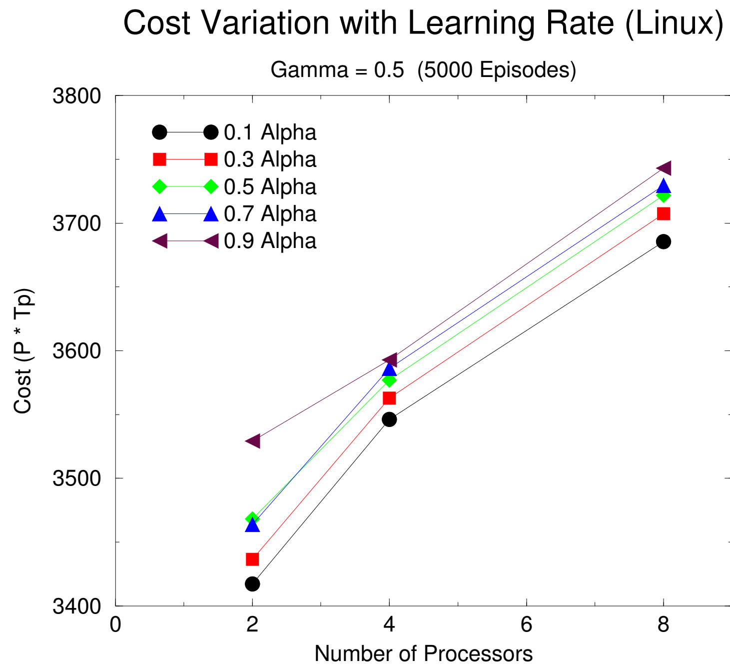

Figure 4.35 Linux: Variation of cost with learning rate for 5000 episodes

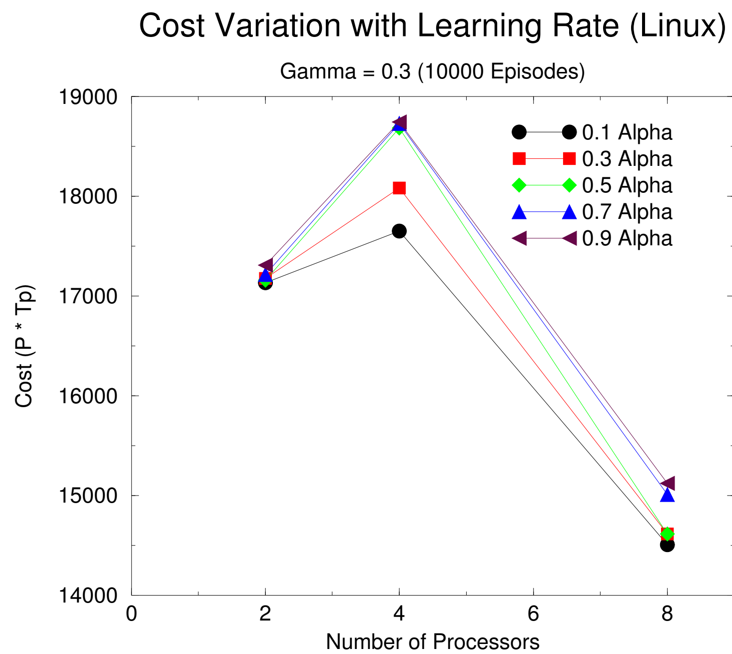

Figure 4.36 Linux: Variation of cost with learning rate for 10000 episodes

episodes, respectively, on the Solaris operating system with Q learning chosen as the reinforcement learning technique. In Figure 4.31, we see the cost variation with the discount rate from 0.1 to 0.9 for a constant learning rate of 0.001. There is considerable cost improvement for lower values of Gamma than higher values of Gamma. This can be attributed to the dynamic variation in the problem characteristics at runtime. From the graph, we can see 12 $\%$, 14$\%$ and 34$\%$ cost improvements for 2, 4 and 8 processors, respectively. This shows that the integrated technique scales well for higher number of processors and for lower discount rates. Similar results were obtained for different combinations of episodes, number of processors, learning algorithms, wave packet sizes, and by fixing different learning rates. From the results obtained, the discount rate of 0.1 proved to be better in most of the cases.

