A Framework for Reinforcement-Based Scheduling in Parallel Processor Systems

Albert Y. Zomaya, Senior Member, IEEE, Matthew Clements, and Stephan Olariu

Abstract—Task scheduling is important for the proper functioning of parallel processor systems. The static scheduling of tasks onto networks of parallel processors is well-defined and documented in the literature. However, in many practical situations a priori information about the tasks that need to be scheduled is not available. In such situations, tasks usually arrive dynamically and the scheduling should be performed on-line or “on the fly.” In this paper, we present a framework based on stochastic reinforcement learning, which is used to solve optimization problems in a simple and efficient way. The use of reinforcement learning reduces the dynamic scheduling problem to that of learning a stochastic approximation of an unknown average error surface. The main advantage of the proposed approach is that no prior information is required about the parallel processor system under consideration. The learning system develops an association between the best action (schedule) and the current state of the environment (parallel system). The performance of reinforcement learning is demonstrated by solving several dynamic scheduling problems. The conditions under which reinforcement learning can be used to efficiently solve the dynamic scheduling problem are highlighted.

Index Terms—Neural networks, parallel processing, randomization, reinforcement learning, scheduling, task allocation.

1 INTRODUCTION

The impressive proliferation in the use of parallel processor systems these days in a great variety of applications is the result of many breakthroughs over the last two decades. These breakthroughs span a wide range of specialties, such as device technology, computer architectures, theory, and software tools. However, there remain many problems that need to be addressed which will keep the research community busy for years to come [32].

A major issue in the operation of parallel computing systems is that of scheduling, which is an important problem in other disciplines such as manufacturing, process control, economics, operation research, and others [10], [16], [29]. To schedule is to simply allocate a set of tasks or jobs to resources in such a way that optimizes the use of these resources. If these tasks are not interdependent, the problem is known as task allocation. In a parallel processor system, one would expect a linear improvement with the increase in the number of processors used. However, this is generally not the case, due to such factors as communication overhead, control overhead, congestion possibilities, and precedence constraints between tasks [7], [16], [21], [29].

The efficiency of a parallel computing system is commonly measured by such factors as completion time, speedup, or throughput, which in turn reflect the quality of the scheduler. Many heuristic algorithms have already been developed which provide effective solutions. Most of these methods, however, can solve only limited classes or instances of the scheduling problem [31].

The scheduling problem is known to be NP-complete for the general case and even for many restricted cases [5], [18]. For this reason, scheduling is usually handled by heuristic methods which provide reasonable solutions for restricted instances of the problem. Most research on scheduling has dealt with the problem when the tasks, interprocessor communication costs, and precedence relations are fully known. When the task information is known a priori, the problem is that of static scheduling. On the other hand, when there is no a priori knowledge about the tasks, the problem is known as dynamic scheduling. For dynamic scheduling problems with precedence constraints (or mutual exclusion constraints), optimal scheduling algorithms do not exist [20].

In nonpreemptive scheduling, once a task has begun on a processor, it must run to completion before another task can start execution on the same processor. In preemptive scheduling, it is possible for a task to be interrupted during its execution, and resumed from that position on the same or any other processor, at a later time. Although preemptive scheduling requires additional overhead, due to the increased complexity, it can perform more effectively than nonpreemptive methods [1], [10]. Further, schedulers can be classified as nonadaptive. A nonadaptive scheduler does not change its behavior according to feedback from the system. This means that it is unable to adapt to changes in system activity. In contrast, an adaptive scheduler changes its scheduling according to the recent history and/or current behavior of the system. In this way, adaptive schedulers may be able to adapt to changes in system use and activity. Adaptive schedulers are usually termed dynamic, since they make decisions based on information collected...
from the system [1], [10]. The scheduler proposed in this work is adaptive in nature.

This paper is divided as follows. In the next section, a brief overview is given of supervised and reinforcement learning paradigms. A more detailed description of reinforcement learning is given in Section 3. The proposed method is given in Section 4, along with the convergence conditions. The experiments are described in Section 5, followed by some examples in Section 6. The results are discussed in Section 7. Finally, conclusions are given in Section 8.

2 Supervised Versus Reinforcement Learning

Starting as early as the 1940s, artificial neural networks, here on referred to as neural networks, began a slow development to the present day [24], [27]. Neural networks (NN) now have many functional paradigms, a few of which are used commonly. NNs are based on a highly simplified model of the structure of the brain. They consist of many neuron-like elements, which all function independently, and there are very many interconnections between these elements [12].

Each element is capable of only simple processing—such as calculating a weighted sum—but the elements can operate in a massively parallel fashion. Although the structure of NNs is very different from that of conventional von Neumann machines, they demonstrate great potential in solving a variety of problems, such as process control, pattern recognition, optimization, and robotics [13], [14], [19].

In general, learning methods have been divided into three main paradigms: unsupervised learning, supervised learning, and reinforcement learning. Unsupervised learning methods do not depend on an external teacher to guide the learning process. Instead, the teacher is built into the learning method. Unlike the unsupervised learning paradigm, both the supervised and reinforcement learning paradigms require an external teacher to provide training signals that guide the learning process. Now, the difference between these two paradigms arises from the type of training signals that guide the learning process.

