Dynamic, Competitive Scheduling of Multiple DAGs in a Distributed Heterogeneous Environment

Michael Iverson and Füsun Özgüner

The Department of Electrical Engineering
The Ohio State University
Columbus, OH 43210
{*iverson,ozguner*}*@ee.eng.ohio-state.edu*

Abstract

With the advent of large scale heterogeneous environments, there is a need for matching and scheduling algorithms which can allow multiple DAG-structured applications to share the computational resources of the network. This paper presents a matching and scheduling framework where multiple applications compete for the computational resources on the network. In this environment, each application makes its own scheduling decisions. Thus, no centralized scheduling resource is required. Applications do not need direct knowledge of the other applications. The only knowledge of other applications arrives indirectly through load estimates (like queue lengths). This paper also presents algorithms for each portion of this scheduling framework. One of these algorithms is modification of a static scheduling algorithm, the DLS algorithm, first presented by Sih and Lee [1]. Other algorithms attempt to predict the future task arrivals by modeling the task arrivals as Poisson random processes. A series of simulations are presented to examine the performance of these algorithms in this environment. These simulations also compare the performance of this environment to a more conventional, single user environment.

Keywords: Matching and Scheduling, DAG, Multiuser, Poisson Random Process, List Scheduling.

1 Introduction

Heterogeneous computing has a number of distinct advantages [2, 3, 4], centering around the ability to utilize the features of different machine architectures. A central theme of heterogeneous computing is the ability to construct a single computational entity from a network of heterogeneous machines. As advanced networking technologies become available, the practical size of these heterogeneous environments is growing to a point where it is possible to create a single computational resource from a set of high performance computers distributed across the globe. In such a system, multiple users will be able to simultaneously utilize the computational resources of this network to execute a variety of large parallel applications. The primary challenge of using such a computing environment is to obtain a near-optimal solution to the matching and scheduling problem. To accomplish this task, there are several unique characteristics of this environment which must be considered: the dynamic nature of the machine and network loads, the size of the network, and the need for multiple users to fairly compete for the computational resources.

Given these issues, this paper presents a framework for executing multiple applications in this heterogeneous environment. These applications have a directed, acyclic graph (DAG) structure. In this framework, each application is responsible for scheduling its own tasks. Thus there is no centralized scheduling authority. This paper also presents a series of algorithms to operate within the framework. One of these algorithms is based upon a static matching and scheduling algorithm, called the DLS algorithm, first presented by Sih and Lee [1]. These algorithms attempt to predict the future loads of the machines, by modeling task arrivals as a Poisson random process. A series of simulations are presented to demonstrate these methods. In the next section, relevant background material is examined, and, in Section 3, an overview of the execution environment is presented. Section 4 gives a detailed presentation of the algorithms used within this environment. The results of a simulation study are discussed in Section 5, and conclusions from these results are offered in Section 6.

2 Background

The majority of the interest in DAG scheduling has been restricted to static environments. One simple and efficient type of heterogeneous scheduling method is the level-based algorithm, which schedules a task based upon that task's depth in the DAG. Some methods which fall into this category include those presented by Leangsuksun and Potter [5], who study a variety of simple, heterogeneous scheduling heuristics, and a method called the LMT algorithm, which assigns all of the tasks at a particular depth in the DAG at one time [6]. Kim and Browne [7] present a static scheduling technique called linear clustering, where tasks are clustered into chains of tasks, and these clusters are mapped onto the physical machines. This heterogeneous scheduling method has limited application to the proposed problem, since it assumes that the individual processors perform uniformly for all code types (i.e. the performance of a task on each heterogeneous processor varies only by a scale factor). A more complex static method, called the MH algorithm, is presented by El-Rewini and Lewis [8]. Again, this method is limited in that it uses the same simple model of a heterogeneous system as in [7]. Another method is the Cluster-M technique introduced by Eshaghian and Wu [9], which clusters tasks together based upon architectural compatibility. Of interest to this research is the method presented by Sih and Lee [1]. This static technique, called Dynamic Level Scheduling, schedules tasks by using a series of changing priorities. The DLS algorithm has been shown by Sih and Lee to be superior to other static DAG scheduling algorithms for heterogeneous systems, and will be discussed in more complete detail below.

While most of the DAG scheduling algorithms are static, there are a few algorithms that examine the problem of scheduling DAGs in a dynamic environment. For heterogeneous systems, Haddad [10, 11] presents a dynamic load balancing scheme for DAGs. This scheme differs from a conventional scheduling algorithm, in that it does not look at the exact structure of a given application. It instead uses a number of metrics that characterize the tasks and the task graph, to balance the computational load. For homogeneous MIMD systems, Rost et al. [12] present a scheduling model called agency scheduling. This model supports decentralized scheduling decisions by giving a set of distributed scheduling tasks control over a local set of processors. Neither of these methods explicitly consider the problem of scheduling multiple applications in a distributed environment. This paper presents a new dynamic scheduling method, which is designed for some of the unique features of this environment. In the next section, these features are examined in detail.