4.3.7 Effect of Varying Episodes and Number of Processors

The effectiveness of varying the problem size and the number of processors is analyzed for fixed learning algorithms, for constant wave-packet sizes and learning parameters. The selection of scheduling algorithms for different problem sizes in terms of number of episodes is compared among Q and $SARSA$ learning. Table 4.1 shows the variation in the selection of the loop scheduling algorithms for three different parallel loops of computation for 1000, 5000 and 10000 episodes, and for a wave packet size of 1501 particles on Linux platform, for 2 processors. The FAC, AF and AWF-C were predominantly selected for

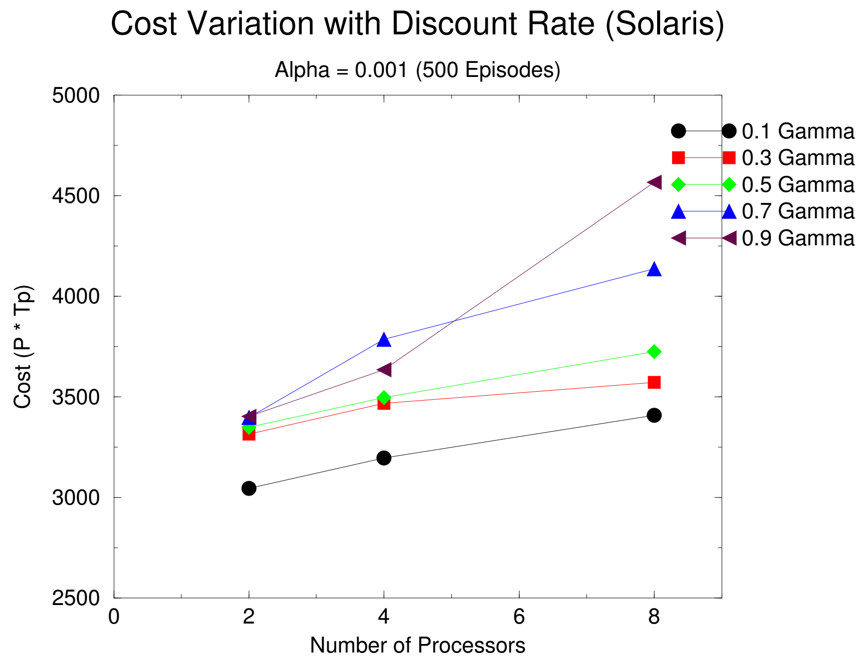

Figure 4.37 Solaris: Variation of cost with discount rate for 500 episodes

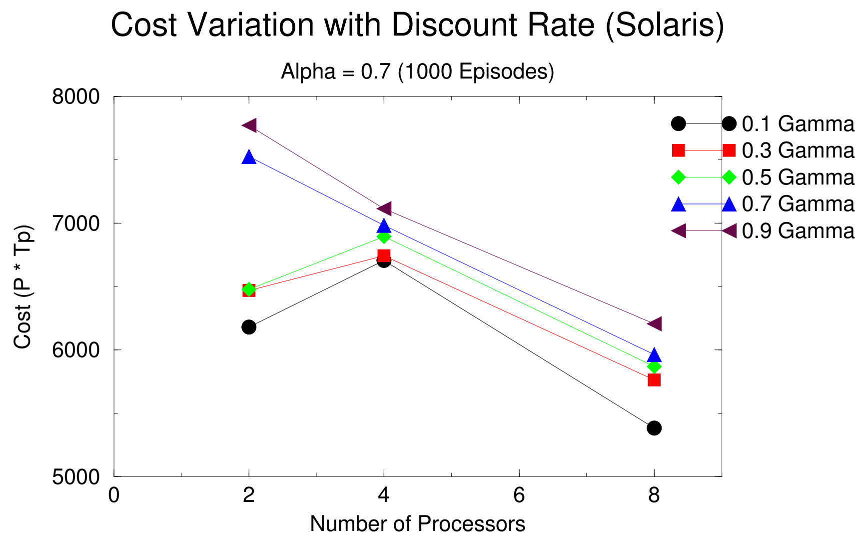

Figure 4.38 Solaris: Variation of cost with learning rate for 1000 episodes

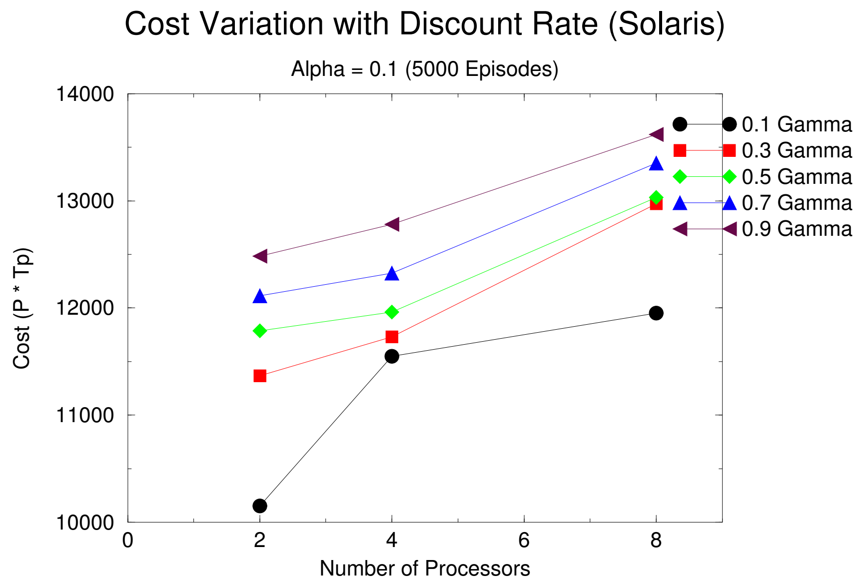

