Centralized and Distributed Dynamic Scheduling for Adaptive, Parallel Algorithms

Sharon L. Smith and Robert B. Schnabel

CU-CS-516-91 February 1991

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado, 80309 USA

This research was supported by NSF grant CDA-8922510, AFOSR grant AFOSR-90-0109, and ARO grant DAAL 03-88-K-0086.

Abstract

We examine a set of dynamic scheduling techniques for parallel adaptive algorithms in a distributed computational environment. We consider three basic scheduling approaches: centralized scheduling, which uses a master-slave model of computation; distributed scheduling, which uses local information about processor workload to determine when tasks should be requested from or sent to other processors; and a new approach that we refer to as centralized mediation, that uses aspects of both centralized and distributed scheduling. We use both distributed implementation and simulation to examine the performance and scalability of these three scheduling approaches when applied to a parallel adaptive algorithm for solving the global optimization problem. In these experiments, the new centralized mediation approach appears to provide the best combination of robustness, efficiency, and ease of implementation.

1 Introduction

There has been much interest in the computational science community in algorithms that are *adaptive* in nature. Adaptive algorithms attempt to identify the parts of a computation that should receive the most attention, and then dedicate the appropriate amount of resources to those parts. A parallel adaptive algorithm may accomplish this by creating or removing tasks dynamically, or by varying the workloads of individual tasks. Because the number and size of tasks can vary during a parallel computation, robust scheduling techniques are needed to ensure that the workload is evenly distributed among the processors.

One approach to scheduling tasks in an adaptive parallel algorithm is to take periodic "snapshots" of the distribution of workload among processors and then determine if the entire workload should be redistributed. This approach is used in [10] and [1] for irregular mesh problems, and in [9] for solving adaptive multigrid problems.

In contrast, the approach taken in this paper is to continuously re-distribute work using dynamic task distribution. We present a set of dynamic scheduling techniques for parallel adaptive algorithms in a distributed computational environment. We consider three basic scheduling approaches: centralized scheduling, using a master-slave model of computation; distributed scheduling, using local information about processor workload to determine when tasks should be requested from or sent to other processors; and an approach we refer to as centralized mediation, that uses aspects of both centralized and distributed scheduling. This paper examines the performance and scalability of these three approaches. Our first concern is how well each of the strategies addresses the problem of dynamic task distribution for parallel adaptive computations. In particular, the success of a strategy is evaluated with respect to processor utilization and overall computation execution time. Our second concern is how the scheduling algorithms scale. We will show how various factors such as the number of processors or the size of tasks affects the performance of each of the three scheduling strategies.

In order to evaluate the performance of the various scheduling techniques, we use a parallel adaptive algorithm for solving the global optimization problem. The principal results from this performance study are obtained by simulating the various scheduling disciplines as they execute this parallel adaptive algorithm, although some of the simulations have been carefully validated by comparison with parallel implementations.

The remainder of the paper is organized as follows. In Section 2 we describe the three scheduling strategies. In Section 3 we discuss how each scheduling strategy is used to implement the parallel adaptive algorithm for solving the global optimization problem. In Section 4 we describe the experi-

mental strategy used to evaluate the scheduling approaches and in Section 5 we present results from simulation experiments.

2 The scheduling algorithms

There are several issues that arise in load balancing that need to be addressed by a scheduling strategy. The first issue is how to decide when to move a task from one processor to another, and which task to move. A second issue is how to decide which processor to send tasks to or to get tasks from. For the scheduling strategies discussed in this paper, a *task transfer policy* describes how the first decision is made, and a *task location policy* describes how the second decision is made. (The distinction between these policies will become clearer in the examples of Section 2.1.) A third issue concerns the type of information used to make these decisions. A policy based on global information would use information about the state of each processor to make decisions, whereas a local policy would use only information about a processor's local state. A non-local policy might use local information and non-local information from neighboring processors to make a decision. A final issue is whether a scheduling strategy has a centralized or distributed implementation. For example, a centralized implementation provides a convenient way to implement global policies. The rest of this section describes the three dynamic scheduling strategies and how they are implemented.

2.1 Description of the algorithms

The first strategy, *centralized scheduling*, uses a master-slave approach. The master processor maintains a queue of the tasks that are ready to run. When a slave processor finishes a task, it sends the master processor any new tasks that it has created. In return, the master processor sends the slave processor the next task to be executed. Sending all tasks to the master processor is part of the simple task transfer policy. The other part is that the master processor decides which tasks to send a slave based on the priority queue ordering of the tasks. Finally, the task location policy in this strategy is that any processor requesting work will receive a task when one is available. Both the task location and transfer policies are global since they rely on information about all the processors and all the tasks in the system.

At the other extreme, we examine a fully *distributed* approach to scheduling. In this approach, each processor maintains a local queue of tasks ready to run and schedules tasks to run locally if possible. The purely local workload may become too heavy or too light, so two strategies are possible for distributing tasks to and from remote nodes. The first is a *receiver initiated* strategy in which

tasks are requested from remote nodes if a processor decides that it needs more work. The second is a *sender initiated* strategy in which tasks are sent out voluntarily if the processor determines that it has too much work and needs to give some away. In either strategy, the local processor also decides which other processor to request work from or send work to. This scheduling approach is related to the adaptive load sharing policies of [4] in distributed systems, but in our work the approach can utilize knowledge about the parallel application.

In a receiver initiated strategy, task location happens in the following manner. If a processor decides that it is too lightly loaded, it sends a request message to another processor for additional tasks to execute. In this implementation, the *requestee* processor is randomly selected, although other methods for selecting a processor are possible (see below). Once the request message arrives at the requestee processor, the requestee processor interrupts its work and determines if it has *extra* tasks to give away. If so, it sends the task to the *requestor* processor that initiated the task request. Otherwise, it forwards the request by randomly selecting another processor and sending the requestor processor's message.

