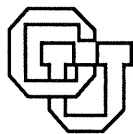
Dynamic Scheduling Strategies for an Adaptive, Asynchronous Parallel Global Optimization Algorithm

Sharon L. Smith and Robert B. Schnabel

CU-CS-625-92 November 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Dynamic Scheduling Strategies for an Adaptive, Asynchronous Parallel Global Optimization Algorithm[1]

Sharon L. Smith[2] and Robert B. Schnabel[3]

CU-CS-625-92 November 1992

[1]Research supported by NSF grant CDA-8922510, AFOSR grant AFOSR-90-0109, and ARO grant DAAL 03-91-G-0151.

[2]CERFACS, 42, Avenue Gustave Coriolis, 31057 Toulouse Cedex, France (smith@orion.cerfacs.fr)

[3]Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado, 80309 U.S.A. (bobby@cs.colorado.edu)

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Dynamic Scheduling Strategies for an Adaptive, Asynchronous Parallel Global Optimization Algorithm

Abstract

This paper explores the use of dynamic scheduling strategies for irregular parallel algorithms in distributed memory computational environments. The target application we consider is an adaptive asynchronous parallel algorithm with irregular structure that is used to solve the global optimization problem. In this algorithm the number of tasks and their sizes may change dynamically, so that dynamic scheduling is needed to insure that the workload is evenly distributed across the processors. We consider three dynamic scheduling strategies for implementing this algorithm: centralized scheduling, which uses a master-slave approach; 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 hybrid approach that we refer to as centralized mediation, that uses aspects of both centralized and distributed scheduling. The implementation of the global optimization algorithm using the scheduling strategies is discussed, and their performance is thoroughly assessed through a combination of analytic modeling, simulation, and distributed implementation. In these performance studies, the centralized mediation strategy often exhibits the best performance for both different numbers of processors and different loading conditions.

1. Introduction

In recent years, the increased availability of inexpensive hardware to construct parallel computer systems has provided researchers with the hardware tools to investigate the design and implementation of parallel algorithms for complex problems. In many cases, significant performance improvements have been realized for algorithms with straightforward parallelism. There are many algorithms, however, that exhibit *irregular* structure and that may suffer from load imbalance, poor processor utilization, and consequently, poor program performance.

In order to develop efficient algorithms for irregular computations on parallel computers,

algorithm designers have begun to consider and develop alternative approaches, such as adaptive and asynchronous algorithms (see, for example [3], [19], [27], and [14]). *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. *Asynchronous* algorithms allow each part of the computation to proceed independently of other parts. Because the use of these techniques may cause variation in the numbers and sizes of tasks, robust scheduling techniques are needed to ensure that the workload is evenly distributed among the processors.

Part of our research has been concerned with developing an adaptive asynchronous parallel algorithm for solving the global optimization problem (see [25]). Our initial experience with this algorithm using a centralized scheduling strategy has led us to consider alternative scheduling strategies and how they may be used to implement irregular parallel computations. The principal focus of this paper is to examine in detail different scheduling strategies for the parallel global optimization algorithm, and how well these strategies perform for distributed systems. Distributed systems in this context will mean computers with local memory connected by a local network and communicating via message passing. While the experimental environment employed is a local area network of computers, this research also is applicable to general distributed memory multiprocessors, such as hypercubes and mesh-connected MIMD computers.

Because implementation of complex parallel algorithms can be difficult and time consuming, we have developed simulation models and analytic models of the scheduling strategies examined in this paper to aid in the evaluation of the different strategies. To demonstrate the usefulness of this type of evaluation, we present validations of these models with actual implementations of two of the scheduling strategies.

Although we limit our attention to the application of global optimization in this paper, it is possible that the results of this study of scheduling methods will also be applicable to other parallel algorithms with similar characteristics. The extent to which this is the case is not known,

At iteration k:

1. Generate a random set of sample points in S and calculate their function values.
2. Select a subset of the low sample points to be start points 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.

Figure 2.1: Static Global Optimization Algorithm

however, and will not be the focus of this paper.

The paper is organized as follows. Section 2 introduces the parallel adaptive asynchronous algorithm for solving the global optimization problem, which is the target of our study. Section 3 describes the centralized, distributed, and hybrid scheduling strategies and how they are used to implement the parallel global optimization algorithm. Section 4 presents a performance evaluation of the three strategies using analytic modeling, simulation, and implementation. Finally, Section 5 presents conclusions about this work.

2. Overview of Adaptive, Asynchronous Global Optimization

Our most extensive performance analysis of the scheduling methods described in this paper has used analytic modeling, simulation, and implementation to assess their performance when applied to one particular parallel algorithm for solving the global optimization problem. In this section we describe this 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. This problem 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 [22]. Figure 2.1 gives a high-level outline of the stochastic approach.

A static synchronous parallel algorithm based on these methods is presented in [5]. 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

For each subregion s of the domain space S, at every iteration, k,

1. **Sampling :**
   Generate a prescribed number of random sample points within s and calculate their function values.

2. **Start point selection :**
   Select sample points with low function values in s to be start points, using a procedure that is similar to the static algorithm but which does not involve communication with any other subregions.

3. **Adaptive decisions :**
   Apply a heuristic procedure to determine the sample size for s for the next iteration, whether s should be split, and whether s should be scheduled or skipped at the next iteration.

4. **Local minimizations :**
   Perform local minimizations, if any, from the start points generated at step 2.

Figure 2.2: Adaptive Asynchronous Global Optimization Algorithm

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 static algorithm performs well for small, regular problems, but has two major weaknesses. First, the synchronizations after steps 2 and 3 can lead to severe load imbalance, because both the time for the start point selection step and for local minimizations can vary greatly between subregions and hence processors. Secondly, the algorithm makes no effort to adapt its sampling density to give more attention to productive subregions, those that appear more likely, as the computation proceeds, to produce the global minimizer.

The adaptive asynchronous algorithm, described in Figure 2.2, addresses these weaknesses. 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 make the adaptive adjustments 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, according to how productive the subregions appear. 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 used to address the load imbalance caused by synchronization. It 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, and by relaxing the global concept of an iteration (see Section 3.3.1). For a complete discussion of the adaptive and asynchronous features, including ways to determine which subregions are most productive, see [25]. 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.