In the supervised learning paradigm, the teacher provides the learning system with the desired outputs for each given input. Learning involves “memorizing” these desired outputs by minimizing the discrepancy between the actual and the desired outputs of the system [24], [25], [27]. In contrast, the role of the teacher in reinforcement learning is more evaluative than instructional. Sometimes called a critic because of this role, the teacher provides the learning system with a scalar evaluation of the system’s performance of a given task according to some performance measure. The objective of the learning system is to improve its performance, as evaluated by the critic, by generating appropriate actions. The critic in this case does not need to know what each optimal response is in order to provide useful advice. Reinforcement learning thus involves two operations: discovering the right outputs for a given input and memorizing those outputs.

The ability to discover solutions to problems makes reinforcement learning important in situations where the lack of sufficient structure in the task definition makes it difficult to define a priori the desired outputs for each input, as required for supervised learning. In such cases, reinforcement learning systems can be used to learn the unknown desired outputs by providing the system with a suitable evaluation of its performance [2].

For the case of dynamic scheduling, reinforcement learning is more appropriate to use since no a priori information is available on the state of the system. Rather, it is necessary for the scheduler to learn while it operates. As there are no “correct” results to learn from, various measures of the system state need to be used to check the performance at each output. This will be described later in more detail.

3 Reinforcement Learning

Reinforcement learning (RL), as opposed to supervised learning, has a more attractive feature in that it replaces the teacher by a performance measure from the environment to grade the “goodness” of the current actions (Fig. 1).

Measurement of the performance of a scheduler is feasible by studying the efficiency of the computing system (e.g., execution time, throughput, utilization rate). Hence, on-line performance measurements can thus form the basis for adaptive RL-based scheduling. The stochastic nature of RL is compatible with that of the on the fly task allocation and assignment, which is an important requirement for dynamic scheduling.

The way that RL works resembles the operation of a stochastic automaton when performing a search to maximize a payoff or reinforcement function [19], [22], [23]. Fig. 1 shows an RL system that interacts with an environment $E$. At each instant of time $t$, $E$ provides the RL system with some pattern $U \in u \subseteq \mathbb{R}^n$. The RL system produces a random output (or action) $Y(t) \in y \subseteq \mathbb{R}$. The action gets evaluated by the environment in the context of the input $U(t)$ and sends the RL system a reinforcement signal $e(t)$, $e(t) \in \mathbb{R} = [-\epsilon, \epsilon]$, with $e(t) = \epsilon$ representing maximum reinforcement, where $\epsilon \in \mathbb{R}$. According to the goodness (if the error is small) or badness (if the error is large) of the reinforcement signal, the learning system has to produce a better output in order to minimize the error $e(t)$.

Formally, the scheduling problem (within the context of RL) can be stated as follows: At each instant of time $t$, the environment provides the RL system with a vector
form the scheduler design into an optimization problem. Cases which it is doing well by exhibiting behavior that is exploratory behavior, hence, discriminate between cases in which it is doing poorly by using greater degrees of signal it receives indicates the highest degree of success. In input pattern an output value for which the reinforcement function, \(e^5\).

In general, the learning system should satisfy several properties. The scheduler must learn to associate with each input pattern an output value for which the reinforcement signal it receives indicates the highest degree of success. In addition, it should be able to improve its performance in cases where it is doing poorly by using greater degrees of exploratory behavior, hence, discriminate between cases in which it’s doing poorly and those in which it’s doing well. This is important in order not to degrade its performance in cases in which it is doing well by exhibiting behavior that is too random. So, by using the proposed approach, we transform the scheduler design into an optimization problem.

### 3.1 The Structure of the RL System

Very few studies have been performed using neuron-like elements capable of learning under reinforcement feedback, which is less informative than using supervised learning (by a teacher) [28]. One can also view the uncertainty inherent in using reinforcement learning as a result of a fundamental problem that faces any learning system, whether it is natural or artificial. That has been called the credit-assignment problem by artificial intelligence researchers [8]. This is the problem of determining what parts of a complex interacting or interlocking set of mechanisms, decisions, or actions deserve credit (blame) for improvements (decrements) in the overall performance of the system.

The credit-assignment problem is especially difficult when evaluative feedback to the learning system occurs infrequently, for example, upon the completion of a long series of decisions or actions. The current approaches to the credit-assignment problem in artificial intelligence largely rely on providing the critic with domain-specific knowledge [8]. In this work, we propose to use a method by which the learning system improves its own internal critic by an on-going learning process.

Contrary to the tradition of using a single element as the critic (a single automaton) in the learning system [2], [3], [4], this work employs a neural network that acts as a critic.

The different elements of a NN in this context can be viewed as “distributed” teams of learning automata which proved to be useful in solving a variety of decentralized problems, such as the routing of traffic in computer and communication systems [22], [23]. Therefore, the combined capabilities of these neuronal elements can be used to arrive at the correct decision as to which processor a task should be mapped.

The NN used in this work is based on the Backpropagation (BP) algorithm. The BP is one of the most popular methods used to train NNs [25] (Fig. 2); however, other NN algorithms (or stochastic automata) could be equally used.

The structure of the BP network consists of a set of inputs, followed by two or more layers of neurons. When inputs are applied to the BP network, each neuron computes a normalized weighted sum of its inputs and passes this result to the next layer of inputs and, eventually, the outputs. During learning, an error value is fed back through the outputs for the network to learn from. This error propagates backward through the network in a similar fashion to input values. The hidden layers act as feature detectors that develop during the learning process [9]. As a result, the BP neural network can be used as a general task scheduling mechanism [33].

BP networks do not require any knowledge of the mathematical function that they are required to solve. The design is also very flexible, with variations possible in many parameters. A major advantage of a BP network is that it typically converges with sufficiently low error when presented with distinct examples during training [9]. A general description of the BP and the conditions for its convergence can be found in [19], [30].