In a sender initiated strategy, if a processor decides that it is too heavily loaded, it searches for a processor in need of work. This is done by sending a task message to a randomly selected processor. If the processor receiving the task message needs work, it accepts the task. If the data associated with a task is too large to send along with task request, the receiving processor sends a message to the originating processor, asking them to send the task data. If a processor receiving a task message is not in need of work, then it randomly selects another processor and forwards the message.

In either strategy, the task transfer decision is based on whether the workload at a processor is too heavy or too light. In this paper we investigate two possible task transfer decisions, one based on local load information that could be obtained from information maintained by most operating systems, and a second decision based on additional information about the types of tasks available at that processor. That is, the first policy uses load information, or information that is independent of an application, while the second policy uses information that is specific to a particular application.

The third dynamic scheduling strategy we consider is a new *centralized mediation* approach that uses aspects of both the centralized and distributed strategies. In this strategy, a processor has a local task queue and schedules tasks to run locally if possible, as is done in the distributed case. If a processor decides it has too much work, however, it sends a task to a centralized *mediator* processor. Likewise, if a processor determines that it needs more work, it sends a request to this same, centralized, mediator processor. The mediator processor then matches available tasks to requests and sends the tasks to the processors that requested them.

```
----------------------------MASTER PROCESSOR-------------------------------
Repeat
        SCHEDULE tasks for idle slave processors.
        TASK finished at some slave processor?
             -> UPDATE computation status.
        NEW TASKS generated by some slave processor?
             -> INSERT into global priority queue.
        if appropriate
             -> CHECK if stopping criteria is satisfied and if it is
                          --> SEND STOP message to all processors.
Until done
--------------------------------------------------------------------------


----------------------------SLAVE/COMPUTATIONAL PROCESSOR-----------------
Repeat
        EXECUTE a task.
        SEND results and newly generated tasks to master processor.
        WAIT for a new task to execute or STOP message.
Until done
--------------------------------------------------------------------------
```

Figure 1: Centralized Scheduling

In this strategy, the task transfer policy is identical to that used in the distributed strategies. The task location policy is simplified because there is no "search" that must take place for processors with tasks to distribute or for processors that need tasks. Additionally, the location policy is non-local since the centralized mediator knows which processors have work to give away and which processors need work [1]. In this way, the mediation approach is a combination of a local task transfer policy and a centralized non-local task location policy, and combines sender and receiver initiated aspects.

A high level pseudo-code description of the control framework for each scheduling strategy is shown in figures 1, 2, and 3.

[1] Note, however, that it is not a global policy since it does not use information about the state of all processors; rather it uses information about only those processors requesting or sending tasks

```
-----------------------------ALL PROCESSORS-------------------------------
Repeat
        SCHEDULE and EXECUTE a task.
        NEW TASKS generated?
           -> INSERT new tasks into the local priority queue.
        UPDATE computation status
        NEED work?
           -> SEND request to a randomly selected processor.
           -> WAIT for a reply.
        if appropriate
             -> CHECK if stopping criteria is satisfied, and if it is
                          --> SEND STOP message to all processors.
        ..........................................................
        Interrupts:
            - When a NEED WORK message is received from some other processor -
                 Is EXTRA WORK available?
                    YES - SEND a task to the requesting processor.
                    NO - Forward request to another randomly selected
                         processor (if not above probe limit).
        ..........................................................
Until done
-------------------------------------------------------------------------
```

Figure 2: Distributed Scheduling (Receiver-initiated approach)

```
-----------------------------MEDIATOR PROCESSOR--------------------------
Repeat
        REQUEST WORK message received?
           - If tasks are available then send a task to the requesting
             processor.
           - Otherwise, insert into the request queue.
        EXTRA WORK message received?
           - If requests are available then send a task to the requesting
             processor.
           - Otherwise, insert into the task queue.
        UPDATE computation status.
        if appropriate
             -> CHECK if stopping criteria is satisfied, and if it is
                          --> SEND STOP message to all processors
Until done
-------------------------------------------------------------------------


-------------------------COMPUTATIONAL PROCESSOR--------------------------
Repeat
        SCHEDULE and EXECUTE a task.
        NEW TASKS generated?
           -> INSERT new tasks into local priority queue.
        UPDATE computation status.
        NEED work?
           -> SEND request to the mediator processor.
           -> WAIT for a reply.
        EXTRA work?
           -> SEND a task to the mediator processor.
Until done
-------------------------------------------------------------------------
```

Figure 3: Centralized Mediation

2.2 Implementation issues and questions

In this section we explain the reasoning behind some of the choices we have made in the implementation of the scheduling strategies. In addition, we introduce some of the questions that will be addressed in assessing the strategies. These are related to the four issues we mentioned at the beginning of Section 2.

At a high level, a goal of this study is to evaluate centralized mediation as a hybrid alternative to pure centralized or pure distributed scheduling strategies. Pure centralized scheduling does not scale well for large numbers of processors, and pure distributed strategies can be complex and difficult to implement. Both strategies have been widely studied [3], [13], but few alternatives have been proposed. In this study, we propose centralized mediation as a simple alternative to a strictly centralized or distributed strategy.

The lower level implementation issues we are interested in studying are the task transfer and task location policies. First we will briefly review some interesting work concerning these issues. There has been considerable research on these policies for load balancing of independent tasks in distributed systems. One issue that has been investigated is whether to use local or non-local information in these policies. Several notable load balancing strategies use non-local information. For example in [7], Kale discusses a scheduling strategy called *Contracting with Neighborhood* that uses information from nearest neighbors to determine if a task should be transferred and if so, which processor should receive it. When compared to a strategy that used random task location, Contracting within Neighborhood was reported to have significantly better performance. The gradient model of [8] also uses information from nearest neighbors in the task transfer and location policies. Other dynamic load balancing algorithms have been proposed that use global information in the transfer and location policies. For example, in [6], three strategies are presented that decide whether a task should be processed locally or remotely based on a global information that the processor periodically obtains about all other processors in the system.