It is useful to illustrate the behavior of the adaptive asynchronous algorithm through an example. Figure 2.3a shows the initial partition of the domain space for some function, f, with a two-dimensional domain. 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 steps for the first iteration, the adaptive heuristics are applied and yield the new division of the domain space illustrated in Figure 2.3b. This figure also shows the new start points for iteration 2 and the local minimizers discovered from them. Figure 2.3c 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 define a *subregion* task to consist of the sampling, start point selection and adaptive heuristic parts of the algorithm, and a *local minimization* task to consist of the local minimization step of the algorithm, the example in Figure 2.3 can be described by the task precedence tree shown in Figure 2.4. 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

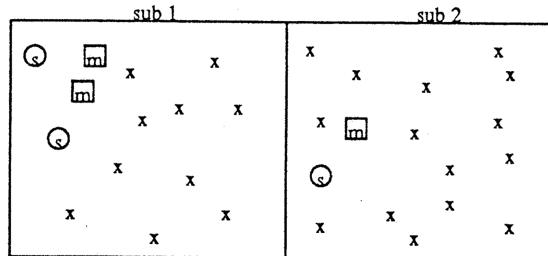


Figure 2.3a: Iteration 1

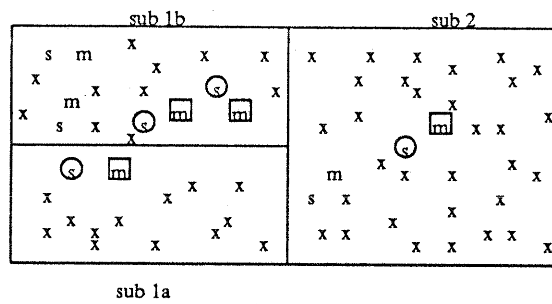


Figure 2.3b: Iteration 2

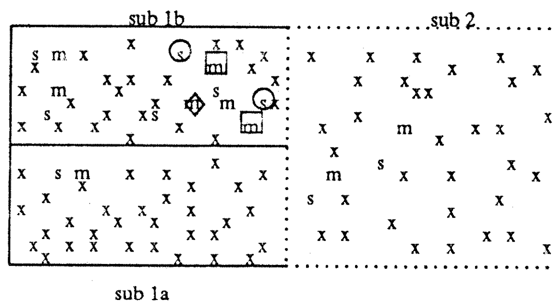


Figure 2.3c: Iteration 3

Figure 2.3: Global optimization example

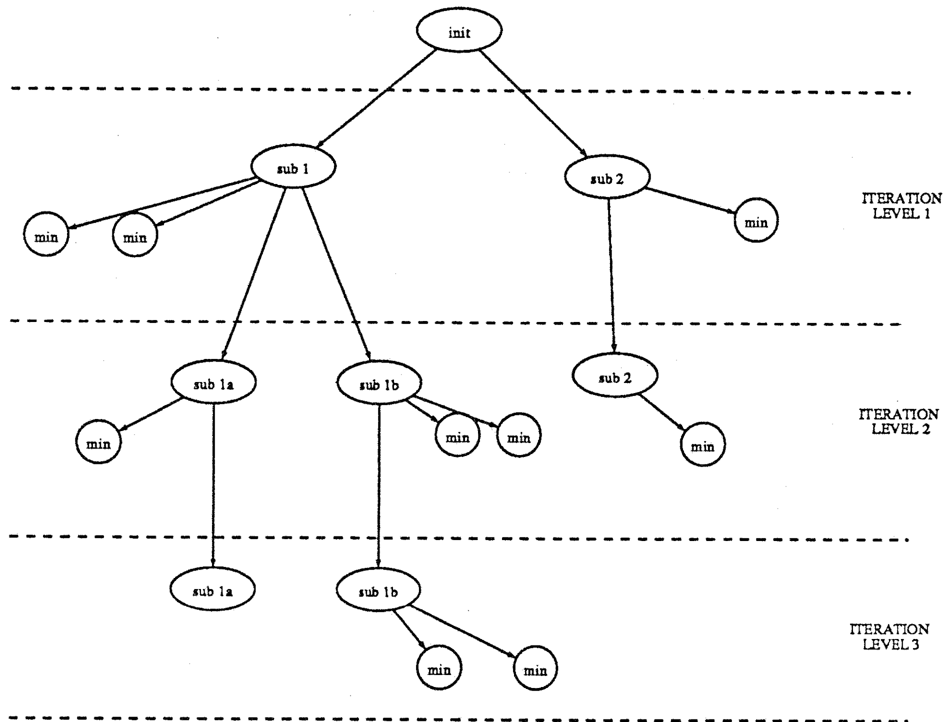


Figure 2.4: Task precedence in the adaptive global optimization algorithm

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 $k+1$, but otherwise the order of execution is not constrained. In the next section we will described the scheduling of these tasks.

3. The scheduling strategies and their implementation of global optimization

This section presents the scheduling strategies, describes related work in this area, and discusses how the scheduling strategies are used to implement the adaptive, asynchronous global optimization algorithm.

3.1. Description of the scheduling strategies

The scheduling strategies considered in this research are centralized strategies, where the scheduling and task distribution decisions are performed by a single processor; distributed strategies, where

```
-----------------------------MASTER PROCESSOR------------------------------
Repeat
        SCHEDULE tasks for all idle slave processors based on priority
             queue of available tasks.
        TASK finished at some slave processor?
             -> UPDATE computation status of processor.
        NEW TASKS generated by some slave processor?
             -> INSERT into global priority queue.
        if appropriate
             -> CHECK if global 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.
        STOP message received?
             -> DONE = TRUE
Until done
--------------------------------------------------------------------------
```

Figure 3.1: Centralized Scheduling

scheduling and task distribution decisions are performed by all processors; and hybrid strategies, which have aspects of both centralized and distributed strategies.

The first strategy, *centralized scheduling*, uses a master-slave approach. The master processor maintains a priority queue of the tasks that are ready to run. When a slave processor finishes a task, it informs the master process and sends it any new tasks that it has created. In return, the master process sends the slave processor the next task to be executed, based on the priority queue ordering of the tasks.

At the other extreme from centralized scheduling is a fully *distributed* approach to scheduling. In this approach, each processor maintains a local queue of tasks that are ready to run, and schedules tasks from the queue to run locally if possible. The purely local workload may become too heavy or too light, however, so two strategies are possible for distributing tasks to and receiving tasks from remote nodes. The first is a *receiver-initiated* strategy in which tasks are requested from remote nodes if a given processor decides that it needs more work than it has locally. The second is a *sender-initiated* strategy in which tasks are sent out without being requested if a given processor determines that it has too much work and needs to give some away. In either strategy, the local

```
----------------------------ALL PROCESSORS------------------------------
Repeat
        SCHEDULE and EXECUTE a task.
        NEW TASKS generated?
           -> INSERT new tasks into the local priority queue.
        UPDATE computation status
           -> PROCESS update messages from other processors.
        NEED work?
           -> SEND request to a randomly selected processor.
           -> BLOCK for a reply.
        if appropriate
           -> CHECK if global stopping criteria is satisfied, and if it is
                           --> SEND STOP message to all other processors.
        STOP message received?
           -> DONE = TRUE
        ..............................................................
        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 3.2: Distributed Scheduling (Receiver-initiated approach)

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 [9] in distributed systems.

The third dynamic scheduling strategy considered 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, rather than sending work directly to other processors, 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. In this way, the mediation approach combines the sender and receiver initiated aspects of distributed scheduling with a centralized approach. In contrast to the centralized strategy, which handles all tasks and requests, the central mediator processor handles only the tasks and requests that cannot be accommodated locally.

High level pseudo-code descriptions of the control framework for the centralized, dis-