### 4 REINFORCEMENT-BASED SCHEDULING (RBS): THE ALGORITHM

The complete algorithm for the reinforcement-based scheduler (RBS) is given in Fig. 3. The general structure of the scheduler (Fig. 1) is replaced by the one in Fig. 3. Note that the two stages of the BP algorithm (the forward and the backward computations) do not follow each other, as is the case with most of the classical applications of the BP. This is due to the fact that a priori knowledge about which actions to take when the system is presented with a task is not available; therefore, there is no teacher that can show the RBS the correct decisions. The learning process in this case is

\[
U(t) = (u_1(t), \ldots, u_n(t)), \quad \text{where each } u_i(t) \text{ is a positive real number (e.g., size of the incoming task and precedence constraints), together with a real-valued payoff or reinforcement signal } e(t) \text{ (e.g., performance of the system at time } t). \quad \text{The RL system produces an output pattern } Y(t) = (y_1(t), \ldots, y_n(t)) \text{ (e.g., schedule), which is received by the environment.}
\]

Now, the problem that the RL system is designed to solve can be summarized as follows: Each vector \(U(t)\) provides information to the RL system about the condition or state of its environment at time \(t\). The vector \(U(t)\) is called the context input, or vector. Different actions, or output patterns, are appropriate in different contexts. As a consequence of performing an action in a particular context, the RL system receives from its environment, in the form of a payoff or reinforcement signal \(e(t)\), an evaluation of the appropriateness of that action in that context. The learning system’s role is to act on each context so as to maximize this payoff. Also, we assume that \(U(t)\) belongs to a finite set \(U(t) = (U^1, \ldots, U^k)\) of context vectors and that, to each \(U^i \in U\), there corresponds a payoff, or reinforcement function, \(e^5\).

In general, the learning system should satisfy several properties. The scheduler must learn to associate with each input pattern an output value for which the reinforcement signal it receives indicates the highest degree of success. In addition, it should be able to improve its performance in cases where it is doing poorly by using greater degrees of exploratory behavior, hence, discriminate between cases in which it’s doing poorly and those in which it’s doing well. This is important in order not to degrade its performance in cases in which it is doing well by exhibiting behavior that is too random. So, by using the proposed approach, we transform the scheduler design into an optimization problem.

### 3.1 The Structure of the RL System

Very few studies have been performed using neuron-like elements capable of learning under reinforcement feedback, which is less informative than using supervised learning (by a teacher) [28]. One can also view the uncertainty inherent in using reinforcement learning as a result of a fundamental problem that faces any learning system, whether it is natural or artificial. That has been called the credit-assignment problem by artificial intelligence researchers [8]. This is the problem of determining what parts of a complex interacting or interlocking set of mechanisms, decisions, or actions deserve credit (blame) for improvements (decrements) in the overall performance of the system.

The credit-assignment problem is especially difficult when evaluative feedback to the learning system occurs infrequently, for example, upon the completion of a long series of decisions or actions. The current approaches to the credit-assignment problem in artificial intelligence largely rely on providing the critic with domain-specific knowledge [8]. In this work, we propose to use a method by which the learning system improves its own internal critic by an on-going learning process.

Contrary to the tradition of using a single element as the critic (a single automaton) in the learning system [2], [3], [4], this work employs a neural network that acts as a critic.

The different elements of a NN in this context can be viewed as “distributed” teams of learning automata which proved to be useful in solving a variety of decentralized problems, such as the routing of traffic in computer and communication systems [22], [23]. Therefore, the combined capabilities of these neuronal elements can be used to arrive at the correct decision as to which processor a task should be mapped.

The NN used in this work is based on the Backpropagation (BP) algorithm. The BP is one of the most popular methods used to train NNs [25] (Fig. 2); however, other NN algorithms (or stochastic automata) could be equally used.

The structure of the BP network consists of a set of inputs, followed by two or more layers of neurons. When inputs are applied to the BP network, each neuron computes a normalized weighted sum of its inputs and passes this result to the next layer of inputs and, eventually, the outputs. During learning, an error value is fed back through the outputs for the network to learn from. This error propagates backward through the network in a similar fashion to input values. The hidden layers act as feature detectors that develop during the learning process [9]. As a result, the BP neural network can be used as a general task scheduling mechanism [33].

BP networks do not require any knowledge of the mathematical function that they are required to solve. The design is also very flexible, with variations possible in many parameters. A major advantage of a BP network is that it typically converges with sufficiently low error when presented with distinct examples during training [9]. A general description of the BP and the conditions for its convergence can be found in [19], [30].

### 4 REINFORCEMENT-BASED SCHEDULING (RBS): THE ALGORITHM

The complete algorithm for the reinforcement-based scheduler (RBS) is given in Fig. 3. The general structure of the scheduler (Fig. 1) is replaced by the one in Fig. 3. Note that the two stages of the BP algorithm (the forward and the backward computations) do not follow each other, as is the case with most of the classical applications of the BP. This is due to the fact that a priori knowledge about which actions to take when the system is presented with a task is not available; therefore, there is no teacher that can show the RBS the correct decisions. The learning process in this case is

\[
U(t) = (u_1(t), \ldots, u_n(t)), \quad \text{where each } u_i(t) \text{ is a positive real number (e.g., size of the incoming task and precedence constraints), together with a real-valued payoff or reinforcement signal } e(t) \text{ (e.g., performance of the system at time } t). \quad \text{The RL system produces an output pattern } Y(t) = (y_1(t), \ldots, y_n(t)) \text{ (e.g., schedule), which is received by the environment.}
\]
indirect since the RBS will learn about the quality of its actions by monitoring a performance metric that reflects how well is the system performing.