There is also some precedent for using only local information in load balancing. The motivation is that the cost of acquiring status information from other processors can be high, especially if global information is used, thereby offsetting the potential benefits that could be realized by collecting this information. In [4], Eager, Lazowska and Zahorjan examine several load sharing strategies and report that there is no conclusive evidence to justify collecting and using extensive state information in task transfer and location policies. A similar conclusion is reached in [5], where a very thorough study of several distributed load balancing policies examines the benefits of using non-local state information

to make load balancing decisions.

The three strategies we examine allow us to compare three different combinations of local and global task transfer and task location policies. Distributed scheduling and centralized mediation involve a local task transfer policy, whereas centralized scheduling uses a global policy. For the task location policy, distributed scheduling uses a local random policy whereas centralized mediator and centralized scheduling use a global policy. The task location strategy in centralized scheduling also serves as the "optimal" scheduling strategy for comparison. Thus the three strategies can be viewed as purely local, purely global, or a combination of local and global.

Finally, since the task transfer decision relies only on local information, the nature of this information can be very important in making an effective decision. We could, for example, only use information that is maintained by most operating systems such as the number of tasks waiting to execute. A task transfer policy for this information would request work when the number of tasks at a processor fell below some threshold. This type of information is *application independent.* Another alternative is to use information that is specific to an application. For example, a computation might assign priorities to tasks based on some criteria specific to the application. This priority ordering could be used to make an estimate of the amount of work a processor has and whether it needs to ask for more work or give some work away. In addition, this information could be used to decide which tasks should then be transferred. This type of information is termed *application dependent.* A final goal of this study is to compare scheduling strategies that use application independent information with strategies that use application dependent information.

3 Application Background

Our most extensive performance analysis of the scheduling methods has involved the simulation of the three strategies when applied to one particular adaptive, dynamic parallel algorithm, for the global optimization problem. In this section we describe this parallel application and how we have implemented it using each of the three scheduling strategies.

3.1 Overview of the adaptive, asynchronous, global optimization algorithm

The global optimization problem is to find the minimum value of a nonlinear function f that may have multiple local minimizers over a domain, S. It arises in many areas of science and engineering, and is often very expensive to solve. Our algorithm to solve this problem is based upon the stochastic methods of [11].

In general, stochastic methods have approached the problem in the following manner. At iteration k:

1. Generate a random set of sample points in S and calculate their function values.
2. Select start points from this set for local minimizations.
3. Perform minimizations from all start points, each terminating at a local minimizer.
4. Decide whether or not to stop, and if not, repeat this process.

A static synchronous parallel algorithm based on these methods is presented in [2]. The parallel algorithm evenly divides the domain S into P subregions, where P is the number of available processors. Each processor performs steps 1 and 2 on its subregion. The processors synchronize after step 2 in order to eliminate potentially redundant searches, and then distribute the start points among the processors, perform the local minimizations of step 3, and synchronize again after this step to determine if the stopping conditions have been satisfied.

The adaptive asynchronous algorithm departs from the static synchronous by addressing load imbalance that often arises both in the start point selection, and the local minimization steps, and by attempting to reduce the amount of computation in subregions that are less likely to produce the global minimum. The adaptive aspect of the algorithm is used to identify portions of the domain space that appear productive and give them more attention, while diverting attention away from portions that are less fruitful. A convenient way to do this is to divide the domain space into subregions, and then to adjust the subregion sizes and/or the amount and frequency of work that is done in different subregions. Specific techniques we use to do this include splitting of subregions, adjustment of sampling densities, and delayed scheduling of particular subregions. The asynchronous behavior is achieved in part by using a new procedure that allows each subregion to decide independently whether to conduct local searches, thus eliminating the need for the synchronization after step 2. For a complete discussion of the adaptive and asynchronous features, see [12]. The experimental results in that paper show that the adaptive and asynchronous features can lead to very large reductions in the execution time required by the algorithm.

In order to explain how we have applied our scheduling strategies to this algorithm, it will be useful to illustrate the behavior of the adaptive algorithm. Figure 4 shows the initial partition of the domain space for some function, f. The sample points(x), start points(s), and local minimizers(m) found in the first step of the algorithm are also displayed. (In the figures new start points are circled and new local minimizers are surrounded by a square.) After the sampling and start point selection