```
-----------------------------ALL PROCESSORS-------------------------------
Repeat
        SCHEDULE and EXECUTE a task.
        NEW TASKS generated?
           -> INSERT new tasks into the local priority queue.
        UPDATE computation status
           -> PROCESS update messages from other processors.
        EXTRA work?
           -> SEND a task message to a randomly selected processor.
        if appropriate
           -> CHECK if global stopping criteria is satisfied, and if it is
                           --> SEND STOP message to all other processors.
        STOP message received?
           -> DONE = TRUE
        ............................................................
        Interrupts:
           -> When a TASK message is received from some other processor -
                 Is EXTRA WORK needed?
                    YES - SEND a request NEW TASK to the processor sending the
                          task message.
                          WAIT for the task data to arrive.
                    NO - Forward request to another randomly selected
                         processor (if not above probe limit).

           -> When a NEW TASK data request is received
                 SEND task.
        ............................................................
Until done
--------------------------------------------------------------------------
```

Figure 3.3: Distributed Scheduling (sender-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.
             -> PROCESS update messages from the computational processors.
        if appropriate
             -> CHECK if global 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.
             -> PROCESS update messages from mediator processor.
        NEED work?
           -> SEND request to the mediator processor.
           -> WAIT for a reply.
        EXTRA work?
           -> SEND a task to the mediator processor.
        STOP message received?
           -> DONE = TRUE;
Until done
---------------------------------------------------------------------------
```

Figure 3.4: Centralized Mediation

tributed receiver-initiated, distributed sender-initiated, and centralized mediation strategies are shown in figures 3.1, 3.2, 3.3, and 3.4, respectively.

3.2. Related Work

The centralized, master-slave organization has been used frequently in the design of parallel applications and can take various forms. For example, in the static synchronous global optimization algorithm described in [5], the master processor is responsible for partitioning the problem, starting the slave processors, synchronization, distributing local minimization tasks, and collecting results, but its scheduling decisions are trivial. In many other contexts, such as the parallel branch and bound algorithm of [27], the master has a complex role in scheduling due to the dynamic numbers

of tasks and the asynchronous character of these computations.

Centralized organizations are also prevalent in the implementation of parallel algorithms on shared memory systems where a shared task queue replaces the master processor and each processor sends all of its tasks to the centralized shared queue. Examples of this arrangement can be found in [7], [14], and [4], and many other systems.

At the systems support level, distributed dynamic load balancing has been an active area of research. The distributed strategies discussed here are similar to the sender-initiated and receiver-initiated policies (also referred to as bidding and drafting policies) investigated in [9], [8], [23], and [20]. Distributed policies that allow both a sender or a receiver to initiate a task transfer have also been investigated, such as the gradient model presented in [17]. In general, research at the systems support level assumes that different tasks come from different applications and are unrelated.

Dynamic load balancing also has been used at the environment and language level to perform load balancing specifically on those tasks involved in one parallel computation. Load balancing strategies in this situation can be similar to those mentioned above. Because the tasks involved are part of the same parallel computation, however, communication requirements may constrain which tasks can be moved from one processor to another. Some examples of environments and languages that employ or provide for dynamic load balancing are the Chare Kernel [12, 13], DIB [10], and the distributed scheduler described in [26].

Strategies related to our centralized mediation strategy have been used in very different contexts. First, a parallel branch and bound algorithm described in [15] uses a somewhat related approach. The processors in this algorithm inform a central scheduler when they need more work, but not when they have too much work. The scheduler obtains work to fill these requests by broadcasting a request to all the processors for more tasks, when it has run out of the tasks from its previous broadcast. Thus, this strategy relies on the centralized scheduler to determine when tasks should be sent from a processor to the centralized work queue, whereas in the centralized mediation strategy described in this paper, this decision is carried out locally, by each processor.

In [1], related ideas are used for two level scheduling in shared memory systems. Scheduling of *threads*, or fine grained tasks, is accomplished by keeping threads in queues local to processors in order to circumvent the waiting associated with accessing a single, centralized shared queue. When the size of a local queue exceeds some predetermined threshold, half of its threads are sent to the centralized shared queue to prevent load imbalance. A processor without threads in its local queue may obtain some from the centralized shared queue.

3.3. Implementation Details

In this section we describe how the scheduling strategies are used to implement the parallel algorithm for global optimization. In particular, for each strategy we describe how task priorities are determined, and the criteria used by processes to determine that they need more work or that they have too much work. First the definitions of global and local iterations are given.

3.3.1. 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 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 number of initial subregions is a multiple of the number of processors.) 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 at 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 at a processor with an iteration number $l < k$, where k is the last completed local iteration at the processor. When this occurs, that processor resets the last completed iteration to $l - 1$, and continues to schedule and execute tasks as before.

3.3.2. Task Scheduling Priorities

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 and centralized mediation at the mediator process): 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. Tasks of the same type and level are prioritized in the order of their arrival. 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. In all of the experiments to be described in Section 4, we have used an asynchronicity level of 2. Several values for the asynchronicity level were examined for these experiments and the value chosen exhibits the best performance overall.

3.3.3. Task Transfer Policies

The task transfer policies for the centralized mediation, distributed receiver, and distributed sender-initiated strategies consist of (1) a policy for giving work away and (2) a policy for determining if work is needed. In both cases, *application specific* criteria and *application independent* criteria are possible and have been examined in [24]. In this paper, however, we restrict our attention

to application specific policies. In general, a policy based on application independent information would use information such as that maintained by most operating systems for an estimate of workload, for example, the number of tasks waiting to execute. In contrast, an application specific policy incorporates some knowledge about the tasks or some other facet of the application.

A policy for determining whether work is needed is used in centralized mediation and distributed receiver-initiated scheduling to request work from the mediator or another processor, and in distributed sender-initiated scheduling to determine if work should be accepted. The application specific policy used is that requests are made or work is accepted 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 rather evenly, when possible.

A policy for giving work away is used in centralized mediation and distributed sender-initiated scheduling to send work voluntarily to the mediator or another processor, and in distributed receiver-initiated scheduling to determine if a request for work should be satisfied. The policy used is that 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.