3 Definitions

As stated above, in this environment, multiple applications are competing for the computational resources of the network. Each application is represented by a set of communicating tasks. These tasks are organized using a DAG, $G = (V, E)$, where the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ represents the set of tasks to be executed, and the set of weighted, directed edges E represents communication between tasks. Thus, $e_{ij} = (v_i, v_j) \in E$ indicates communication from task v_i to v_j, and $|e_{ij}|$ represents the volume of data sent between these tasks. The execution environment consists of a set of heterogeneous machines, which can be represented by the set $M = \{m_1, m_2, \ldots, m_q\}$. The computation cost function, $C : V \times M \rightarrow \Re$, represents the execution cost of each task on each available machine. Thus, the cost of executing task v_i on machine m_j will be denoted by $C(v_i, m_j)$. If a particular task cannot be executed on a given machine, the function will evaluate to infinity.

In this environment, each machine is limited to executing one task at a time. There are several compelling reasons to adopt this organization.

1. In DAG scheduling, when a task is scheduled, it is desirable to know the time at which the task will complete execution. In a system where multiple tasks can simultaneously execute on a machine, the completion time of a given task depends upon the other tasks which are also executing on the machine. Since tasks from other applications may arrive at any time, it is not possible to determine the completion time *a priori*.

2. When system loads are able to change after a task has been assigned, task migration is necessary to balance machine loads. Task migration can be difficult in a heterogeneous environment, since there is no guarantee that there is another machine available to execute a given task.

To ensure that each machine can execute only one task at a time, each machine will have a FIFO queue. Tasks wanting to execute on a machine must wait in the machine's queue until the machine is available. However, it is possible for a task to receive data from predecessors while another task is executing, and likewise send data to successor tasks. Since such communication is not processor intensive, it should have little effect upon the execution of the running task.

The above organization does not include any resources for making scheduling decisions. Therefore, there will be another set machines in the network: scheduling machines. Each application will execute a scheduling task (on a scheduling machine), which is responsible for making all of the scheduling decisions for that application. Ideally, these scheduling machines would be general purpose workstations (possibly even the user's workstation). A conceptual model of this environment is shown in Figure 1.

Since each application is self-scheduling, an application only has direct knowledge of its own tasks. The

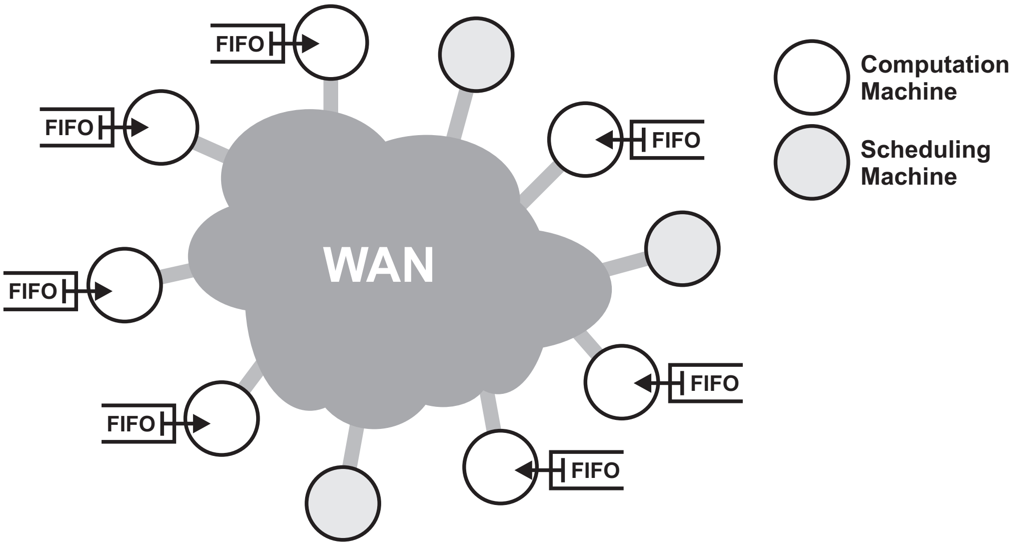

Figure 1: Conceptual model of execution environment.

only information it has about other applications is in the form of machine and network load estimates. The scheduling machines are responsible for maintaining this dynamic state information. The load of a particular machine m_j is characterized by the queue length Q_{m_j}, the task arrival rate λ_{m_j}, and the average size l_{m_j} of the arriving tasks, in terms of execution time. Although beyond the scope of this paper, techniques will be needed to acquire and estimate this loading data. One such method is presented by Hou and Shin [13], who use Bayesian decision theory to predict queue lengths using observations which may be out-of-date (due to the network latency).