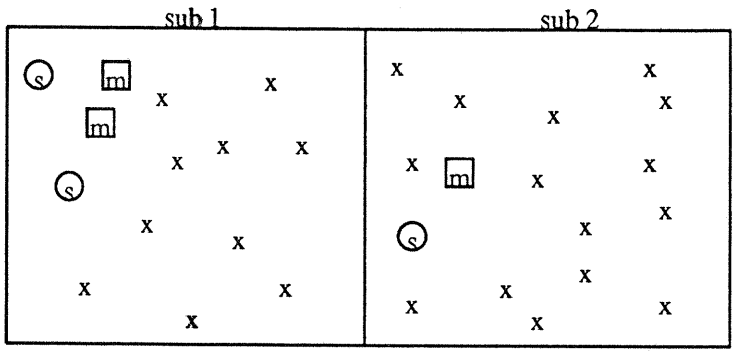

Figure 4: Global optimization example - iteration 1

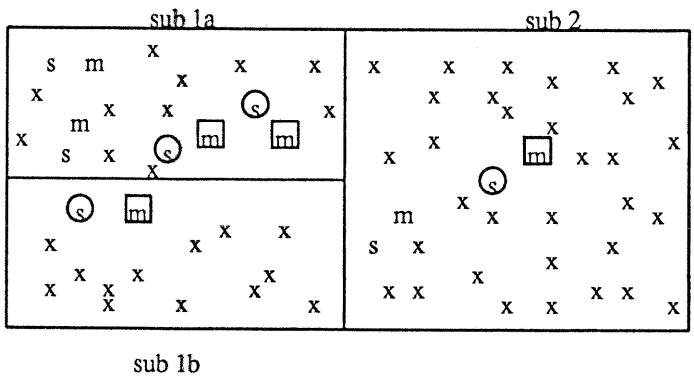

Figure 5: Global optimization example - iteration 2

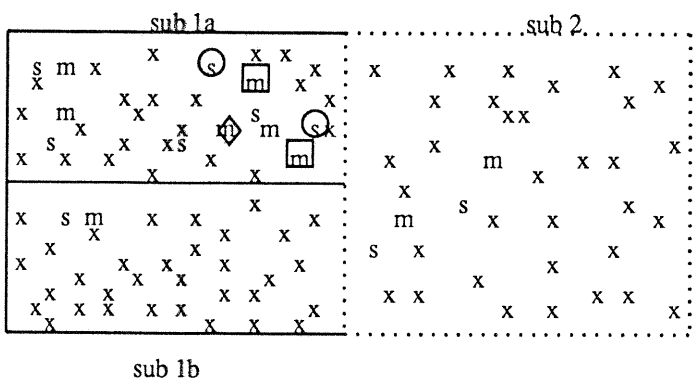

Figure 6: Global optimization example - iteration 3

steps for the first iteration, the adaptive heuristics are applied and yield the new division of the domain space illustrated in Figure 5. This figure also shows the new start points for iteration 2 and the local minimizers discovered from them. Figure 6 illustrates the final division of the domain space and the final set of start points and minimizers discovered. The global minimum is surrounded by a diamond. In this figure, subregion 2 is bordered by a dotted line to indicate that the adaptive heuristics found this subregion to be "unproductive", so that no sampling, start point selection, or local minimizations were executed for this subregion in iteration 3.

If we associate a *subregion* task with the sampling, start point selection and adaptive heuristic parts of the algorithm, and a *local minimization* task with the local minimization step of the algorithm, the example in Figures 4 - 6 can be described by the task precedence tree in Figure 7. In this figure, the subregion tasks are depicted by oval nodes, and the local minimization tasks are depicted by circles. The precedence tree shows the relationship between tasks in an execution of the algorithm, defining an order in which tasks are created and executed. In particular, a subregion task at level k precedes the creation and execution of its child local minimization tasks at iteration k, and the creation and execution of a subregion task at level k precedes the creation and execution of its child subregion task at level $k+1$.

While there is a strict ordering on the creation and execution of tasks in the tree that are direct descendents of each other, there is no imposed ordering between tasks that are siblings. The order of execution of subregion tasks at level k is nondeterministic, as is the order of execution of local minimization tasks at level k. The only constraint is that the execution of subregions at level k precede the execution of local minimizations at level k. However, it is possible that dynamic scheduling may cause this to be violated. The remainder of this section describes the way in which these tasks are executed and distributed in each of the three scheduling strategies.

3.2 Implementation of the parallel algorithm

In order to describe the implementation of the global optimization algorithm for the various scheduling strategies, for each of the strategies we need to describe (1) how tasks are scheduled, (2) the criteria used for task distribution, (3) how the computation status information is updated and distributed, and (4) the criteria used for stopping. First we need to define what global and local iterations are.

Global and Local Iterations

In the subsequent discussion, a *global iteration* will consist of all the tasks at a particular level of the task precedence tree. Global iteration k is said to complete when all the subregion and local