The RBS in Fig. 3 is an abstract machine that randomly selects actions according to some stored probability distribution and receives feedback from the environment evaluating those actions. The machine RBS then uses the feedback from the environment (parallel processor system) to update its distribution so as to increase the expectation of favorable evaluations for future actions. Such a machine can be viewed as a stochastic learning automaton.

Considering the task of learning a general input-output mapping using an automaton, it is clear that the automaton needs to consider inputs other than the reinforcement signal, which is called, as mentioned earlier, the context input. For automata with context input, the preferred action may change in different contexts.

Due to the complex nature of the scheduling problem, a further modification to the above structure is necessary. This will be detailed in the next section.

5 The Model

A generic representation of a parallel system was selected to conduct the experiments (Fig. 4). In this model, tasks arrive through a queue (Q) and get fed into a task allocation/scheduling mechanism which assigns each task to one of n processors.

The representation allows any set of tasks to arrive at the system and the scheduler may process those tasks in any manner. If the scheduler extracts all tasks from a known set before scheduling, the scheduler is deterministic. Alternatively, if the scheduler only considers a subset of the total number of tasks, the scheduler is stochastic. A single schedulable unit in this representation is called a task. A Task Generator (TG) supplies tasks to the scheduler. Processor $P_i$ handles tasks at arbitrary rates. Tasks are considered to be single schedulable units. Any partitioning of a larger program that may be necessary is assumed to have been explicitly defined by the programmer, or effected by some other partitioning system. The number of tasks to schedule is assumed to be much greater than the number of processors available and processing of tasks is nonpreemptive.

Each task can be considered as the set of parameters shown below, based on the “enhanced directed acyclic graph” [18]:

$$T_j = \{l_j, \prec_j, t_j\},$$

(1)

where $l_j$ is the length of task $j$, $\prec_j$ is the set of precedence relations between $T_j$ and other tasks, and $t_j$ is the creation time of $T_j$. The task generator may be any system which can provide a set of tasks to the scheduler. When considering a known problem, the task generator would be a set of known tasks along with other necessary information about them. For random task sets, the TG could be a system which generates tasks on the fly.

Chu et al. [7] presented a model that used a set of modules to be executed in place of the TG. The TG is simply an extension of this, which allows dynamic creation of tasks and time-stamping of tasks at scheduler entry (or task creation).

Each processor is considered to be independent of all others, i.e., there is no communication between tasks running on other processors. There is, however, a requirement that some control information be communicated between the processors so that task flow-dependencies are preserved. There is a total of $n$ available processors, each of which may have different processing capabilities. Each processor has an associated speed, $S_j$. For a single task $T_i$ executing on processor $P_j$, the time for completion is simply given by:

$$T(i, j) = \frac{l_j}{S_j} + W_{ij},$$

(2)

where $W_{ij}$ is a term which considers a processor being stalled while waiting for other tasks to complete, due to precedence requirements. If a task $T_i$, begins on processor $P_j$ at time $t_{\text{begin}}$, and the precedence relation $\prec_i$ is satisfied at time $t_{\text{permitted}}$, then $W_{ij}$ can be defined as:

$$W_{ij} = \max\{t_{\text{permitted}} - t_{\text{begin}}, 0\}.$$  

(3)

This arrangement of processors can be considered similar to an MIMD (multiple instruction streams multiple data streams) architecture, with a fast shared memory used to
maintain the flow-dependency information. Queues exist between each component of the above system so that tasks may be scheduled more rapidly than they can be processed. These queues may be of arbitrary depth, so that overflow never occurs, but are finite (as opposed to infinite queues as in Chu et al. [7]).

The simulation has been implemented using objects in C++ to ensure that modules can be interchanged easily. The simulator runs in discrete time, while the model is in continuous (real) time, so an ordering has been devised to ensure that the nature of the continuous-time system is not affected. The simulator allows changes in the many parameters, including cycles to be executed, number of processors, processor speed, number of dependencies (or complexity of the task graph—see Section 6), and some scheduler parameters.

Processors are considered as separate entities capable of independent processing, as in MIMD architectures. Queues have been created with a maximum of 100 elements. Tasks contain processing requirements, dependency information, and time-stamping.

The simulator is capable of generating a variety of metrics and other data for examination at the completion of a simulation. Metrics include processor load, queue length, cost function, and final completion time.

5.1 Scheduling Experiments
All schedulers are stochastic in nature, being able to “observe” only the current task at any time and relying on the RBS’s memory to track the current system state. A random scheduler was used as a benchmark for the RBSs. Three categories of RBSs were developed based on processor queues, loads, and throughput. Each of these scheduling systems is described below.

It was expected that successful RBSs should perform considerably better than the random scheduler, in the “formidable” cases. These were cases where the sets of conditions that made scheduling most difficult, i.e., a small number of processors with little power and a very complex task structure.

5.1.1 Random Scheduler
For any reasonable comparison between schedules, it is useful to have a reference to which they can be related. A random scheduler was chosen as the base reference.

The random system allocates each task to a random processor, irrespective of the task parameters. The random function used provides a (nearly) flat distribution, so that all processors share an equal number of tasks.

A flat distribution of tasks across the processors should provide reasonable load balancing on average. It was expected that this scheduler would behave reasonably, but that it would not be a particularly effective method once dependencies between tasks were considered.