Figure 4.39 Solaris: Variation of cost withdiscount rate for 5000 episodes

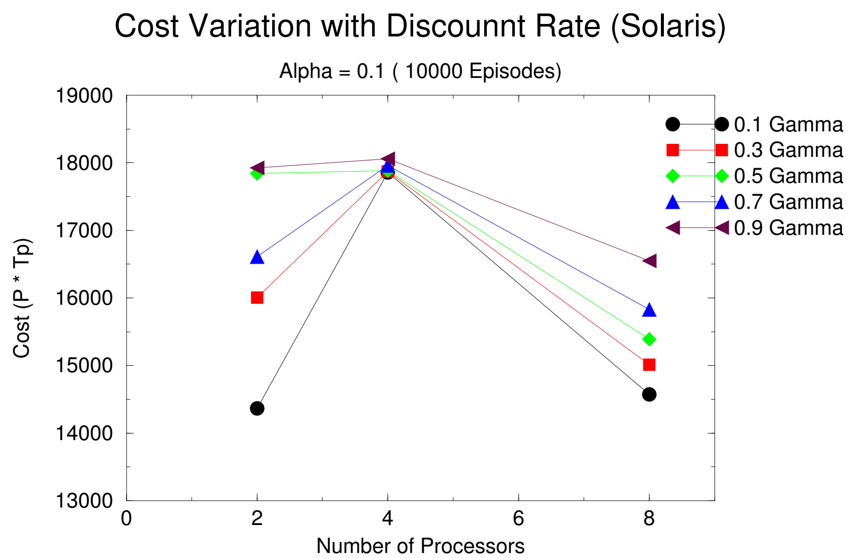

Figure 4.40 Solaris: Variation of cost withdiscount rate for 10000 episodes

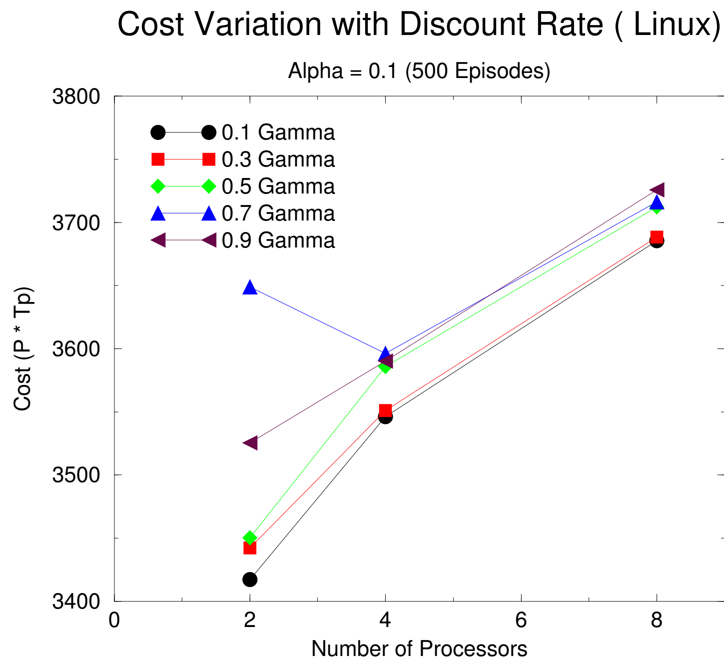

Figure 4.41 Linux: Variation of cost with discount rate for 500 episodes

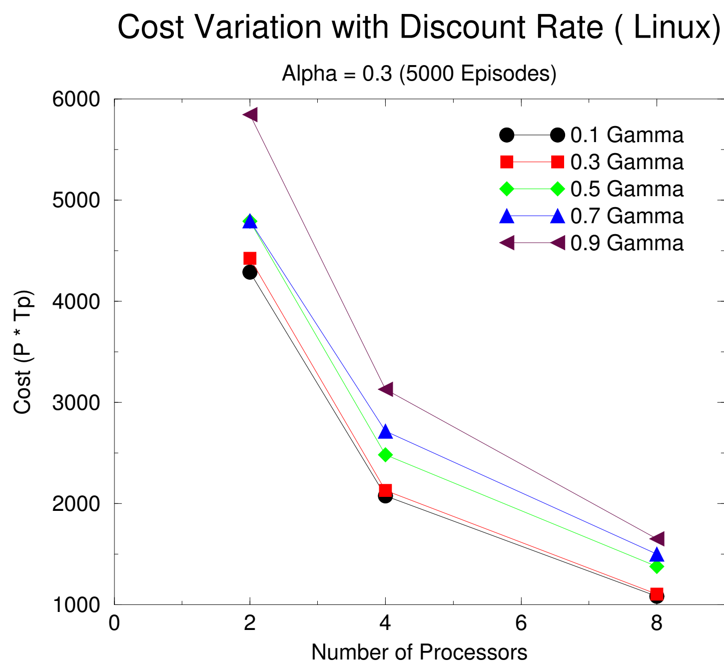

Figure 4.42 Linux :Variation of cost with discount rate for 5000 episodes

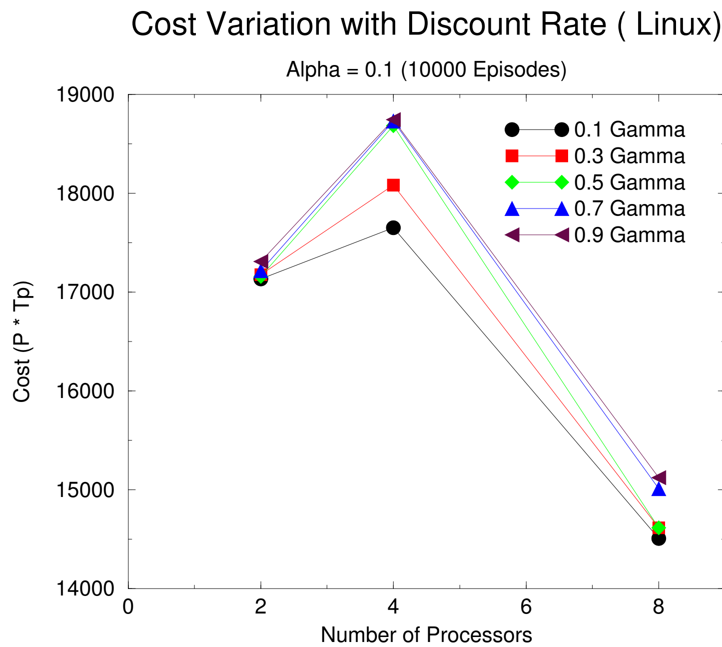

Figure 4.43 Linux :Variation of cost with discount rate for 10000 episodes

both Q and $SARSA$ learning. From the overall analysis, we find that AWF-C and FAC were selected in most of the cases. Similar results were observed for different wavepacket sizes and varying number of processors.

4.4 Summary

This section presents the performance analysis of the integrated design for selecting the scheduling algorithms to load balance complex scientific applications in dynamic