The communication costs are represented using the function $D : E \times M \times M \rightarrow \Re$. Thus, the cost of sending the message from task v_r to task v_s (represented by edge e_{rs}) from machine m_i to machine m_j will be $D(e_{rs}, m_i, m_j)$. Communication between tasks complicates the matching and scheduling process in this environment. In order for a task to begin execution, two conditions must be satisfied: (1) data from previous tasks must be available, and (2) the task must be at the head of the queue. Ideally, both of these conditions would be satisfied at the same time. Achieving this goal is not likely, however. Thus, there are two possible scenarios:

1. *Data is available before the task reaches the head of the queue.* In this case, the task will have to wait to begin execution (the task should have been placed in the queue earlier). This queuing delay can potentially increase the completion time of the application. Other applications are unaffected.

2. *Task reaches the head of the queue before data is available.* In this case, the task was placed in the queue too early, and the task will have to wait for the data. As it waits, the machine will be idle, and the other tasks in the queue will have their progress blocked.

There is a significant problem with having a task block the queue. When a task is allowed to block the queue for an arbitrary amount of time, it is no longer possible to accurately determine the completion time of a task when it is placed in the queue. However, there is a means of limiting the effects of this behavior. It is possible to require the scheduling algorithm to estimate the amount of blocking time that a task is likely to incur, based upon the queue length and the time at which the data is expected to arrive. This estimate can then be explicitly included as part of the queue length estimates, thus minimizing the effects of any blocking time. Even with this modification, blocking time still represents a waste of machine resources.

Some additional definitions are needed by the matching and scheduling algorithms defined below. The *static level* of task v_i, $L_{\text{static}}(v_i)$, is defined to be the largest sum of the median execution times of the tasks along any directed path from task v_i to an end node of the graph. Since the environment is heterogeneous, the median execution time of a task v_i, denoted $\bar{C}(v_i)$, is used to characterize the overall behavior of that task. If the actual median is infinite, the median value will be replaced with the largest finite execution time. In order to differentiate between different machines, the term

$$\Delta(v_i, m_j) = \bar{C}(v_i) - C(v_i, m_j) \tag{1}$$

is defined to indicate the speed that machine m_j executes task v_i, relative to the median value. A large value of Δ implies a fast machine. Given this set of definitions, the details of matching and scheduling in this environment will be presented in the next section.

4 Matching and Scheduling Algorithm

In this section, we will define a framework for solving the matching and scheduling problem in the environment defined above, and then present specific algorithms for each portion of the framework. The framework will be defined as a series of sets containing the tasks of the application, and a series of heuristics to move tasks between these sets. The first set will be the set of unscheduled tasks, denoted U. As its name implies, the set U will contain tasks which have not been scheduled to execute on a particular machine. At the beginning of the algorithm, all tasks are members of the set U.

In list scheduling methods, no task can be scheduled until all of its predecessors have been scheduled. Using this ordering, we can define the set of ready tasks R to be the set of tasks whose predecessors have been scheduled, and thus can be assigned to a machine. From the definition above, it is clear that $R \subset U$. Below, a heuristic will be defined that will choose the best-suited task (and the machine to execute it on) from this set of ready tasks R. This will be called the *matching decision policy*. Since much of the information used by the matching decision policy is dynamic in nature, it is important to consider the time at which the matching and scheduling decisions are made. This time will be determined by a heuristic called the *scheduling time policy*. The combination of these two policies will move tasks from set R to a new set P: the set of pending tasks. The set of pending tasks P is the set of tasks which have been assigned to execute on a particular machine, but have yet to be placed in the appropriate machine's queue. This set is partitioned into q subsets $P_1, P_2, \ldots, P_q$. Each subset P_j contains the tasks which have been assigned to machine m_j. When a task has been placed in the queue, it is moved into a final set S, the set of scheduled tasks. The time at which a task is placed in the appropriate queue is determined by a heuristic called the *queuing time policy*. Figure 2 illustrates the sets defined above, and the flow of tasks from one set to another.

Given these definitions, the process of making a decision within this framework can be broken down into three steps: determining how to make a matching and scheduling decision (matching decision policy), determining when to make a matching and scheduling decision (scheduling time policy), and determining when to place a scheduled task in the chosen queue (queuing time policy). Figure 3 shows the algorithm which governs how each of these steps is executed. Every T time units, this algorithm will use the policies discussed above to move tasks between the various sets. The value of T is called the *examination interval*. In the subsections below, algorithms will be presented for each of these policies. The matching decision policy will be a dynamic adaptation of the DLS algorithm presented by Sih and Lee [1]. The scheduling time and queuing time policies will use probabilistic methods to determine the appropriate time to schedule or queue tasks.