5.1.2 Reinforcement-Based Schedulers (RBSs)
Common Structure and the History Queue. The RBSs examined were all based on the BP network described earlier. All of the networks were able to be created with one or more hidden layers. The schedulers have outputs, where is the number of processors in the system. These outputs are considered as a processor weighting, with the highest level output being the most favored destination processor. If the processor queue of the most favored destination was full, the next best choice was taken, until the task was either scheduled or held in the task queue waiting for a queue space to become available at any processor.

Errors were calculated from the system state after scheduling the task according to the action made by the RBS. The values used are based on a metric determined according to the scheduling algorithm used. These errors were immediately fed back into the RBS to improve the decision-making process.

In all of the following formulas, $n$ represents the number of processors and $err_i$ represents the error calculated for output $i$ of the RL system. This error corresponds to the error at processor $i$.

Fig. 5 also shows the additional components that were added to the scheduler of Fig. 4. The History Queue (HQ) shown in the figure is some form of long-term memory that compensates for the inadequacy of the short-term memory provided by the synaptic weights. This modification is
necessary since short-term memory can only work well for moderately difficult problems. For a complex problem such as scheduling, the HQ is needed to aid the RBSs to reach more accurate decisions.

At the outset of the run, the HQ is empty. When a task arrives, the scheduler assigns the task to a processor, after which the performance of the parallel processor system is measured through some predetermined criteria (described below). The size of the HQ is chosen with respect to the complexity of the problem (e.g., 50). It is important to note that the size of the HQ should not be made too large (e.g., thousands of elements) because, in the case of complex systems, the state of the system changes so rapidly that it makes information stored in the long-term memory obsolete in a short period of time. By maintaining a reasonably sized HQ, one can assure that all the information is relevant and can be used to guide the scheduling process more accurately.

When the tasks start to accumulate in the HQ, a better picture of the system’s behavior begins to emerge. For every task, the HQ holds the size of the task (\(q_i\)), which processor was the task assigned to (\(\text{decision } i\)), and the performance of the system after the decision to map the task was made (\(\text{critic } i\)). Now, when a new task arrives, the RBS goes through the previous iterations of (task, decision, critic) and updates its short-term memory (i.e., weight) at every iteration and, then, the new task is presented and a decision is made on to which processor the task should be mapped. The new task with the decision and the critic are added to the HQ. This process enables the RBS to search its long-term memory and check the decisions that were made in the last 50 iterations and, then, make a decision on how to schedule the new task.

When the size of the HQ reaches its predetermined limit (50 in this case) and a new task arrives and a decision gets made, the oldest triplet of (task, decision, critic) will be discarded and the new entry is added to the queue. The whole process is repeated again for every newly arriving task. The value of the “critic” is decided according to different criteria that can be used to measure how well the parallel processor system is performing. These are described below.

**Queue-Based Schedulers** Two schedulers were designed based on optimizing queue lengths and, thus, increasing throughput. The RBS structure for both of these schedulers was identical, with \(n\) inputs, \(h\) hidden layers, and \(n\) outputs. When considering queue lengths, the nature of the task itself was not considered—any task is simply one more added to the queue of any processor. Two different mechanisms were attempted using queues: averaging (or balancing) and minimization.

In queue averaging, the average queue length is determined and the error is calculated as below. Queue length for processor \(i\) is represented by \(q_i\) and average queue length by \(\overline{q}\).

\[
\overline{q} = \frac{1}{n} \sum_{i=1}^{n} q_i
\]

\[
err_q = \overline{q} - q_i,
\]

For queue minimization, the optimal queue length was considered to be zero, so that the error was calculated by,

\[
err_q = 0 - q_i
\]

The minimization error calculations will always be negative using this function, which will tend not to reinforce the learning system’s decision at any time. As a result, this was likely to not prove as effective as the first method.

**Load-Based Schedulers.** Another pair of RBSs were developed to focus on processor load. The processors were capable of tracking the total load processed for use in these algorithms. Total load (rather than average load) was required to ensure that effects of scheduling later in time would still have a noticeable effect on the RBS error calculations. When using averaging over time, the effect of any fixed change on the average reduces with time.

The two scheduling systems were based on load averaging (or load balancing) and load maximization. In the first case, load balancing, the idea is to attempt to use the optimum allocation method for independent scheduling. Load maximization is a simple method to attempt to achieve maximum processor utilization, another goal of many parallel systems. For load balancing, the average total load processed was calculated and errors determined from this. The total load processed by processor \(i\) is represented by \(l_i\), and the average total load by \(\overline{L}\) (calculated similarly to average queue length in (5)).

\[
err_l = \overline{L} - l_i
\]

For load maximization, the maximum load was considered to be 1.0, so that the error was calculated by,

\[
err_l = 1.0 - l_i
\]

This equation may suffer from similar faults to the queue minimization method, as the correction to every output will always be positive. This will tend to reinforce every output as correct.

**Throughput-Based Schedulers.** Two RBSs were devised to focus on processor throughput. These schedulers were striving for balanced throughput, representative of load balancing, and maximum throughput, one of the overall system goals. The equations used to calculate error for the two schedulers are similar to those outlined above for throughput, where \(t_i\) represent the processor throughput, and \(\overline{T}\) the average. For throughput averaging:

\[
err_t = \overline{T} - t_i
\]

For throughput maximization (which suffers from the same shortcomings as load maximization):

\[
err_t = 1.0 - t_i
\]

### 6 Examples

#### 6.1 Benchmarks

The schedulers are each tested on the system configurations given in Table 1. All processors are considered to be non-preemptive and homogeneous, having identical processing...
speeds. The number of tasks is much greater than the number of processors. These tests consider a system which is comprised of a small number of processors (five or 10—arbitrarily chosen) which are capable of executing completely independently of one another. Tasks running on a processor require no communication with other tasks, so that communication overhead can be ignored. The inclusion of dependencies means that these tasks are not necessarily independent, however.