environments. The qualitative and the qualitative analysis proves that the application of the automatic selection fo loop scheduling provides considerable performance improvements when compared to the conventional methods of exhaustive testing of individual loop schedul-

Table 4.1 Linux: Scheduling methods selected for 2 processors and varying episodes

Variation in learning algorithms with varying episodes									
Wave packet size: 1501 Number of processors : 2 Cluster : Linux									
Alpha = 0.3 Gamma = 0.1									
Episodes:	1000			5000			10000		
Loops:	QP	QF	DV	QP	QF	DV	QP	QF	DV
Q	FAC	FAC	AF	FAC	FSC	AWF-C	AWF-C	AF	AF
SARSA	AWF-C	FAC	AF	FSC	FAC	AWF-C	AF	AWF-C	AF

Table 4.2 Linux: Scheduling methods selected for 4 processors and varying episodes

Variation in learning algorithms with varying episodes									
Wave packet size:1501 Number of processors :4 Cluster : Linux									
Alpha = 0.3 Gamma = 0.1									
Episodes:	1000			5000			10000		
Loops:	QP	QF	DV	QP	QF	DV	QP	QF	DV
Q	FAC	AWF-C	AWF-C	AWF-C	FAC	AF	FSC	AWF-C	FAC
SARSA	AF	FAC	AF	AF	FAC	FSC	AWF-C	FSC	AF

Table 4.3 Linux: Scheduling methods selected for 8 processors and varying episodes

Variation in learning algorithms with varying episodes									
Wave packet size:1501 Number of processors : 8 Cluster : Linux									
Alpha = 0.3 Gamma = 0.1									
Episodes:	1000			5000			10000		
Loops:	QP	QF	DV	QP	QF	DV	QP	QF	DV
Q	AWF-C	AF	AF	AF	FAC	AF	FSC	AWF-C	AF
SARSA	FAC	FSC	AWF-C	AWF-C	AF	FSC	AWF-C	AF	AF

Table 4.4 Solaris: Scheduling methods selected for 2 processors and varying episodes

Variation in learning algorithms with varying episodes									
Wave packet size:1001 Number of processors : 2 Cluster : Ultra									
Alpha = 0.1 Gamma = 0.1									
Episodes:	1000			5000			10000		
Loops:	QP	QF	DV	QP	QF	DV	QP	QF	DV
Q	AF	FAC	AWF-C	AF	AF	AWF-C	FSC	AWF-C	FAC
SARSA	AWF-C	FAC	AWF-C	AWF-C	AWF-C	AF	FSC	AF	AWF-C