In conjunction with the policies involved with task transfer, there is a *location* policy that determines which processor receives a task, or a request for a task. In both the distributed policies, the processor wanting to make a task request or send a task randomly selects another processor to receive this information. The receiving processor accepts the information if it can, or forwards the task or task request to another processor. This is called *probing*, and is carried out a limited number of times. In the centralized mediation and the centralized scheduling strategies, the location policy is a first-in-first-out policy, that sends the first task in the priority queue to the first processor that requests it.

4. Performance of the Scheduling Strategies

This section examines the performance and scalability of centralized scheduling, distributed scheduling and centralized mediation as applied to the parallel algorithm for global optimization. One concern is how well each strategy addresses the problem of dynamic task distribution for parallel adaptive computations under different types of task loading conditions. In particular, the success of a strategy is evaluated with respect to processor utilization and overall execution time for the global optimization algorithm for problems that produce light, medium, and heavy task workloads. Another concern is how the scheduling algorithms scale. In this case, we examine how each of the scheduling strategies performs as the number of processors and the problem size increases.

The performance of the three scheduling strategies is assessed using a combination of simulation, analytic modeling, and actual implementations of the parallel programs. First, a simulation model for the centralized scheduling strategy was developed, using execution traces and measurements from an existing implementation of centralized scheduling (see [25]). Next, the centralized scheduling simulation model was compared with performance measures from the actual implementation in order to validate the simulation model. This validated model for the centralized scheduler provided a basis for performing simulation experiments with the other scheduling strategies. Both distributed receiver-initiated scheduling and distributed sender-initiated scheduling, with both threshold and application specific task transfer policies were investigated, although in this paper only performance results for the receiver-initiated application specific task transfer policy (the best of these alternatives) are presented. The experimentation influenced the design of the scheduling strategies, leading to the consideration and development of the centralized mediation strategy and a simulation model for it.

The simulation models for centralized scheduling, distributed scheduling, and centralized mediation provide the opportunity to study a wide variety of situations such as different numbers of processors and different task loading conditions. Because there are possibly many interesting test cases to explore, we developed analytic models of centralized scheduling and centralized mediation to assist in identifying the most interesting test cases to examine by simulation. Finally, because

of the favorable performance results of centralized mediation in the simulation experiments, this strategy was chosen to verify the results of the modeling process. A parallel implementation of this strategy was developed, tested, and compared against the simulation results.

Section 4.1 describes the analytic models and their results, Section 4.2 describes the simulation methodology, including the use of the execution traces and the simulation results, and Section 4.3 describes the implementations used to validate the centralized scheduler and verify the centralized mediation simulation.

Both the implementations and the simulations discussed in these sections are evaluated using different configurations of and execution traces from a single test problem that is described in [25]. The advantage of using a single test problem is that we were able to make a detailed and controlled comparison of various aspects of the scheduling strategies, such as the number of tasks involved in the computation and the scalability of processors and task sizes. The disadvantage of using a single test problem is that the results may not be representative of the general behavior of the scheduling strategies. Although we believe that the experiments associated with this test problem have indicated some interesting general differences between the scheduling strategies, more extensive experiments will be necessary before general conclusions about the strategies can be made.

Finally, while we model communication costs accurately, we do not discuss their impact in the remainder of this section. The reason for this is that communication costs are a minor factor in these algorithms, since the average computation time of a task is usually 15 to 20 times larger than the average time required to send or receive a message.

4.1. Analytic Models

This section presents a simple queueing network model to describe the general behavior of the centralized scheduling and the centralized mediation implementations of the adaptive, asynchronous algorithm for parallel global optimization. The model allows us to characterize the performance differences between the two scheduling strategies in terms of the number of processors involved in the computation. This analysis has been used to guide the selection of the simulation experiments

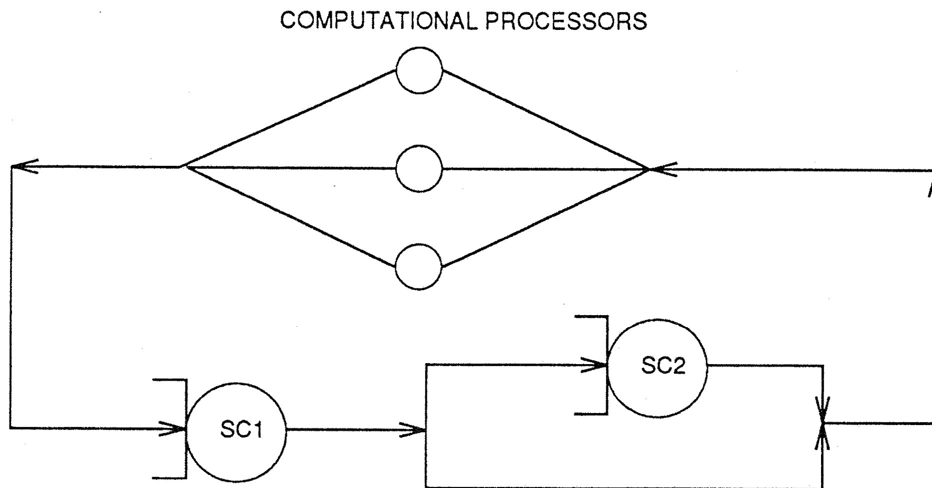


Figure 4.1: High level queueing network model for centralized strategies

described in Section 4.2.

The queueing network model that we describe is based on results from *operational analysis* [6], [16]. Both the centralized scheduler and the centralized mediator are modeled using a closed queueing network with two service centers and one infinite server. The general model used for both strategies is shown in Figure 4.1. One of the service centers, SC_1, represents the centralized scheduler or mediator, the other service center, SC_2, models any queueing delay incurred in scheduling, and the infinite server models the computational processors.

The queueing network models the behavior of a parallel computation in the following manner. The computational processors perform one or more tasks. Tasks in this case are the basic units of useful work in a parallel computation. After processing the tasks, a processor sends a "transaction" to the service center SC_1. In centralized scheduling, a transaction can be a request for a new task or a new task message, which announces that a new task is available. In the case of centralized mediation, a transaction can additionally be a "computation status" message, which informs the mediator of the current workload and computation progress at a processor. After the transaction at SC_1 is finished, some percentage of the request transactions are also processed at SC_2, which models the time spent waiting for a task to become available. A processor resumes computation after the transaction exits from either SC_1 or SC_2.

In general, in order to evaluate a closed queueing network model such as the one described here, a set of model inputs are given and, using the technique of mean value analysis [16], [21], a set of model outputs can be derived. These model outputs can then be used with other input parameters to express the total parallel computation time.