This is similar to a small MIMD architecture, executing a set of data-independent tasks. The MIMD architecture consists of a set of processing elements (PE), each with its own processor and local memory. A control and communication network is used to program these processors and to supply them with data and access to external devices or systems. The processors can then function almost totally independently. Evaluating these schedulers on an MIMD architecture is most useful, since this is both a most flexible parallel architecture and arguably the most complex to handle efficiently.

In order to relate the maximum number of dependencies, as used in the simulator implementation, to a more common model, some analysis was performed on the task sets generated. Table 2 shows the relationship between the maximum number of dependencies and the number of edges in a corresponding task graph. Every graph tested consists of 2,000 nodes and range in complexity from simple (0) through to extremely complex (6).

It is important to note that these graphs are generated randomly, and the RBS does not have a priori information about the task graph under consideration. When an arriving task needs to be scheduled at some given instant in time, the RBS will have information about that particular task only (i.e., task size and number of precedence constraints), and not the whole problem. As mentioned earlier, the RBS will be also exposed to some information about the environment to show how good is the performance of the system at that particular instant of time.

From each of these tests, the scheduler performance was gauged according to a set of metrics, each of which is outlined below.

### 6.2 Cost Function

The cost function is determined by the system simulator and is the average time taken between task creation by the task generator and completion of a task by a processor (measured in cycles). Costs for each scheduler are then compared. This provides an accurate indication of system throughput. Note that, in this case, the cost function is not analytical but it evolves as the system performance changes [21].

### 6.3 Queue Lengths

Queue lengths for each processor are recorded and the average and median across the processors for each scheduler are calculated. Queue lengths are examined for the occurrence of queue overflow and the averages and medians for each processor are compared graphically. The queue length is a simple indication of load balancing between processors.

### 6.4 Processor Loads

The average processor load is calculated by the processor as follows:

$$\text{load} = \frac{\sum l_i}{n},$$

where $l_i$ is the load for each task completed by that processor, and $n$ is the total number of cycles completed. Processor load calculated this way provides more accurate information on processor load balancing. It is also useful as a metric for determining processor utilization, which is a common goal for scheduling algorithms.

### 6.5 Processor Throughput

Throughput (in tasks per cycle) is calculated by each processor as shown below, where $m$ is the number of tasks completed, and $n$ is the total number of processing cycles run.

$$\text{throughput} = \frac{m}{n}$$

This shows the throughput of each processor in the system, which the cost function does not, but is influenced more by the task sizes allocated to each processor (for example, a scheduler may learn to assign all small tasks to a single processor giving it a very high throughput).

### 6.6 Completion Time

The completion time is determined by allowing the system to continue running until all processor queues are empty (i.e., all tasks have been completed). Once all tasks have been allocated (after 2,000 cycles in this case), no other tasks are generated or scheduled, and the system should eventually complete all tasks. If deadlock occurs, this is detected and reported.

The completion time is the number of the cycle on which the last processor to finish completes its final task. This time does not include the time required for the scheduler to operate, but merely reflects the number of cycles required to complete the processing of all tasks.

### 7 Results

Altering some parameters caused noticeable and consistent changes across all types of schedulers. The most consistent
effects were caused by changing the number of hidden layers and by using dependencies (precedence constraints among tasks).

7.1 Number of Hidden Layers in RBSs

Increasing the number of hidden layers led to the deterioration of the performance of all RBSs. The main reason for this is probably due to the extremely long learning time required to train RBSs via the BP algorithm.

The effect of an extra hidden layer, in most cases, seemed to reduce the stability of the RBSs, as can be seen from Fig. 6. This graph shows the worst difference in loads between processors for a five processor system, using the most effective RBSs with one and two hidden layers. Note that, with two hidden layers, the scheduler is noticeably less effective at load balancing.

7.2 Effect of Dependencies

When the dependencies between tasks were considered, the performance of every scheduler was reduced, as expected. The effect is caused simply by the wasted cycles incurred during blocking. Evidence of this is given in Fig. 7, which illustrates the number of cycles wasted per processor for each scheduler, under the most difficult system conditions tested—five processors, each with a speed of 0.1, and up to six dependencies. As cycles are wasted, the effectiveness of any scheduler is bound to decrease.

The above does not indicate a weakness in the schedulers, however, as the inclusion of dependencies was expected to reduce performance. What this graph does show, though, is the ranking of schedulers in order of overall efficiency. The schedulers with the least percentage of wasted cycles, were the most effective overall, when considering all other metrics.

7.3 Effect of Overall Complexity (Number of Dependencies)

For the case of five processors with speeds of 0.1, the most formidable environment, increasing the number of dependencies, caused a noticeable deterioration in performance. Under other, more relaxed conditions, the effect of increased complexity on these systems was only slight.

In all cases, there is a considerable difference between task allocation (no dependencies) and scheduling (with dependencies) when considering the cost function. Fig. 8 shows a summary of the cost function, for all scheduler types, over the environments considered. Each bar represents the average cost for the final 20 cycles—at which point the cost function curve is generally leveling off.