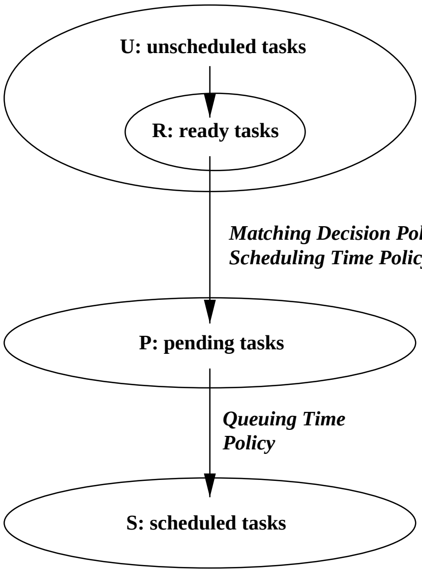

Figure 2: Dynamic DAG Scheduling Framework.

4.1 Matching Decision Policy

As mentioned above, the matching decision algorithm is based upon a dynamic adaptation of the DLS algorithm. Therefore, a brief overview of the relevant portions of the original static version of the DLS algorithm will be presented first.

4.1.1 Static DLS Algorithm

In [1], Sih and Lee present the DLS algorithm: a compile time, static algorithm for scheduling a DAG onto a set of heterogeneous machines. This algorithm, can be categorized as a list scheduling algorithm, where the tasks are assigned to the machines in topological order. As with nearly all list scheduling algorithms, the DLS algorithm operates by assigning a priority, called a level, to each task in that graph. This priority is then used to choose among the set of tasks which are ready to be scheduled at that time. The the DLS algorithm differs from previous algorithms in that the level of a task depends upon the tasks which have already

```
While $U \neq \phi, do$ :
 begin
    While the scheduling time policy indicates
       that tasks should be scheduled:
    begin
          Use matching decision policy to
             choose a task-machine pair
             (choose a task to move from
             set $R$ to set $P$).
          Check precedence constraints of
             tasks
             (find any tasks which can be
             moved into set $R$).
    end
    Use queuing time policy to examine
       pending tasks
       (move tasks ready to be queued from
       set $P$ to set $S$).
    Wait $T$ time units.
 end
```

Figure 3: Generic matching and scheduling algorithm for the framework

been assigned. This concept is called the *dynamic level* of a task, and is defined to be

$$\begin{aligned} L_{\text{dynamic}}(v_i, m_j) = \\ & L_{\text{static}}(v_i) - \max[t_{\text{data}}(v_i, m_j), t_{\text{free}}(m_j)] \\ & +\Delta(v_i, m_j), \end{aligned} \tag{2}$$

where $t_{\text{free}}(m_j)$ denotes the time at which machine m_j will be idle, and $t_{\text{data}}(v_i, m_j)$ denotes the time when task v_i's data will be available on machine m_j. The first term of the expression is the static level of the task. This term indicates the path length to the end of the graph. Since a long path is more likely to be the critical path of the graph, a large value of L_{static} will increase the scheduling priority. The second term indicates when the task can begin on the machine, based upon the time when the data is available on the machine, and the time at which the machine is able to execute the task. An earlier starting time will imply a higher priority. The third term indicates how fast the machine m_j will execute the task, relative to the other machines in the system. With this dynamic level expression, making a matching decision is equivalent to finding the ready task and machine which maximize the above expression. Another advantage to this approach is that both the task and processor are chosen at the same time. Sih and Lee show that this policy is superior to independently selecting either the task or the processor.

4.1.2 Dynamic Matching Decision Policy

As mentioned above, the purpose of the matching decision policy, given a set R of tasks ready to be scheduled, is to find the "best" task-machine pair from the set of ready tasks R. This decision policy can be constructed using a modified version of the dynamic level equation presented above.

In order to operate in a dynamic environment, the terms of the dynamic level expression, shown in equation 2, will require modification. Examining the individual terms of this expression, it can be seen that the first and last terms do not depend upon any data which is dynamic in nature. However, the middle term, $\max[t_{\text{data}}(v_i, m_j), t_{\text{free}}(m_j)]$, which denotes when the task will be able to begin execution on the specified machine, does depend upon dynamic information. The two arguments of the max operator represent the two independent events which need to be satisfied in order for a task to begin executing on a machine. For the environment discussed above, the two terms needed are the time at which the data is available on the chosen machine and the time at which the machine is idle, and thus able to execute the task. For this environment, this second quantity is defined to be

$$t_{\text{free}}(m_j) = t + Q_{m_j} + \sum_{\forall v_k \in P_j} C(v_k, m_j), \tag{3}$$

where t is the current time. The second term in the expression is the execution time of the tasks in machine m_j's queue at this time, and the last term represents the total execution time of the tasks in set P_j (i.e. waiting to be placed in machine m_j's queue). With this modification, this portion of the DLS algorithm can be used as the matching decision policy. The next issue of interest is the process of determining when to make a scheduling decision.