The model inputs are the number of processors (P), the demand at service centers SC_1 and SC_2 in Figure 4.1, and the "think" time (Z) of each processor. The values for the model inputs that we use in the analytic experiments are either measured from actual experiments or projected to reflect different situations of interest. In the case of the number of processors, P, we use several values between 8 and 64 to examine the scalability of the strategies. For expressing the demand at service center, SC_1, which represents the total amount of time spent processing transactions at the centralized scheduler or the centralized mediator, we have used values from actual experiments with the centralized scheduling implementation. The same values for SC_1 (485 ms for test problem A, and 615 ms for test problem B) are used in the analytic models for both centralized scheduling and centralized mediation, since the times these two models take to process transactions are expected to be similar. The demand at the second service center, SC_2, corresponds to a "task-matching" wait experienced with either the centralized scheduling or the centralized mediation strategy. We model the worst case situation in which a request for a task must always wait for a task to arrive to fill that request. The values used in the analytic experiments for the centralized scheduler (829 ms for test problem A and 1112 ms for test problem B) are average values obtained from measurements of the implementation for centralized scheduling, and the values used for the centralized mediation strategy (2437 ms for test problem A and 1585 ms for test problem B) are obtained from preliminary simulations. Finally, Z, the think time of the processors, is expressed in terms of the average task time, s, and the communication time, c, which are both measured experimentally. (For example, we obtained s = 9500 ms for problem A, s = 13300 ms for problem B, and c = 25 ms for both test problems). In centralized scheduling a transaction occurs after every task, so the think time is $Z = s + 2c$. In centralized mediation, the think time between transactions to the centralized mediator is considerably longer than in centralized scheduling since tasks are scheduled locally if

possible. We express this think time as $Z = ks + 2c$, where k is the number of tasks that execute in between transactions and where k is measured from the simulation of the centralized mediator with 8 processors, and is also used for larger numbers of processors. From our simulations, we obtained k = 2.6 and k= 1.8 for test problems A and B.

In addition to the inputs that are to be used directly in the calculation of the queueing model outputs, two additional parameters are used in the calculation of the parallel computation time. These parameters are n, the total number of tasks in the computation and t, the total number of transactions in the computation. In all the above models, n and t are measured experimentally from the centralized scheduling implementation for 8 processors (we obtained n = 151 and t = 53 for test problem A, and n=120 and t=65 for test problem B). To project situations where $P = 8K$ (for some integer $K > 1$), both n and t are multiplied by K. This has the effect of keeping the amount of work done by each processor the same for all values of P.

As mentioned earlier, we can calculate several model outputs using mean value analysis and the given model inputs for the queueing network model. The quantities of interest can then be derived from these outputs and the model inputs. In particular we are interested in the overall execution time of the parallel computation, the utilization of the centralized processor (service center SC_1), and the utilization of the computational processors.

For both the centralized scheduling and the centralized mediation strategies, the overall execution time for each processor is the time spent by each processor doing useful work plus the time each processor spends waiting. In general, each processor may execute different numbers of tasks with varying computational requirements. To make the model as simple as possible, however, we make the assumption that each processor executes the same number of tasks ($\frac{n}{P}$). The computational time used for each of these tasks will be the average computation time, s, as measured in the centralized scheduling experiments, so that the total time that a processor spends doing useful work is $\frac{n}{P}s$. The total time spent waiting for transactions to complete is the sum of the communication time to and from the service centers and the time resident while being serviced there, or $\frac{t}{P}(2c + R_{sys})$, where R_{sys} is a model output calculated using mean value analysis

Table 4.1: Centralized Scheduler Analytic Experiment Results

	Problem A			Problem B		
P	OVERALL	U_{sched}	Ucomp	OVERALL	U_{sched}	Ucomp
8	4:11	29 %	71 %	5:28	27 %	73 %
16	3:52	64 %	78 %	4:01	59 %	79 %
24	4:06	90 %	73 %	5:11	85 %	76 %
32	4:57	99 %	61 %	6:00	98 %	66 %
40	6:08	99 %	49 %	7:22	99 %	54 %
48	7:22	100 %	41 %	8:51	100 %	45 %
56	8:36	100 %	35 %	10:19	100 %	38 %
64	9:49	100 %	31 %	11:48	100 %	33 %

that represents the mean time a transaction spends at service center SC_1 and SC_2, and t is the total number of transactions. We also assume that each processor executes the same number of transactions. The overall time of the computation is thus given by

$$OVERALL = \frac{n}{P}s + \frac{t}{P}(2c + R_{sys}).$$

The utilization for the processors is the ratio of the time spent doing useful work to the overall time. That is, the average utilization of a processor is

$$U_{processor} = \frac{ns}{ns + t(2c + R_{sys})}.$$

These equations characterize the general behavior of both the centralized scheduling and the centralized mediation strategies. The differences between the analytic model for two strategies are the values of SC_2 and k, and the percentage of tasks that are serviced as requests at SC_2.

A number of cases of the analytic model were run to examine the scalability of each scheduling strategy, and to identify interesting cases for use in the simulation study. This was done by varying the number of processors, P in the model. In particular we wanted to determine the number of processors that place the centralized scheduler or the centralized mediator in a "heavily loaded" state, which occurs when their utilizations approach 100%. The values of P examined are multiples of 8, from 8 to 64.

The results of the model for two problems are given in Tables 4.1 and 4.2. $OVERALL$, U_{sched}, U_{med} and U_{comp} represent the overall execution time, the utilization of the centralized sched-

Table 4.2: Centralized Mediation Analytic Experiment Results:

	Problem A			Problem B		
P	OVERALL	U_{med}	U_{comp}	OVERALL	U_{med}	U_{comp}
8	3:33	11 %	85 %	4:22	9 %	91 %
16	3:22	24 %	89 %	4:16	19 %	93 %
24	3:19	37 %	90 %	4:15	30 %	94 %
32	3:18	50 %	91 %	4:14	40 %	94 %
40	3:17	64 %	91 %	4:14	51 %	94 %
48	3:18	76 %	91 %	4:14	61 %	94 %
56	3:19	86 %	90 %	4:14	71 %	94 %
64	3:23	95 %	88 %	4:14	80 %	94 %

uler processor, the utilization of the centralized mediation processor, and the average utilization of the computational processors, respectively.

We draw the following conclusion from these results. The degradation of performance is more gradual in the case of centralized mediation because there are far fewer transactions to the centralized mediator than there are to the centralized scheduler. The centralized scheduler becomes heavily loaded around 32 processors, and the centralized mediator around 64 processors or more. These results are reflected in the choice of the numbers of processors examined for the simulation experiments.

4.2. Simulation

This section describes the simulation methodology and performance results for the three scheduling strategies as applied to the parallel global optimization algorithm.