Each set of four bars represents the system response with zero, two, four, and six dependencies, maximum. Notice that, for all schedulers, the cost for the independent system (first) in each set is considerably smaller than all other costs. There is not, however, a consistent increase in cost with increased complexity.

For the load and throughput based schedulers, the costs are roughly equal for any number of dependencies greater than zero. The random and queue-based schedulers, however, show a reduction in performance proportional to the maximum number of dependencies. Fig. 8 shows that this is evident only in the first set of results, which is the most formidable case—five processors with a speed of only 0.1.

The completion times for the schedulers are mostly unaffected by the number of dependencies. For the random and queue-based schedulers, however, there is generally an increase in expected completion times, proportional to the number of dependencies. The queue-minimizing scheduler shows dramatically reduced performance once there are six dependencies (maximum).
7.4 Schedulers’ Performance

7.4.1 Random Scheduler

The random scheduler was tested simply to act as a benchmark for assessing the performance of the RBSs. It proved to be a very effective process for scheduling. This could be tested by comparing the effectiveness of random scheduling with some common heuristic techniques.

In the most formidable cases, where processor speed was slow and the task-graph was very complex, the random scheduler caused a deadlock to occur. This only occurred with the random scheduler. It is also expected that the performance of such a scheduler will deteriorate when the number of processors increases; however, more work is required to determine the validity of this assumption.

7.4.2 Queue-Based Schedulers

The two queue-based schedulers proved to be the only effective RBSs developed. Both queue minimizing and queue balancing techniques were effective when dependencies were not considered, but the queue minimization method degraded considerably when they were. If dependencies were not considered, the performance of both queue-based schedulers was better than the random scheduler in the long term, under all metrics considered.

Once dependencies were considered, the performance of the queue balancing scheduler remained comparable with the random scheduler, and better in the long term in all metrics. The queue-minimizing scheduler, however, showed a severe drop in performance under these conditions. Although it still remained more effective than other RBSs, the queue-minimizing scheduler became considerably less effective than the queue-balancing or random schedulers.

Fig. 9 illustrates the queue-lengths of the random and queue-based schedulers with zero and six dependencies, for five processors with speed of 0.1 (most difficult case). This clearly demonstrates that the queue-balancing RBSs were more effective than the random scheduler in the long-term without dependencies. Initial performance cannot be seen clearly in this graph. It also demonstrates the dramatic reduction in performance of the queue-minimization scheduler once dependencies were included.

Fig. 9 shows the queue-lengths of each scheduler type, although similar results can be seen in all metrics except the cost function. The queue-balancing RBS was considerably more effective than any other scheduler at balancing queue lengths and processor loads. As a result, it was able to operate for the longest period without overflowing any of the processor queues.

Queue-based schedulers proved the most effective at maximizing processor utilization, being the only schedulers to provide greater than 90 percent utilization without dependencies, for processors with a speed of 0.1. The queue-balancing scheduler also provided the highest average processor utilization, with dependencies of nearly 75 percent.

With faster processors, the queue-balancing RBS was able to maintain queue lengths of two or less throughout its operation, without dependencies. Even with dependencies, this scheduler maintained the shortest processor queues and highest processor throughput of any scheduling system.

The completion time for the queue-balancing RBS is always the shortest. When independent tasks are considered, the queue-minimizing RBS is usually next fastest, followed by the random scheduler. Once dependencies are considered, however, the queue-minimizing scheduler once again fails.

Completion times for each scheduler, under all simulation environments, is shown in Fig. 10. Notice the random scheduler bars for four and six dependencies with five processors having speeds of 0.1. These two cases caused deadlock in the system, so completion time should really be infinite.

7.4.3 Load-Based Schedulers

There were three load-based RBSs attempted, none of which proved particularly effective. The first scheduler attempted was a load-balancing scheduler, as described earlier. This scheduler was not particularly effective at balancing loads, nor at any of the other metrics considered.

Scheduling using this method produces inconsistencies. Learning according to this method will reduce the likelihood of scheduling to a processor that has done much work, and increase the likelihood of scheduling to a processor that has done little.
This is conceptually incorrect, since a scheduler that has done little has probably done little because it is blocked, or has several very large tasks to handle—thus reducing its total processed load. In this case, the scheduler should actually stop scheduling to this processor, until the load improves.

A scheduler was briefly tested which used the reciprocal of the calculated error. That is, the scheduler would favor those processors which have the highest throughput, although preliminary testing proved so inadequate, that this was abandoned. The first scheduler to complete a task immediately had the highest throughput and was favored, so that its throughput continued to grow, and no other processors were allocated tasks. In this way, the imbalance between processors caused the scheduler to be highly ineffective.

The second main load-based RBS attempted to maximize the load at each processor, i.e., attempting to attain maximum processor utilization. This suffered from a similar failing to the first scheduler, in that processors with low completed load were favored, so that load would be maximized on all processors. Again, there are many cases when a processor performs slowly due to heavy loading and should not be allocated any further tasks—the opposite of this system.

Reversing this method, by using the negative of the error calculation, is bound to fail. In this case, the system will load up a single processor, as described above.

Load-based RBs are unable to work effectively, since they are based on unreliable information. Processed load does not provide a consistent representation of the state of the system (unlike the processor queue which reliably detects blocked, fast, and, slow processors, assuming allocation is initially random).

Possibly, using a load-based RBS that favors average-loads rather than the extremes, may prove more effective. There is, however, little intuitive reason as to why this would prove to be successful.

### 7.4.4 Throughput-Based Schedulers

The two throughput-based RBs failed for similar reasons to those described above for load-based RBs. Once again, they performed badly in all of the metrics considered.