4.2 Scheduling Time Policy

As shown by Sih and Lee [1], it is better to choose the task and machine simultaneously, rather than choose either independently of the other. Thus, the purpose of the scheduling time policy is to determine when it is appropriate to make matching and scheduling decisions, not when to schedule a particular task. There is a tradeoff inherent in deciding when to schedule a task. Since the loading information used by matching decision policy will change with time, if a task is scheduled early, the information used to make the decision could be too inaccurate to be of use. However, if the algorithm waits too long to schedule the task, it is possible that a desired machine will be unavailable (due to a long queue) and the task will be forced to execute on a suboptimal machine.

In this paper, the following heuristic is proposed to determine when scheduling decisions should be made. A scheduling decision will be made if the probability that any ready task will experience queuing delay on its most desired machines exceeds a predefined threshold value. To simplify the derivation of this probability, consider a single machine m_j. Using the above information, we can derive an expression defining the probability that a task will experience queuing delay on this machine if it is not assigned to the machine now (if it is not assigned now, it would have to wait a minimum of T time units until the scheduler examines the situation again). In other words, we would like to find the probability that a sufficient number of tasks from other applications will arrive (and be placed in the queue) in the next T time units such that the task will be forced to experience queuing delay.

To determine an expression for this probability, it is necessary to define a "critical number" of tasks—the minimum number of average-sized tasks which would have to arrive within the examination interval T, such that any task assigned at time $(T + t)$ would experience queuing delay. Assuming that the arriving tasks are of average size l_{m_j}, the critical number will lie within the interval

$$\left(\left\lceil \frac{t_{\text{slack}}}{l_{m_j}} \right\rceil , \left\lceil \frac{T + t_{\text{slack}}}{l_{m_j}} \right\rceil \right) , \tag{4}$$

where t_{slack} denotes the difference between the time at which the data will be available and the time at which the queue will be empty. This term, called the slack time, is defined to be

$$t_{\text{slack}} = \left\{ \begin{array}{ll} t_{\text{data}} - (Q_{m_j} + t) & \text{if } t_{\text{data}} > (Q_{m_j} + t) \\ 0 & \text{otherwise} \end{array} \right. . \tag{5}$$

While it is more likely that the critical number will be closer to the upper bound of this interval, it is better to use the lower bound, due to the behavior of the method in the boundary condition (which will be explained below). Therefore, in this paper, the critical number of tasks z will be defined to be

$$z = \left\lceil \frac{t_{\text{slack}}}{l_{m_j}} \right\rceil . \tag{6}$$

Now, assuming that the task arrivals can be reasonably approximated using a Poisson random variable, the probability that the number of arrivals k within the interval T will be greater than or equal to z can be defined to be

$$\begin{aligned} P_{m_j}[k \geq z] &= 1 - P[k < z] \\ &= 1 - \sum_{k=0}^{z-1} \frac{(\lambda_{m_j} T)^k}{k!} e^{-\lambda_{m_j} T} . \end{aligned} \tag{7}$$

The reason for choosing to use the lower boundary in equation 5 is due to the case when t_{slack} is equal to zero. By defining the quantity z in this manner, equation 7 will evaluate to one.

This expression only considers the probability of experiencing queuing delay on a single machine. In reality, it is likely that there will be more than one machine available to execute the task. It is therefore desirable to expand the above expression to consider the possibility of a task experiencing queuing delay on more than one of its best performing machines. Thus, the scheduling time policy will be defined to schedule tasks if there is a ready task which has a probability of experiencing queuing delay on its *three* fastest machines that is greater than a predefined threshold β. So, to determine if a task is "in danger" of not getting a desired machine, the algorithm finds the machines m_i, m_j, and m_k on which the task executes the fastest. Then, the algorithm computes the critical number z for each of the above machines, (denoted z_{m_i}, z_{m_j}, and z_{m_k}) and the probability of experiencing queuing delay on each of these machines, using equation 7 above. With these values, the overall probability of not getting one of these three machines is

$$\begin{aligned} P_{\text{queue}} = {} & (P_{m_i}[k \geq z_{m_i}]) \cdot \\ & (P_{m_j}[k \geq z_{m_j}]) \cdot (P_{m_k}[k \geq z_{m_k}]). \end{aligned} \tag{8}$$

The choice of three is clearly a heuristic. The advantage of using a fixed number, like three, is primarily computational: the algorithm will be more efficient, which is important in a dynamic environment. If there are fewer than three machines on which a task can execute, the probability will be computed using this lesser number of machines. The choice of the value β is also a heuristic. A series of experiments are performed to evaluate the choice of a value for the parameter β. These results will be presented in Section 5.