To model the execution of the adaptive parallel algorithm using various scheduling strategies, we have used simulations based on the execution traces generated by an implementation of the parallel global optimization algorithm using the centralized scheduling strategy. The execution traces were obtained from experimental runs for different problems running on up to 8 slave processors. The execution traces consist of computation times for the various steps of the algorithm, and information about what is happening in each subregion, such as the number of local searches generated at each iteration, the number of subregions, and the sample size of each subregions. This

information is used in discrete event simulations for each of the scheduling strategies.

As mentioned at the beginning of the section, the simulations will address two issues to assess the performance of the various scheduling strategies. The first issue is how well the scheduling strategies distribute tasks under different loading conditions. The second issue is how well the scheduling strategies scale. In order to investigate these issues using simulation, the execution traces have been modified.

To investigate different loading conditions, the execution traces have been modified to represent light, medium, and heavy loading. Lightly loaded conditions might occur, for example, if there were fewer minimizers in the test problem, or if the function evaluations of this test problem were more expensive. Conversely, heavily loaded conditions might occur if there were more minimizers in the test problem, or if the function evaluations of this test problem were less expensive. For light loading, the traces have been modified 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 are not enough tasks to keep all the processors busy. In a similar manner, we have modified the traces to depict heavy loading conditions by duplicating productive branches of the task precedence tree. This modification results in an abundance of work being available to the processors. No modification is necessary to observe the behavior of medium loaded conditions.

To investigate how well the scheduling strategies scale, each set of traces has been duplicated (in multiples of 8) to examine the behavior of the algorithms on as many as 64 processors.

The simulation results presented here show the behavior of the global optimization algorithm for 3 different scheduling strategies, centralized scheduling, centralized mediation, and distributed receiver-initiated scheduling, as applied to two versions of a single global optimization test problem (called A and B) under various workload and processor combinations. (The two versions differ only in the initial sample density). Preliminary results indicated that the distributed sender-initiated strategy performed worse than the distributed receiver-initiated strategy, so we did not experiment further with this strategy. The main objective of this evaluation is to contrast the performance characteristics of centralized mediation with the pure centralized and distributed

strategies.

There are three tables for each of the two problems, organized as follows. The first table 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 table presents the simulation results for a medium number of tasks in the system, enough work to go around if carefully distributed. The final table presents the results for a large number of tasks and abundant work in the system.

Each table contains the simulation results for each scheduling strategy for 8, 16, 32, and 64 processors. For each case it shows the mean computation times, the 95% confidence intervals for the computation times, and the range of processor utilizations. For centralized scheduling and centralized mediation, it also shows the master utilization, which is the utilization of the processor running the centralized scheduler or centralized mediator process.

4.2.1. Centralized Scheduling vs. Centralized Mediation

First consider the situation in which a small number of tasks are created (Tables 4.3 and 4.4). For 8 processors, the confidence intervals show conclusively that the centralized mediator strategy is slower on both test problems. The range of computational processor utilizations with centralized scheduling indicates that there is not enough work to keep the computational processors occupied all the time. The range of computational processor utilizations for the centralized mediator show that it is not as effective at distributing the tasks. For 16 processors, centralized scheduling is, on average, faster for problem A and slower for problem B. Under the same loading conditions with 32 and 64 processors, the centralized mediator always performs better than the centralized scheduler. In these cases, the differences between the two scheduling strategies are significant since there is no overlap in their confidence intervals.

For medium numbers of tasks (Tables 4.5 and 4.6) and large numbers of tasks (Tables 4.7 and 4.8) in the system, there is no significant difference between the performance of the two scheduling strategies at 8 processors, as reflected by the large overlap in the confidence intervals for these situations. For 16 or more processors, however, the advantage of centralized mediation

is apparent, and this strategy can be significantly faster (up to 4.3 times as fast in some cases) than centralized scheduling. The centralized scheduling processor becomes a bottleneck at 32 or 64 processors for both problems in both the medium and heavily loaded situations, while the centralized mediator is no more than 33 % and 67 % loaded at 32 and 64 processors, respectively, in any of these cases.

In summary, the centralized scheduler becomes a bottleneck considerably more quickly than the centralized mediator, and as a result, experiences significant performance degradation. The centralized scheduler is superior only for a light workload and a very small number of processors.

Table 4.3: Comparison of scheduling techniques for problem A tests involving small numbers of tasks for 8, 16, 32, and 64 processors

Number of processors	Scheduling Strategy	Comp. time (Mean)	Comp. time (95 % conf. int.)	Processor util (Range)	Master util (Mean)
8	centralized scheduler	2:41	2:35 - 2:46	50 % - 81 %	38 %
	centralized mediator	3:40	3:22 - 4:23	24 % - 90 %	11 %
	distr. receiver	4:03	3:46 - 4:20	23 % - 76 %	-
16	centralized scheduler	2:27	2:22 - 2:34	42 % - 78 %	32 %
	centralized mediator	2:51	2:37 - 3:04	25 % - 95 %	21 %
	distr. receiver	3:18	3:14 - 3:38	21 % - 74 %	-
32	centralized scheduler	3:52	3:41 - 4:07	26 % - 54 %	87 %
	centralized mediator	2:53	2:42 - 3:04	26 % - 92 %	41 %
	distr. receiver	3:24	3:21 - 3:44	20 % - 77 %	-
64	centralized scheduler	6:57	6:07 - 7:45	14 % - 29 %	94 %
	centralized mediator	3:14	2:36 - 3:51	23 % - 89 %	76 %
	distr. receiver	3:37	3:18 - 3:55	18 % - 73 %	-

4.2.2. Centralized Mediation vs. Distributed Receiver-Initiated Scheduling

We first note that in comparing the centralized mediation and the distributed receiver-initiated scheduling strategies, we have, for convenience, always used one additional processor for the centralized mediator strategy than for the distributed receiver strategy. This processor is dedicated to the centralized mediator process. By examining the simulation results, we see that the work done by this processor is fairly small in all cases, amounting to approximately 1 % of the total computational effort. In comparison to the performance difference between the centralized mediator and the distributed receiver strategies, we therefore see that combining this processor with a

Table 4.4: Comparison of scheduling techniques for problem B tests involving small numbers of tasks for 8, 16, 32, and 64 processors

Number of processors	Scheduling Strategy	Comp. time (Mean)	Comp. time (95 % conf. int.)	Processor util (Range)	Master util (Mean)
8	centralized scheduler	3:06	2:56 - 3:15	67 % - 90 %	43 %
	centralized mediator	4:30	4:09 - 4:53	40 % - 94 %	8 %
	distr. receiver	4:43	4:20 - 5:06	33 % - 82 %	-
16	centralized scheduler	3:15	3:05 - 3:26	59 % - 84 %	67 %
	centralized mediator	2:59	2:48 - 3:01	54 % - 95 %	18 %
	distr. receiver	3:34	3:11 - 3:43	42 % - 80 %	-
32	centralized scheduler	5:37	5:05 - 5:39	34 % - 50%	85 %
	centralized mediator	2:56	2:46 - 3:01	55 % - 96 %	34 %
	distr. receiver	3:47	3:21 - 3:50	36 % - 96 %	-
64	centralized scheduler	9:55	8:58 - 9:21	18 % - 30%	92 %
	centralized mediator	3:20	3:03 - 3:36	50 % - 91 %	61 %
	distr. receiver	3:43	3:04 - 4:20	33 % - 79 %	-

computational processor (thus giving each strategy an equal number of processors) would not be expected to alter the overall comparison between the strategies.