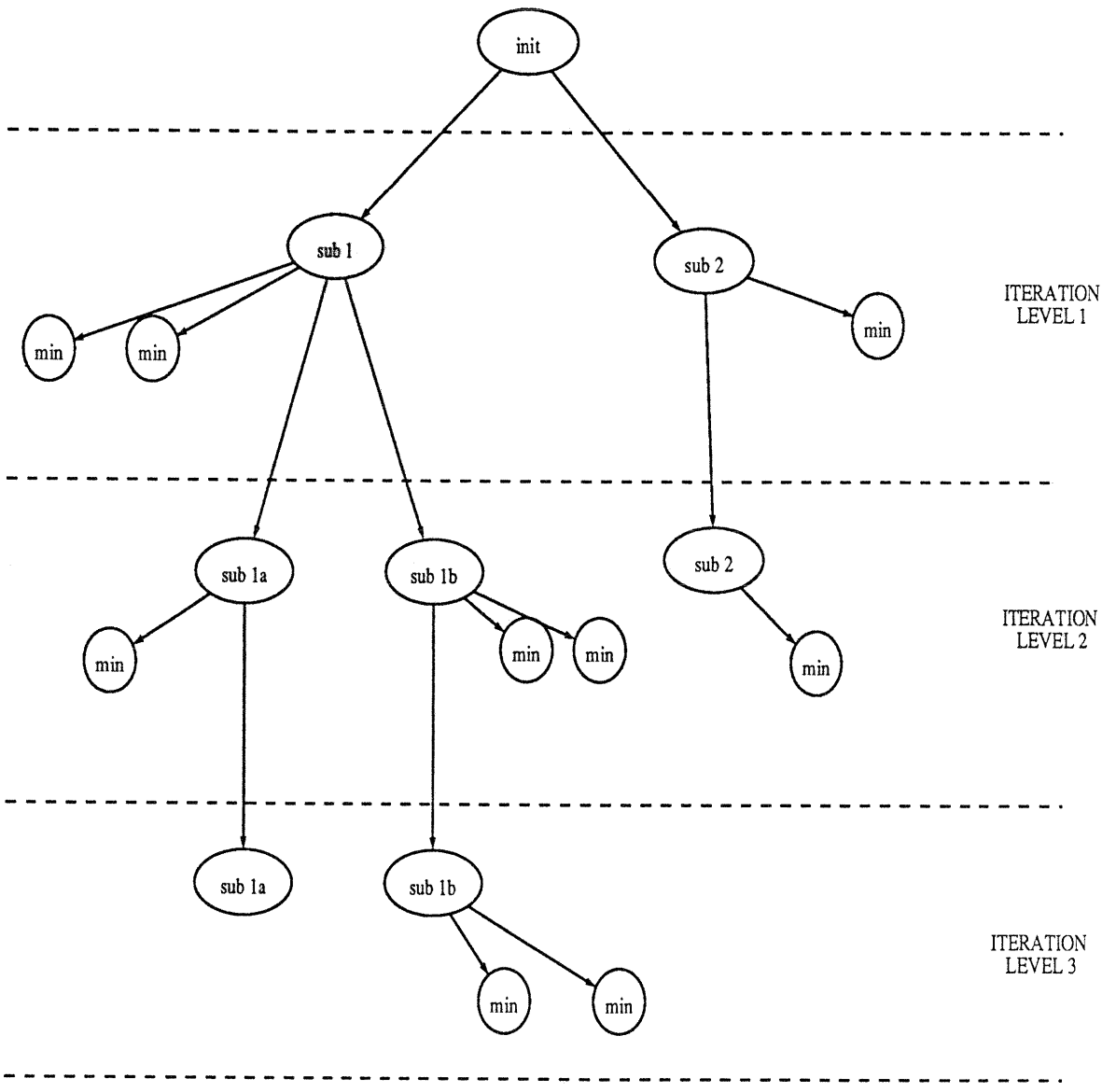

Figure 7: Task precedence in the adaptive global optimization algorithm

minimization tasks at level k in the tree have completed execution.

A *local iteration* is the subset of a global iteration seen by one processor. This subset arises in the following way. Initially, each processor starts out with one or more subregion tasks. The tasks generated by these initial subregion tasks are inserted into a local priority queue. Local iteration k is said to be complete when all tasks at level k in the local priority queue have completed execution.

It is possible that a subregion or local minimization task will be transferred to another processor to achieve load balance. When a task of level k is moved away from a processor, this processor no longer considers the task part of its local iteration k. Conversely, when this same task of level k arrives to a new processor, this processor inserts the new task into its priority queue and the new task is considered to be part of that processor's local iteration k (unless, of course, the task is moved again).

Finally, it is possible that a task or group of tasks may arrive to a processor with an iteration number $l < k$, where k is the last completed local iteration at the processor. When this occurs, the system resets the last completed iteration to $l - 1$, and it continues to schedule and execute tasks as before.

Task Scheduling

In all of the scheduling strategies, tasks are placed into a priority queue according to the following criteria which is used regardless of whether the queue is local (for centralized mediation and distributed scheduling) or global (for centralized scheduling): All tasks of level k precede tasks of level $k+1$, and all subregion tasks of level k precede local minimization tasks of level k. Note that this ordering is consistent with the task precedence tree.

Tasks with the highest priority are scheduled from the queue to run on a processor if they are *current.* Current tasks are tasks with an iteration level within an acceptable distance from the last known completed global iteration. This distance is known as the *asynchronicity level* because it limits the number of iterations that can be in progress at one time, and it ensures that the task precedence tree is explored in close to a breadth-first manner.

If no current tasks are available, the situation may change if new current tasks arrive or if the last known completed global iteration is incremented. In centralized scheduling, the response is to wait until one of these situations arises and not schedule tasks until then. When no current tasks are available in distributed receiver and centralized mediation, a request for new tasks from other processors is made. This is one of the ways in which task distribution is initiated. If such a request has already been made and denied, then the processor may schedule tasks beyond the boundaries

imposed by the asynchronicity level.

Task Distribution

In both centralized mediation and the distributed receiver strategy, new work, if needed, is requested at the end of a local iteration. To determine if new work is needed, either an *application specific* policy or a *threshold* policy is used.

The specifics of the task transfer policy for application specific centralized mediation and distributed receiver scheduling are as follows. Each processor sends a subregion task to the centralized mediator or to a requesting processor if it has at least two *current* subregion tasks. If a processor has only one current subregion task, and more than one local minimization task, it sends a local minimization task to the mediator or the requesting processor. Requests are made in the centralized mediation or distributed receiver initiated strategies when a processor has no current subregion tasks. The philosophy behind this strategy is that subregion tasks represent immediate work, as well as the possibility of future work. Therefore, it is desirable to distribute subregion tasks first, when possible.

In application independent centralized mediation or distributed receiver scheduling, a request is made for tasks when the number of tasks falls below some threshold. These tasks can be current or non-current since the operating system cannot distinguish between these type of tasks. In centralized mediation or distributed receiver initiated scheduling, a request is satisfied by a processor if the number of tasks it has is above some threshold.

Updating the computation status

Our strategies for scheduling tasks to execute and for testing the stopping criteria require that the last completed global iteration be known. In the centralized scheduler strategy, the master processor keeps track of all tasks of a particular iteration. As the tasks complete execution, the status of the iteration is checked and updated if necessary. In the distributed and centralized mediation strategies, computation status messages are sent out by each processor at the end of a local iteration. In the distributed strategy this is done by sending a message to every other processor. In the centralized mediation strategy this is done by sending a message to the mediator processor, who in turn informs the other processors each time a global iteration completes. The result in either case is that each processor knows the last global iteration to have completed.

Stopping Criteria

In the static, synchronous algorithm, the stopping criteria are tested at the end of an iteration. In

the asynchronous version, the end an iteration is not as conveniently recognized, however the concept of an iteration is maintained to preserve precedence relationships and to test the stopping criteria in a uniform way for all the scheduling strategies. The information needed to test the stopping criteria are the number of local minimizers discovered and the sample size.

The process that is used to maintain precedence information about the global iterations completed also provides a convenient way to obtain the information necessary to test the stopping conditions. The global stopping test is based on the local minimizers that have been discovered by local minimization tasks. In both of the centralized strategies, this information is sent along with status information messages to the centralized processor. In the distributed strategy, this information is incorporated into the status message sent at the end of every local iteration. In all cases, the check regarding whether the global stopping condition has been satisfied is made at the end of a local iteration. The main difference is that the three strategies may receive the information about local minimizations that is necessary to make the global stopping tests at different times. In the centralized scheduling strategy, local minimization information is received at the completion of every local minimization task. In the centralized mediator and distributed approaches, local minimization information is sent around less frequently, i.e. whenever a processor finishes a local iteration. In each strategy, when the stopping criteria have been satisfied, all the processors are informed and the computation is terminated.