4.3 Queuing Time Policy

As discussed above, when placing a task in a queue, there are two possible scenarios: the task is placed in the queue too early and experiences blocking delay, or the task is placed in the queue too late, and experiences queuing delay. The ideal time to place a task in the queue lies between these two extremes. The formulation of the queuing time policy will use a probabilistic formulation similar to the scheduling time policy described above. To construct a heuristic to attempt to place a task in the queue at the appropriate time, a pair of cost functions will be defined. The first cost function, C_{block}, will indicate the blocking cost the task will experience if it is placed in the queue now. The second cost function, C_{queue}, will indicate the probable queuing cost the task will experience if the queuing algorithm waits another T time units to assign the task. The goal of the queuing time policy is to place each task in the queue in a manner which minimizes both the queuing and blocking cost.

The blocking cost function is defined to be the amount of blocking time the machine will experience. Given the time at which the data is available, t_{data}, and the queue length at time t, Q_{m_j}, the blocking cost is

$$C_{\text{block}} = t_{\text{data}} - (Q_{m_j} - t) = t_{\text{slack}}. \tag{9}$$

This value is equal to the t_{slack} term defined above.

The definition of the queuing cost function is more elaborate. While the blocking cost is a deterministic quantity (provided that t_{data} and Q_{m_j} are accurate) the queuing cost is stochastic, in that it will depend upon the probability of future arrivals in the queue. Like the probable queuing cost definition from the scheduling time policy, this cost function will depend upon a critical number z. This number represents the minimum number of averaged-sized tasks which have to arrive in order for the task to experience queuing time. As before, z is defined to be

$$z = \left\lceil \frac{t_{\text{slack}}}{l_{m_j}} \right\rceil. \tag{10}$$

However, unlike the scheduling time case, we are not interested in the probability of experiencing queuing delay, but the probable amount of queuing delay. To determine this quantity, first consider a simpler task of finding how much queuing time a task would experience if exactly k tasks were to arrive in the time interval T. In this case, the queuing cost is

$$t_{\text{queue}}(k) = \left\{ \begin{array}{ll} kl_{m_j} - (t_{\text{slack}} + T) & \text{if } kl_{m_j} > (t_{\text{slack}} + T) \\ 0 & \text{otherwise} \end{array} \right. . \tag{11}$$

This expression can then be used to determine the probable queuing cost, by multiplying by the discrete probability of exactly k arrivals, and summing over all possible values of k:

$$\begin{aligned} C_{\text{queue}} &= \sum_{k=0}^{\infty} t_{\text{queue}}(k) P[n = k] \\ &= \sum_{k=z}^{\infty} (kl_{m_j} - (t_{\text{slack}} + T)) P[n = k] \\ &= \sum_{k=z}^{\infty} (kl_{m_j} - (t_{\text{slack}} + T)) \frac{(\lambda_{m_j} T)^k}{k!} e^{-\lambda_{m_j} T}. \end{aligned} \tag{12}$$

Since it is undesirable to compute an infinite summation, the expression can be rearranged to become

$$\begin{aligned} C_{\text{queue}} &= (kl_{m_j} - (t_{\text{slack}} + T)) \\ &\quad - \sum_{k=0}^{z-1} (kl_{m_j} - (t_{\text{slack}} + T)) \frac{(\lambda_{m_j} T)^k}{k!} e^{-\lambda_{m_j} T}. \end{aligned} \tag{13}$$

Now, given these two cost functions C_{queue} and C_{block}, the queuing time policy will place a task in its queue when the blocking cost is greater than the queuing cost. However, as mentioned previously, the queuing cost and blocking cost may not have the same effect upon the system. Therefore, an additional parameter γ will be introduced, in order to modify the relative weight of the two cost functions. Thus, to decide whether or not to queue a particular task, the algorithm will compute the quantity $C_{\text{queue}} - \gamma C_{\text{block}}$. Every T time units, the algorithm will compute this quantity for each of these pending tasks. If, for a given task, the quantity is negative, the machine will not place the task in the queue at this time. Otherwise, if the quantity is positive, the task is placed in the queue. As was the case for the scheduling time policy, the choice of the parameter γ will be examined experimentally in Sections 5.

5 Results

To evaluate these methods, a series of simulations were performed, using a custom, event-based simulator. These simulations examine the effects of the parameters β and γ, and compare the performance of the algorithms to the static DLS algorithm, which uses a more conventional environment where each user has exclusive use of the machines for a period of time. A representative set of results are shown in the figures 4 and 5. In this case, eight 64-task applications are scheduled on a 16 machine heterogeneous system. The execution times, task graphs, and computation costs were randomly generated, and it is possible for a task not to be able to execute on every machine. The graphs were generated such that they were capable of using about 8 machines in parallel.