In comparing the two strategies, we rely heavily on the confidence interval results to discern significant differences between the two strategies. For the majority of the 8 processor experiments, there is some overlap in the confidence intervals of the computation times for the two strategies. Because of this and the fact that the centralized mediator strategy uses one additional processor, there is no significant difference between the two strategies even though the computation mean times are lower for centralized mediation in all these cases.

For 16 and 32 processors, there are some clear differences between the two strategies. In the situation where small and medium numbers of tasks are present in the system (see Tables 4.3, 4.4, 4.5 and 4.6), centralized mediation exhibits a performance gain that ranges from 14% to 28 %, with no overlap in the computation time confidence intervals of the distributed receiver strategy. With large numbers of tasks in the system, the centralized mediation strategy exhibits significantly faster performance than the distributed receiver strategy in problem B; for problem A the mean computation time for centralized mediation is lower but the computation time confidence intervals overlap (see Tables 4.7 and 4.8). For 64 processors, centralized mediation is significantly faster than distributed receiver in problem B for medium and large numbers of tasks. Under all other

Table 4.5: Comparison of scheduling techniques for problem A tests involving medium numbers of tasks for 8, 16, 32, and 64 processors

Number of processors	Scheduling Strategy	Comp. time (Mean)	Comp. time (95 % conf. int.)	Processor util (Range)	Master util (Mean)
8	centralized scheduler	3:30	3:16 - 3:44	77 % - 95 %	51 %
	centralized mediator	3:08	3:01 - 3:25	62 % - 96 %	7 %
	distr. receiver	3:36	3:14 - 3:39	66 % - 92 %	-
16	centralized scheduler	4:02	3:51 - 4:36	64 % - 84 %	82 %
	centralized mediator	3:11	3:05 - 3:26	59 % - 97 %	15 %
	distr. receiver	3:52	3:28 - 4:18	59 % - 91 %	-
32	centralized scheduler	6:57	6:27 - 8:47	33 % - 52 %	94 %
	centralized mediator	3:18	3:11 - 3:34	58% - 97 %	20 %
	distr. receiver	3:55	3:34 - 4:12	52 % - 91 %	-
64	centralized scheduler	13:31	11:07 - 15:54	16 % - 28 %	97 %
	centralized mediator	3:34	3:00 - 4:09	53% - 95 %	63 %
	distr. receiver	4:08	3:28 - 4:48	43 % - 85 %	-

conditions, although the mean computation times of the centralized mediator strategy are lower than for the distributed receiver strategy, the confidence intervals of the computation times overlap enough so that the performance improvements may not be significant.

When the centralized mediator strategy does exhibit better performance than the distributed receiver strategy, a key reason is that the processor utilizations are generally higher for centralized mediation than for distributed receiver under all loading conditions. This is because the productive utilization of each computational processor in either strategy is limited by the amount of overhead it incurs in scheduling. In the distributed strategy, as the number of processors increases, the number of requests in the system usually increases linearly with the number of processors, while the number of computation status messages increases proportional to square of the number of processors. In the centralized mediation strategy, the centralized mediator incurs most of the overhead of scheduling, and the increase in scheduling overhead for the computational processors is far lower. Thus, the computational processors realize higher processor utilizations in this strategy.

It is also interesting to compare these results to those predicted by the analytic models. Although the analytic models show the centralized mediator beginning to saturate at 64 processors, the worst processor utilization time observed by the mediator at 64 processors in the simulations is 76 %. (Recall that the analytic model made some worst case assumptions regarding the latency

Table 4.6: Comparison of scheduling techniques for problem B tests involving medium numbers of tasks for 8, 16, 32, and 64 processors

Number of processors	Scheduling Strategy	Comp. time (Mean)	Comp. time (95 % conf. int.)	Processor util (Range)	Master util (Mean)
8	centralized scheduler	4:14	4:00 - 4:28	86 % - 98 %	46 %
	centralized mediator	3:46	3:45 - 4:03	83 % - 97 %	7 %
	distr. receiver	4:23	4:03 - 4:49	86 % - 98 %	-
16	centralized scheduler	5:08	4:41 - 5:16	74 % - 89 %	78 %
	centralized mediator	3:43	3:37 - 3:58	83% - 97 %	15 %
	distr. receiver	5:09	4:36 - 5:13	88 % - 92 %	-
32	centralized scheduler	8:43	7:50 - 8:47	39 % - 56 %	90 %
	centralized mediator	3:38	3:33 - 3:57	82 % - 99 %	17 %
	distr. receiver	4:57	4:23 - 5:16	77 % - 93 %	-
64	centralized scheduler	16:04	15:01 - 17:06	19 % - 32 %	95 %
	centralized mediator	3:46	3:40 - 3:51	80 % - 99 %	59 %
	distr. receiver	5:21	4:27 - 6:13	73 % - 89 %	-

between new task messages arriving to the centralized mediator; these presumably account for the differences.) This is under conditions that place the most strain on the centralized mediator, namely, there are small numbers of tasks in the system and not enough work to go around, so that many requests are made to the mediator.

In summary, the centralized mediator never performs significantly worse than the distributed scheduler in these tests, and performs significantly better in a number of cases. The centralized mediator is not yet a bottleneck at 64 processors in these tests, although it would becomes one for P sufficiently large.

4.3. Validation and Verification of the Simulation Models

This section presents the experimental results for validating the centralized scheduler simulation model and for verifying the centralized mediation model.