4 Experimental Strategy

We have compared the three scheduling approaches using a combination of simulation and parallel implementation. In this section we describe how each of these methods has been used in assessing the performance of the different approaches.

4.1 Implementation

We have implemented a parallel version of the adaptive global optimization algorithm using the centralized scheduler strategy. This implementation runs on a network of SUN workstations using GRAIL, a set of message based library routines for parallel processing in a network environment. We have run experiments using this implementation with up to 9 processors (1 master processor and 8 slave processors).

We have used this implementation to establish the effectiveness of the adaptive techniques, generate execution traces that are used by the simulation strategies, and to validate the centralized

scheduler simulations. The next section discusses how the execution traces have been developed and used.

4.2 Simulation

To model the execution of the adaptive parallel global optimization algorithm using various scheduling strategies, we have used simulations based upon the execution traces generated by our parallel implementation. These execution traces were obtained from experimental runs of the implementation for 5 different problems running on up to 8 slave processors. The execution traces are comprised of timings from the sampling, start point selection, local minimization, and communication steps of the algorithm. These timings are then used in a discrete event simulation based on these events and the desired scheduling policy.

To investigate other situations we have modified the traces in the following ways. First, we have modified the traces to simulate the situation in which fewer tasks are created by the adaptive algorithm. This is accomplished by pruning branches of the task precedence tree for an experimental run. This allows us to observe the performance of the scheduling strategies when there is not enough work to keep all the processors busy. In a similar manner, we have modified the traces to simulate more tasks being created. This is accomplished by duplicating productive branches of the task precedence tree. This modification results in an abundance of work being available to the processors. Finally, we have duplicated each set of traces (in multiples of 8) to examine the behavior of the algorithms on as many as 64 processors.

We have validated the simulation for the centralized strategy that is based upon the unmodified traces from the experimental runs, and found the simulation performance measures to be on the average within 3.5 % and at worst within 10 % of the actual experiments. This indicates that our simulation results should be reliable.

5 Results and Conclusions

In this section we discuss some of the simulation experiment results that have been useful in evaluating the three different scheduling strategies, and our conclusions from these experiments.

5.1 Test problems

In this section we show the simulation results from two representative problems, referred to as problem A and problem B. These problems are described in [12].

The tables in this section are organized as follows. There are three sets of tables for each problem. The first set of tables presents the simulation results when a small number of tasks are created and there is not enough work to keep all the processors busy all the time. The second set of tables presents the simulation results for a medium number of tasks in the system, enough work to go around if carefully distributed. The final set of tables presents the results for a large number of tasks and abundant work in the system.

Each table set consists of simulations for 8, 16, 32, and 64 processors. Each table shows the system execution times and processor utilization for 4 different scheduling strategies: centralized scheduling, centralized mediation, distributed receiver application dependent scheduling, and distributed receiver application independent scheduling. Preliminary results indicated that the distributed sender strategy performed worse than the distributed receiver strategy, so we did not further experiment with this strategy. For centralized scheduling and centralized mediation, the master utilization refers to the utilization of the processor running the centralized scheduler or centralized mediation process.

5.2 Results

The main objective of this evaluation is to contrast the performance characteristics of centralized mediation with the pure centralized and distributed strategies. In addition, we are interested in evaluating some of the implementation details for task location and transfer policies.

1. **Centralized Scheduling vs. Centralized Mediation**

   We first consider the situation in which a small number of tasks are created (Tables 1 and 4). In this situation, centralized scheduling runs up to 31 % faster for 8 processors. The range of processor utilizations for the slave processors in centralized scheduling indicates that there is not enough work to keep processors occupied all the time. The range of processor utilizations for the centralized mediator show that it is not as effective at distributing the tasks.

   Under the same loading conditions with 16, 32, and 64 processors, the centralized mediator always performs better than the centralized scheduler. In the case of 16 processors it is up to 33 % faster, with 32 processors it is up to 53% faster, and with 64 processors, it is up to 69 % faster. The lower bound processor utilizations for centralized mediation and centralized scheduler are similar for these cases, while the upper bound utilizations are much larger for centralized mediation. In all the cases, the best balanced processor utilization (or the smallest range of utilizations) is realized with the centralized scheduling strategy, even though the processor utilizations become quite low since the processor dedicated to the centralized scheduler

becomes a bottleneck at 32 processors.

For a medium number of tasks in the system (Tables 2 and 5), the centralized mediator consistently outperforms centralized scheduling. The gains for 8 and 16 processors are modest, ranging from 10 % to 37 %. For 32 and 64 processors, the master processor in centralized scheduling is very saturated, so that centralized mediation performs up to 6 times as fast.

Centralized mediation also outperforms centralized scheduling when the system has a large numbers of tasks present (Tables 3 and 6). For 8 and 16 processors, the centralized mediation has performance gains that range from 8 % to 40 %. For 32 and 64 processors, centralized mediation again performs up to 6 times as fast. With this loading condition, the computational processor utilizations are all quite high, and both the scheduling strategies achieve good load balance.

2. **Centralized Mediation vs. Distributed receiver-initiated scheduling**

   Under all loading conditions the centralized mediator performs consistently better than distributed scheduling for 8, 16, and 32 processors. In the situation where a small number of tasks are present in the system (see Tables 1 and 4), the performance gain ranges from 5 % to 25 %. For a medium number of tasks in the system (see Tables 2 and 5), the performance improvement realized by the centralized mediator ranges from to 13 % to 23 %. Finally, with large numbers of tasks in the system, the performance improvement ranges from 14 % to 27 % (see Tables 3 and 6). The range of computation processor utilizations are generally higher for centralized mediation than for distributed receiver under all loading conditions.

   At 64 processors, the centralized mediator processor is becoming heavily loaded, and performance in this case is slightly better with the distributed strategy. On the average (over all loading conditions), the distributed receiver strategy is 5 % faster than centralized mediation. In the best situation, centralized mediation is 5 % faster than distributed receiver, and in the worst case, it is 10 % slower.