The starting time of each of the applications was chosen over a random interval between 0 and 200 time units, to limit any artificial effects from starting all the applications at the same time. The examination interval T was chosen to be 1 time unit. The results shown in the graphs are an average of 5 separate simulations, to minimize any effects caused by specific graph structures. For each simulation, the schedule length of the applications was recorded, and the efficiency of the computation was determined. The efficiency measures the amount of blocking time in the system, relative to the amount of computation. For example, an efficiency of 0.75 would indicate that 75% of the total CPU time used was useful computation, and the remaining 25% was blocking time.

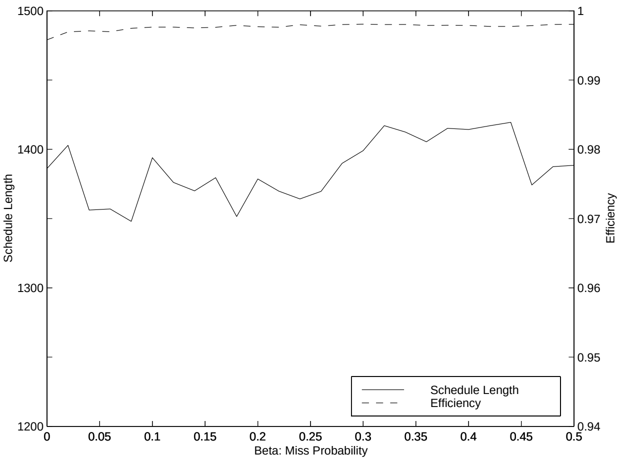

Figure 4: Average Schedule Length and Efficiency vs. Miss Probability (β).

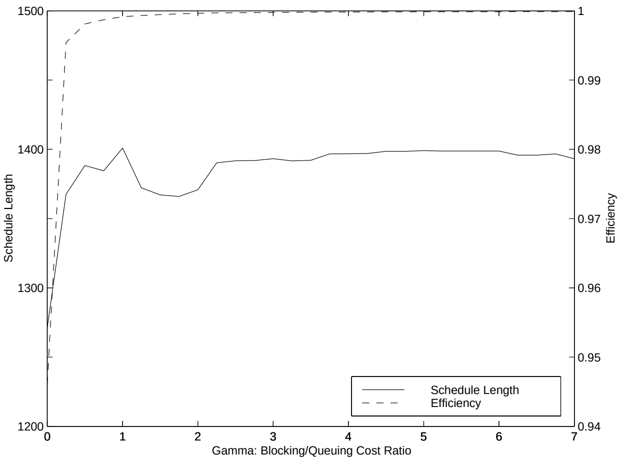

Figure 5: Average Schedule Length and Efficiency vs. Blocking/Queuing Cost Ratio (γ).

Figure 4 shows the schedule length and efficiency versus the parameter β (probability of missing the three fastest resources). These values are averaged over all of the values of γ. Overall, the parameter β has a relatively small effect upon the schedule length, which is likely due to the good quality of the loading information presented to the algorithm. However, for larger values of β, the schedule length tends to be long, since there is a higher probability that a task will have to execute on a suboptimal processor. Likewise, for very small values of β, the schedule length is also long, due to the fact that the scheduling decision was made with less accurate loading information. The efficiency is more or less constant with respect to β, since this parameter has no real effect upon the blocking time of the system.

Figure 5 shows the other case: the schedule length and efficiency versus the parameter γ (queuing/blocking cost ratio), averaged over all of the values of β. These results show the negative effect of blocking time upon the system. Using small values of γ, it is possible to get lower schedule lengths by placing the tasks in the queue early, and incurring blocking time. However, this has an adverse effect upon the efficiency, and, for slightly larger values of γ, tends to have a negative effect upon the schedule length (since processor resources are wasted). For simulations with more applications, the blocking time has an even greater impact upon the schedule length. For larger values of γ, there is a distinct minimum in the graph, representing the best trade-off between blocking time and queuing time (for this simulation). At this point, the best schedule lengths can be obtained without incurring large amounts of blocking time.

It is also desirable to compare the performance of this method to a static scheduling paradigm, where each application has exclusive use of all (or a portion) of the machines in the network. To accomplish this, each task graph was also scheduled using the static DLS algorithm. Using this data, the speedup was computed to be the total time needed to execute all eight applications sequentially, divided by the total time needed to execute all eight applications in the dynamic environment. Given that each application used in this experiment is, on average, capable of utilizing eight machines, the closest comparison of these two environments is for an eight machine system. In this case, the maximum speedup over all parameter values was found to be 1.21, a 21% improvement over the static environment. The speedup in the 16 machine system is considerably higher, since, on average, half of the machines will remain idle in the static environment (where all 16 machines are dedicated to the application). In this case, the maximum speedup was found to be 2.36. As expected, the dynamic scheduling method outperforms the static method, since it allows other applications to use computational resources which would be left idle in a static scheduling paradigm.