Table 4.5 Solaris: Scheduling methods selected for 4 processors and varying episodes

Variation in learning algorithms with varying episodes									
Wave packet size:1001 Number of processors : 4 Cluster : Ultra									
Alpha = 0.1 Gamma = 0.1									
Episodes:	1000			5000			10000		
Loops:	QP	QF	DV	QP	QF	DV	QP	QF	DV
Q	AWF-C	FAC	AWF-C	FSC	AWF-C	AF	FAC	AWF-C	AF
SARSA	AWF-C	Af	AF	AWF-C	AWF-C	FSC	FSC	AF	AF

Table 4.6 Solaris: Scheduling methods selected for 8 processors and varying episodes

Variation in learning algorithms with varying episodes									
Wave packet size:1001 Number of processors : 8 Cluster : Ultra									
Alpha = 0.1 Gamma = 0.1									
Episodes:	1000			5000			10000		
Loops:	QP	QF	DV	QP	QF	DV	QP	QF	DV
Q	AF	AWF-C	AF	AF	AWF-C	AF	AWF-C	AWF-C	AF
SARSA	AWF-C	AWF-C	AF	AWF-C	AF	FSC	AWF-C	AF	FSC

ing techniques. The experimnetal results obtained using the integrated approach consistently outperform the ones using the conventional method from the application perspective, scheduling algorithms perspective and the reinforcement learning perspective by employing automatic selection and addressing the load balancing problem at finer granular level, which validates the hypothesis. The justification of a need for the use of such an integrated design to address the variation in computational requirements within a application is justified from the results obtained. The portability and the scalability of the integrated approach is revealed by the experimental results obtained for different operating systems and for different number of processors, episodes and wavepacket sizes. The results obtained support all the design goals by a successful implementation, and a succesful validation of the approach by experimental results.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the challenging issues that were addressed and the contributions made by this thesis. It is followed by an outline of the conclusions and the possible directions for future research work.

The motivation, the research problem and the contributions are presented in the first chapter. The primary motivation for this thesis arose from the need for an automatic selection of scheduling algorithm for scientific applications in a distributed environment using machine learning. The hypothesis is well supported by the design and implementation of the integrated design, which improves the performance of large applications with parallel loops whose iterate execution times vary dynamically. Embedding the integrated technique into the scientific applications improves the performance by cost minimization and efficient utilization of computational resources.

The literature review presents the research advancements in two areas, scheduling and reinforcement learning, and highlights the need to integrate these two research domains for time-stepping applications running on distributed memory architectures. The third chapter presents a detailed discussion about the design and implementation details of the generic framework. The fourth chapter presents various experiments conducted to verify

and validate the integrated approach. It also gives a qualitative and quantitative analysis of the performance from the numerous experimental results.

The integrated design was exhaustively tested from the application, scheduling algorithms and reinforcement learning perspectives. From the application side, the selection of the optimal scheduling algorithm was tested for varying problem sizes, and computational loops. From the reinforcement learning point of view, performance analysis was conducted for varying learning rate and discount rate for $Qlearning$ and $SARSA$ learning. From the scheduling point of view, the selection of the optimal scheduling algorithm was analyzed for different problem size, number of processors. The interplay of the parameters of each design layer with the other design layer was also analyzed. The analysis validated the hypothesis and the performance improvement achieved using the integrated design. It also proved the portability, scalability and the genericity, which were the design goals of this thesis. The generic design facilitates the software development of the application by providing the capability of continuous and successive integration of new scheduling techniques or reinforcement learning techniques.