3. **Application Independent vs. Application specific strategies**

   For both problems and their configurations, we tested the two distributed receiver strategies described in section 3, and in general, the application dependent strategy performed the best. We found that the largest performance gains (22 % - 32 %) occur when there are a small number of tasks in the system. The smallest performance gains (-2 % - 27 %) occur when a large number of tasks are present in the system. When medium numbers of tasks are present

in the system, the gains are between 10 % and 27 %. This result indicates that the choice of a task transfer policy is more important to the scheduling strategy when there are fewer tasks to distribute and the placement of each task becomes more critical to performance.

5.3 General Conclusions

In conclusion, we make the following general observations about the scheduling strategies.

1. Centralized mediation performs better than centralized scheduling in most cases. This is to be expected because the master process in centralized scheduling quickly becomes a bottleneck. The exception occurs for the case when there are only a small number of tasks in the system and a small number of processors operating on these tasks. In addition, centralized mediation degrades more slowly with respect to the number of processors than centralized scheduling.

2. For 8, 16, and 32 processors, centralized mediation performs consistently better than distributed receiver, but it loses the performance advantage when the number of processors grows because the centralized mediator processor becomes a bottleneck. We expect that this problem can be overcome with a hierarchical mediation strategy (see Section 5.4).

3. Application specific heuristics in the distributed receiver strategy do a better job of task distribution than application dependent heuristics, especially when there are a small number of tasks in the system and the placement of each task becomes more important to the overall system performance.

5.4 Future Work

Our research is continuing to investigate the scheduling approaches, in a broader set of applications and in actual parallel implementations. In particular, since the centralized mediation strategy seems to offer a good combination of efficiency and ease of implementation in many cases, we are investigating several enhancements to the centralized mediator strategy. First, we are considering adaptive extensions that adjust the number of requests and tasks sent to the centralized mediator processor depending on the number of tasks being created by an algorithm. For example, if there are a small number of tasks in the system, the centralized mediator will instruct the processors to make less frequent requests. Additionally, the criteria to send tasks to the mediator will be relaxed so that the mediator will have enough tasks to satisfy requests.

We are also investigating hierarchical approaches to dynamic scheduling that would address the scaling problem that eventually arises in the centralized medation strategy. For example, a two level

centralized mediation strategy would mediate processors in clusters at the first level, and then resolve imbalance between clusters at the second level. This strategy may extend the efficiency and simplicity of the centralized mediation strategy to larger numbers of processors.

References

[1] M.J. Berger. Adaptive mesh refinement for parallel processors. In *Parallel Processing for Scientific Computing*, December 1987.

[2] R. H. Byrd, C. L. Dert, A. H. G. Rinnooy Kan, and R. B. Schnabel. Concurrent stochastic methods for global optimization. *Mathematical Programming*, 46:1–29, 1990.

[3] T. L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, SE-14:141–154, 1988.

[4] D. L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12:662–675, 1986.

[5] Dirk Grunwald. Circuit switched multicomputers and heuristic load placement. Technical Report UIUCDCD-R-89-1514, University of Illinois at Urbana-Champaign, September 1989. PhD Thesis.

[6] C.H. Hsu and J. Liu. Dynamic load balancing algorithms in homogeneous distributed systems. In *Proceedings of the 6th International conference on Distributed Computing Systems*, August 1986.

[7] L. V. Kale and W. Shu. Comparing the performance of two dynamic load distribution methods. In *International Conference of Parallel Processing*, August 1988.

[8] F.C.H. Lin and R.M. Keller. Gradient model: A demand-driven load balancing scheme. In *Proceedings of the 6th International conference on Distributed Computing Systems*, August 1986.

[9] S. McCormick and D. Quinlan. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocesors: performance results. Technical report, University of Colorado at Denver, October 1988.

[10] D. M. Nicol and J.H. Saltz. Schedules for mapping irregular parallel computations. Technical Report 87-52, Institute for Computer Applications in Science and Engineering, September 1987.

[11] A. H. G. Rinnooy Kan and G. T. Timmer. A stochastic approach to global optimization. In P. Boggs, R. Byrd, and R. B. Schnabel, editors, *Numerical Optimization*, pages 245 – 262. SIAM, Philadelphia, 1984.

[12] S. L. Smith, Elizabeth Eskow, and Robert B. Schnabel. Adaptive, asynchronous stochastic global optimization algorithms for sequential and parallel computation. Technical Report CU-CS-449-89, Department of Computer Science - University of Colorado, Boulder, October 1989.

[13] Y. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34:204–217, 1985.

Test case	System time	processor utilization	master util
centralized scheduler	2:41	50 % - 81 %	38 %
centralized mediator	3:40	24 % - 90 %	11 %
distr. receiver	4:03	23 % - 76 %	-
distr. receiver independent	5:28	19 % - 64 %	-

Test case	System time	processor utilization	master util
centralized scheduler	2:08	38 % - 62 %	80 %
centralized mediator	2:01	22 % - 95 %	28 %
distr. receiver	2:32	15 % - 71 %	-
distr. receiver independent	3:44	10 % - 62 %	-

Test case	System time	processor utilization	master util
centralized scheduler	3:33	15 % - 32 %	97 %
centralized mediator	1:41	21 % - 86 %	66 %
distr. receiver	2:14	9 % - 67 %	-
distr. receiver independent	3:14	6 % - 61 %	-

Test case	System time	processor utilization	master util
centralized scheduler	6:50	6 % - 15 %	99 %
centralized mediator	2:24	10 % - 62 %	94 %
distr. receiver	2:13	5 % - 65 %	-
distr. receiver independent	2:50	4 % - 60 %	-

Table 1: Comparison of scheduling techniques for problem A tests involving small numbers of tasks for 8, 16, 32, and 64 processors

Test case	System time	processor utilization	master util
centralized scheduler	3:30	77 % - 95 %	51 %
centralized mediator	3:08	62 % - 96 %	7 %
distr. receiver	3:36	66 % - 92 %	-
distr. receiver independent	4:54	33 % - 77 %	-