6 Conclusions

In this paper, a means of competitively scheduling multiple DAG-structured applications in a distributed heterogeneous environment is presented. Initial results show that this type of scheduling is practical, and confirm the assumptions made about the behavior of the scheduling environment. Currently, the authors are working on implementing these algorithms on an actual distributed network, to better evaluate and refine the techniques presented here.

References

[1] G. C. Sih and E. A. Lee, “A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures,” *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 175–187, Feb. 1993.

[2] J. B. Andrews and C. D. Polychronopoulos, “An analytical approach to performance/cost modeling of parallel computers,” *J. Parallel Distributed Computing*, vol. 12, pp. 345–356, 1991.

[3] R. F. Freund and H. J. Siegel, “Heterogeneous processing,” *IEEE Computer*, vol. 26, pp. 13–17, June 1993.

[4] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, “Heterogeneous supercomputing: Problems and issues,” in *Proc. of the 1992 Workshop on Heterogeneous Processing*, pp. 3–12, IEEE Computer Society Press, Mar. 1992.

[5] C. Leangsuksun and J. Potter, “Designs and experiments on heterogeneous mapping heuristics,” in *Proc. of the 1994 Heterogeneous Computing Workshop*, (Cancún, Mexico), pp. 17–22, IEEE Computer Society Press, Apr. 1994.

[6] M. A. Iverson, F. Özgüner, and G. Follen, “Parallelizing existing applications in a distributed heterogeneous environment,” in *Proc. of the 1995 Heterogeneous Computing Workshop*, (Santa Barbara, CA), pp. 93–100, IEEE Computer Society Press, Apr. 1995.

[7] S. J. Kim and J. C. Browne, “A general approach to mapping parallel computations upon multiprocessor architectures,” in *the 1988 Inter. Conf. on Parallel Processing*, vol. 3, pp. 1–8, CRC Press, 1988.

[8] H. El-Rewini and T. G. Lewis, “Scheduling parallel program tasks onto arbitrary target machines,” *J. Parallel Distributed Computing*, vol. 9, pp. 138–153, 1990.

[9] M. M. Eshaghian and Y.-C. Wu, “Mapping and resource estimation in network heterogeneous computing,” in *Heterogeneous Computing* (M. M. Eshaghian, ed.), pp. 197–223, Artech House, 1996.

[10] E. Haddad, “Load distribution optimization in heterogeneous multiple processor systems,” in *Proc. of the 1993 Workshop on Heterogeneous Processing*, pp. 42–47, IEEE Computer Society Press, 1993.

[11] E. Haddad, “Dynamic optimization of load distribution in heterogeneous systems,” in *Proc. of the 1994 Heterogeneous Computing Workshop*, (Cancún, Mexico), pp. 29–34, IEEE Computer Society Press, Apr. 1994.

[12] J. Rost, F.-J. Markus, and L. Yan-Hua, “Agency scheduling: A model for dynamic task scheduling,” in *Proc. of the 1st Inter. EURO-PAR Conf.*, (Stockholm), pp. 93–100, Springer-Verlag: Lecture Notes in Computer Science, Aug. 1995.

[13] C.-J. Hou and K. G. Shin, “Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems,” *IEEE Trans. Computers*, vol. 43, pp. 1076–90, Sept. 1994.

Michael Iverson received the B.S. degree in Computer Engineering at Michigan State University in 1992, and the M.S. degree in Electrical Engineering at The Ohio State University in 1994. He is currently researching topics in heterogeneous distributed computing for his Ph.D. dissertation. In addition to his research, Mr. Iverson is building Internet video conferencing systems and wireless networking systems for University Technology Services at Ohio State.

Füsun Özgüner received the M.S. degree in electrical engineering from the Istanbul Technical University in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975. She worked at the I.B.M. T.J. Watson Research Center with the Design Automation group for one year and joined the faculty at the Department of Electrical Engineering, Istanbul Technical University in 1976. Since January 1981 she has been with The Ohio State University, where she is presently a Professor of Electrical Engineering. Her current research interests are parallel and fault-tolerant architectures, heterogeneous computing, reconfiguration and communication in parallel architectures, real-time parallel computing and parallel algorithm design. She has served as an associate editor of the IEEE Transactions on Computers and is the Program Vice-Chair for Fault Tolerance and Reliability for the 1998 International Conference on Parallel Processing.