5.1 Future Work

The extension for this framework is to incorporate monitoring control to account for the overhead, latency and bandwidth in the dynamic environment, which contribute to variations in the system. A possible way to achieve this is to use reinforcement learning techniques within the scheduling and load balancing layer, where monitoring of the spe-

cific parameters such as latency can be undertaken. Another path for the extension of this research is to model the parallel processors as individual reinforcement learning agents. In this research direction, each processor can request the master processor for work depending on its computational capacity, and can provide co-ordination with other processors in small clusters. This method will be advantageous when there is a need to control a large number of processors and will facilitate the development of effective hierarchical scheduling and load balancing schemes.

REFERENCES

[1] M. Balasubramaniam, *Performance Analysis and Evaluation of Dynamic Loop Scheduling Techniques in a Competitive Runtime Environment for Distributed Memory Architectures*, master's thesis, Mississippi State University, Mississippi State, Mississippi, May 2003.

[2] I. Banicescu, S. Ghafoor, V.Velusamy, S. Russ, and M. Bilderback, "Experiences from Integrating Algorithmic and Systemic Load Balancing Strategies," 2001, vol. 13, pp. 121–139, John Wiley and Sons Ltd.

[3] I. Banicescu and S. F. Hummel, "Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations," *Proceedings of Supercomputing'95 Conference (on CD-ROM)*, San Diego, California, 1995, IEEE Computer Society Press.

[4] I. Banicescu and Z. Liu, "Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes," *Proceedings of High Performance Computing Symposium*, 2000, pp. 122–129.

[5] I. Banicescu and V. Velusamy, "Load Balancing Highly Irregular Computations with Adaptive Factoring," *Proceedings of the IEEE - International Parallel and Distributed Processing Symposium (IPDPS 2002), (on CD-ROM)*, Fort Lauderdale, Florida, 2002, IEEE Computer Society Press.

[6] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the Scalability of Adaptive Weighted Factoring," *The Journal of Networks, Software Tools and Applications*, vol. 6, no. 3, 2000, pp. 213–226.

[7] K. Govindaswamy, *An API for Adaptive Loop Scheduling in Shared Address Space Architectures*, master's thesis, Mississippi State University, Mississippi State, Mississippi, July 2003.

[8] S. Haykin, *Neural Networks : A Comprehensive Foundation*, second edition, Prentice Hall, Upper Saddle River, New Jersey, 1999.

[9] S. Hummel, I. Banicescu, C.Wang, and J.Wein, "Load Balancing and Data Locality via Fractiling: an Experimental Study," *Languages, Compilers and Run-Time Systems for Scalable Computers*, 1996, pp. 85–98.

[10] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Communications of the ACM*, vol. 35, no. 8, Aug. 1992, pp. 90–101.

[11] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, 1996, pp. 237–285.

[12] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Trans. Software Eng SE-11(10)*, Oct. 1985, pp. 1001–1016.

[13] E. Luke, I. Banicescu, and J. Li, "The Optimal Effectiveness Metric for Parallel Application Analysis," *Information Processing Letters- Elsevier*, vol. 66, no. 5, June. 1998, pp. 223–229.

[14] P. Mehra, *Automated Learning of Load Balancing Strategies for a Distributed Computer System*, doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, 1993.

[15] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans on Computers C-36(12)*, Dec. 1987, pp. 1425–1439.

[16] S. I. Reynolds, *Reinforcement Learning with Exploration*, doctoral dissertation, The University of Birmingham, Birmingham, United Kingdom, December 2002.

[17] S. Russell and P. Norwig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Upper Saddle River, New Jersey, 1995.

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

[19] C. Watkins, *Learning from Delayed Rewards*, doctoral dissertation, University of Cambridge, 1989.

[20] C. Watkins and P. Dyan, "Q Learning," *Machine Learning*, vol. 8, 1992, pp. 279–292.

[21] A. Y. Zomaya, M. Clements, and S. Olariu, "A Framework for Reinforcement-Based Scheduling in Parallel Processor Systems," *IEEE Transactions on Parallel and Distributed Systems.*, vol. 9, no. 3, March. 1998, pp. 249–260.