Test case	System time	processor utilization	master util
centralized scheduler	3:48	53 % - 71 %	93 %
centralized mediator	2:23	66 % - 97 %	19 %
distr. receiver	2:53	50 % - 91 %	-
distr. receiver independent	3:52	23 % - 78 %	-

Test case	System time	processor utilization	master util
centralized scheduler	7:18	19 % - 34 %	99 %
centralized mediator	2:19	48% - 93 %	50 %
distr. receiver	2:52	38 % - 83 %	-
distr. receiver independent	3:30	18 % - 74 %	-

Test case	System time	processor utilization	master util
centralized scheduler	13:10	8 % - 17 %	99 %
centralized mediator	3:04	36% - 83 %	96 %
distr. receiver	2:44	31 % - 78 %	-
distr. receiver independent	3:25	13 % - 69 %	-

Table 2: Comparison of scheduling techniques for problem A tests involving medium numbers of tasks for 8, 16, 32, and 64 processors

Test case	System time	processor utilization	master util
centralized scheduler	3:48	82 % - 95 %	47 %
centralized mediator	3:03	77 % - 98 %	7 %
distr. receiver	3:34	75 % - 96 %	-
distr. receiver independent	4:19	52 % - 86 %	-

Test case	System time	processor utilization	master util
centralized scheduler	3:48	58 % - 74 %	92 %
centralized mediator	2:17	74 % - 98 %	20 %
distr. receiver	2:40	67 % - 94 %	-
distr. receiver independent	3:18	43 % - 83 %	-

Test case	System time	processor utilization	master util
centralized scheduler	6:55	22 % - 37 %	98 %
centralized mediator	2:13	62 % - 95 %	53 %
distr. receiver	2:34	56 % - 90 %	-
distr. receiver independent	2:54	28 % - 77 %	-

Test case	System time	processor utilization	master util
centralized scheduler	13:35	8 % - 17 %	99 %
centralized mediator	2:27	44 % - 94 %	94 %
distr. receiver	2:34	47 % - 87 %	-
distr. receiver independent	2:41	16 % - 72 %	-

Table 3: Comparison of scheduling techniques for problem A tests involving large numbers of tasks on 8, 16, 32, and 64 processors

Test case	System time	processor utilization	master util
centralized scheduler	3:06	67 % - 90 %	43 %
centralized mediator	4:30	40 % - 94 %	8 %
distr. receiver	4:43	33 % - 82 %	-
distr. receiver independent	6:57	21 % - 65 %	-

Test case	System time	processor utilization	master util
centralized scheduler	2:40	46 % - 68 %	83 %
centralized mediator	1:58	52 % - 95 %	29 %
distr. receiver	2:24	34 % - 78 %	-
distr. receiver independent	3:27	24 % - 69 %	-

Test case	System time	processor utilization	master util
centralized scheduler	4:47	17 % - 29%	96 %
centralized mediator	1:34	39 % - 91 %	62 %
distr. receiver	1:53	23 % - 73 %	-
distr. receiver independent	2:44	16 % - 68 %	-

Test case	System time	processor utilization	master util
centralized scheduler	9:11	7 % - 15%	99 %
centralized mediator	1:52	23 % - 91 %	88 %
distr. receiver	1:41	15 % - 72 %	-
distr. receiver independent	2:24	11 % - 63 %	-

Table 4: Comparison of scheduling techniques for problem B tests involving small numbers of tasks for 8, 16, 32, and 64 processors

Test case	System time	processor utilization	master util
centralized scheduler	4:14	86 % - 98 %	46 %
centralized mediator	3:46	83 % - 97 %	7 %
distr. receiver	4:23	86 % - 98 %	-
distr. receiver independent	4:57	60 % - 87 %	-

Test case	System time	processor utilization	master util
centralized scheduler	4:14	58 % - 77 %	90 %
centralized mediator	2:40	80% - 96 %	21 %
distr. receiver	3:28	71 % - 94 %	-
distr. receiver independent	3:52	51 % - 87 %	-

Test case	System time	processor utilization	master util
centralized scheduler	7:42	21 % - 37 %	97 %
centralized mediator	2:08	74 % - 97 %	53 %
distr. receiver	2:46	68 % - 93 %	-
distr. receiver independent	3:26	47 % - 83 %	-

Test case	System time	processor utilization	master util
centralized scheduler	15:21	8 % - 18 %	99 %
centralized mediator	2:24	61 % - 94 %	92 %
distr. receiver	2:21	61 % - 85 %	-
distr. receiver independent	3:05	35 % - 78 %	-

Table 5: Comparison of scheduling techniques for problem B tests involving medium numbers of tasks for 8, 16, 32, and 64 processors

Test case	System time	processor utilization	master util
centralized scheduler	4:02	87 % - 94 %	44 %
centralized mediator	3:43	85 % - 98 %	7 %
distr. receiver	4:59	88 % - 97 %	-
distr. receiver independent	4:52	79 % - 94 %	-

Test case	System time	processor utilization	master util
centralized scheduler	3:56	61 % - 80 %	89 %
centralized mediator	2:34	79 % - 97 %	21 %
distr. receiver	3:30	80 % - 96 %	-
distr. receiver independent	3:48	73 % - 91 %	-

Test case	System time	processor utilization	master util
centralized scheduler	7:09	22 % - 39 %	97 %
centralized mediator	2:07	76 % - 97 %	50 %
distr. receiver	2:37	71 % - 93 %	-
distr. receiver independent	3:06	53 % - 87 %	-

Test case	System time	processor utilization	master util
centralized scheduler	13:55	8 % - 18 %	99 %
centralized mediator	2:23	63 % - 98 %	92 %
distr. receiver	2:14	64 % - 90 %	-
distr. receiver independent	3:03	50 % - 83 %	-

Table 6: Comparison of scheduling techniques for problem B tests involving large numbers of tasks on 8, 16, 32, and 64 processors