By favoring those processors which show relatively or absolutely low throughput, any processor which is blocked or slowed by large tasks will be favored and swamped with work. Similarly, scheduling tasks to a processor which has been performing effectively will be discouraged. Reversing the system will once again cause the schedulers to favor a single processor as the best solution, which is obviously incorrect and highly ineffective.

### 7.5 Summary of Results

These results show that the load-based and throughput-based RBs are not effective in performing dynamic scheduling. The queue-based RBs are more successful, but the efficiency of the queue-minimization scheduler is highly dependent on the simulation environment, and the problem complexity.

Of all the schedulers tested, only two proved to be capable of providing good results in all situations. These were the random scheduler and the queue-balancing RBS. The queue balancing RBS generally performed better, and had the advantage of being able to schedule for a longer period before any queue overflow took place. The queue-balancing RBS also was shown to provide a better balance of work across processors, so that load balancing was improved and throughput (measured by each processor) was generally higher. Each scheduler and its performance are briefly outlined below:

**Random scheduler:** Capable of extremely efficient dynamic scheduling when the processors are relatively fast (speed = 0.2). Under more difficult conditions, its performance is significantly and disproportionately reduced.

**Queue-balancing scheduler:** The most effective scheduler developed.

**Queue-minimizing scheduler:** Performance is dramatically affected once there are up to six dependencies allowed. The change in performance is much more than when dependencies are initially introduced, or raised to smaller amounts. In other more "relaxed" system configurations, however, it is largely unaffected by dependencies.

**Load-based and throughput-based schedulers:** The final cost is notably affected by the introduction of dependencies between tasks, but is largely unaffected by the number of dependencies considered. Completion times are affected more by the processor speed than by other parameters. This can be explained by noting that these schedulers tend to allocate tasks to a single processor for a considerable time, and then switch to another processor, causing very heavy loading on a single processor at a time. When dependencies are considered this can cause a considerable number of wasted cycles, since the long queue in one processor may contain a task necessary for many other processors.

### 8 Conclusions

This paper introduced a new framework for scheduling in parallel processor environments, based on reinforcement learning techniques. Some of the methods introduced proved to be more effective than others; however, the reinforcement learning paradigm seems to hold great potential in capturing the complexity of the scheduling problem.

To improve the real-time applicability of RBs, one should exploit the parallelism inherent in the structure of the learning system (neural network in this work). Parallel implementations of neural networks can considerably speed up computations [11]. Other unorthodox techniques that might provide good solutions for the dynamic scheduling problem should also be investigated (e.g., genetic algorithms [15] and fuzzy logic [26]).

### Acknowledgments

The authors would like to thank the referees for their constructive suggestions and the associate editor, Professor Michael Palis, for his patience and help in obtaining the reviews and finalizing the decision on the status of the paper. Albert Y. Zomaya acknowledges the support of the...
Australian Research Concil Grant 04/15/412/194. Stephan Olariu acknowledges the support of U.S. National Science Foundation Grant CCR-9407180 and U.S. Office of Naval Research Grant N00014-1-95-0779.

REFERENCES


Albert Y. Zomaya received his PhD from Sheffield University, United Kingdom. He is a professor in the Department of Electrical and Electronic Engineering at the University of Western Australia, where he also leads the Parallel Computing Research Laboratory. The author/coauthor of more than 90 publications in technical journals, collaborative books, and conferences, he is currently an associate editor for IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Systems, Man, and Cybernetics, the International Journal in Computer Simulation, the International Journal on Parallel and Distributed Systems and Networks, and the Journal of Future Generation of Computer Systems. He previously served on the editorial boards of the IFAC Control Engineering Practice Journal and the Journal of Parallel Algorithms and Applications. He is also the founding editor of the Wiley Book Series on Parallel and Distributed Computing.

He is the author/coauthor of three books and the editor of two volumes. He is the editor-in-chief of the Parallel and Distributed Handbook (McGraw-Hill, 1996). Professor Zomaya is a board member of the International Federation of Automatic Control (IFAC) Committee on Algorithms and Architectures for Real-Time Control, and serves on the executive committee of the IEEE Technical Committee on Parallel Processing. He served on the steering and program committees for several national and international conferences. Professor Zomaya is founding cochair of the 1998 Workshop on Biologically Inspired Solutions to Parallel Processing Problems (BioSP3). He is a charter engineer and a senior member of the IEEE, a member of the ACM, the Institute of Electrical Engineering (U.K.), and Sigma Xi. His research interests are in the areas of parallel algorithms, scheduling, machine learning and robotics, scientific computing, and adaptive computing systems. He can be reached at: zomaya@ee.uwa.edu.au or http://www.ee.uwa.edu.au/staff/zomaya.a.html.
Matthew Clements graduated from the Department of Electrical and Electronic Engineering, the University of Western Australia, in 1995 with a Bachelor of Information Technology (with honors). Currently, he is employed as a technical consultant by Digital Equipment Corporation, where he implements solutions over a wide range of areas. This includes network operating system architecture review, project management, imaging systems, and enterprise management. He works mainly with government departments and the military.

Stephan Olariu received his MSc and PhD degrees in computer science from McGill University, Montreal, in 1983 and 1986, respectively. In 1986 he joined the Computer Science Department of Old Dominion University.

Dr. Olariu has published extensively in various journals, book chapters, and conference proceedings. His research interests include image processing and machine vision, parallel architectures, design and analysis of parallel algorithms, computational graph theory, computational geometry, and mobile computing.