For the implementations that are discussed here, we ran experiments for the centralized scheduling and centralized mediation implementations using a dedicated test environment of 9 SUN 3/60 workstations (8 computational processors plus 1 master processors) for each test problem. The applications were implemented using Grail, a set of message passing library routines developed by Maybee [18], with primitives similar to those provided by PVM [2] and P4 [11]. In both tests

Table 4.7: Comparison of scheduling techniques for problem A tests involving large numbers of tasks on 8, 16, 32, and 64 processors

Number of processors	Scheduling Strategy	Comp. time (Mean)	Comp. time (95 % conf. int.)	Processor util (Range)	Master util (Mean)
8	centralized scheduler	3:48	3:36 - 3:58	82 % - 95 %	47 %
	centralized mediator	3:03	3:00 - 3:27	77 % - 98 %	7 %
	distr. receiver	3:34	3:20 - 3:47	75 % - 96 %	-
16	centralized scheduler	4:18	4:11 - 4:37	60 % - 85 %	80 %
	centralized mediator	3:04	2:59 - 3:32	75 % - 99 %	14 %
	distr. receiver	3:45	3:20 - 4:10	71 % - 95 %	-
32	centralized scheduler	7:15	6:54 - 7:48	37 % - 54 %	94 %
	centralized mediator	3:17	3:10 - 3:48	67 % - 98 %	33 %
	distr. receiver	3:36	3:28 - 4:10	68 % - 93 %	-
64	centralized scheduler	13:55	12:53 - 14:56	18 % - 30 %	97 %
	centralized mediator	3:19	2:54 - 3:43	69 % - 97 %	67 %
	distr. receiver	3:44	3:17 - 4:10	62 % - 89 %	-

shown here, the experimental results for each problem are averaged over 10 runs. The performance comparisons shown in Tables 4.9 and 4.10 are the mean and 95 % confidence interval of the system time, and the range of processor utilizations.

4.3.1. Validation of the Centralized Scheduler

We have validated the centralized scheduler simulation model by comparing the simulation based upon the unmodified execution traces derived from the experimental runs to the actual experimental runs. Table 4.9 shows some of the comparisons. The simulation computation times are very close to the times observed in the actual implementations. There is also a large overlap in the confidence intervals of the computation times indicating that there are no statistically significant differences between the simulation results and the implementation results. These results show that a discrete event simulation can be constructed from execution traces that closely agrees with actual runs of the program. This observation led to the initial decision to use simulation as a means to explore different scheduling strategies and parallel environments for the adaptive asynchronous global optimization algorithm.

Table 4.8: Comparison of scheduling techniques for problem B tests involving large numbers of tasks on 8, 16, 32, and 64 processors

Number of processors	Scheduling Strategy	Comp. time (Mean)	Comp. time (95 % conf. int.)	Processor util (Range)	Master util (Mean)
8	centralized scheduler	4:02	3:58 - 4:04	87 % - 94 %	44 %
	centralized mediator	3:43	3:33 - 4:05	85 % - 98 %	7 %
	distr. receiver	4:59	4:53 - 5:03	88 % - 97 %	-
16	centralized scheduler	4:46	4:37 - 4:51	74 % - 89 %	74 %
	centralized mediator	3:34	3:26 - 4:03	84 % - 97 %	14 %
	distr. receiver	5:09	4:33 - 5:18	88 % - 96 %	-
32	centralized scheduler	7:45	7:28 - 7:53	40 % - 59 %	89 %
	centralized mediator	3:36	3:32 - 3:59	85 % - 99 %	30 %
	distr. receiver	5:13	4:49 - 5:18	85 % - 96 %	-
64	centralized scheduler	14:37	14:06 - 15:07	19 % - 32 %	94 %
	centralized mediator	3:44	3:25 - 4:03	78 % - 99 %	58 %
	distr. receiver	5:20	5:06 - 5:33	83 % - 94 %	-

Table 4.9: Comparison of the simulation and implementation results for the centralized scheduling strategy

Test Case	Evaluation Strategy	Computation Time (mean)	Computation Time (95 % conf. int.)	Processor Util (range)	Master Util (mean)
A	simulation	3:30	3:16 - 3:44	77 % - 95 %	51 %
	implementation	3:25	3:02 - 3:32	82 % - 91 %	54 %
B	simulation	4:14	4:00 - 4:28	86 % - 98 %	46 %
	implementation	4:05	3:46 - 4:17	95 % - 97 %	54 %

4.3.2. Verification of Centralized Mediation

The scheduling extensions made to the original simulation model were verified by implementing the centralized mediation strategy on a network of workstations. A representative subset of the simulation results and the parallel implementation results are shown in Table 4.10. In both test problems, the complete implementation execution times are, on average, within 6% of the simulation times. The implementation processor utilizations are, in all cases, slightly higher than the processor utilizations in the simulation. This may indicate that the centralized mediator strategy obtains better load balance than predicted by the simulations. The general trends in load balancing are estimated with reasonable accuracy by the simulations, however, since the computation times and processor utilization ranges for both the simulations and the implementations are comparable in

Table 4.10: Comparison of simulation results and implementation results for the centralized mediation strategy

Test Case	Evaluation Strategy	Computation Time (mean)	Computation Time (95 % conf. int.)	Processor Util (range)	Master Util (mean)
A	simulation	3:08	3:01 - 3:25	62 % - 96 %	7 %
	implementation	3:14	2:57 - 3:28	83 % - 96 %	4 %
B	simulation	3:46	3:45 - 4:03	83 % - 97 %	7 %
	implementation	3:32	3:14 - 3:49	89 % - 99 %	3%

all test problems.

The mediator utilizations in both implementation cases are considerably smaller than predicted by the simulation. Since all the estimates for the service time by the centralized mediator that were used in the simulations were based on measurements from the centralized scheduler, it is possible that these measurements were an overestimation of the centralized mediator's actual service requirement. To investigate the implications of this overestimation, we reran the analytic and simulation models for the centralized mediator with a reduced service time. The system execution times did not change significantly for the situations examined in this paper.

5. Summary

This paper has described several dynamic scheduling strategies and their implementation for a parallel adaptive algorithm for the global optimization problem. The adaptive and asynchronous features of the parallel algorithm are used to enhance the performance of the computation, but they complicate the implementation of the algorithm on distributed memory computers, thus necessitating the use of dynamic scheduling strategies. We described how three scheduling strategies, centralized scheduling, distributed scheduling, and centralized mediation, are used to implement the parallel global optimization algorithm.

We have evaluated the three different scheduling strategies using a combination of simulation, implementation, and analytic modeling. Not only were these evaluation techniques useful for assessing the performance of the strategies, but they were also instrumental in the development of a new scheduling strategy. We initially considered two strategies, centralized and distributed

scheduling, but the simulations using these strategies led us to develop and consider the centralized mediation strategy.

The performance evaluation of these three scheduling strategies focused on the scalability of the approaches in the context of the global optimization algorithm. The centralized mediation often exhibited the best performance of the scheduling strategies at 16 and 32 processors, and shared the best performance with the distributed receiver strategy at 64 processors. At 64 processors, however, the utilization of the centralized mediator is always above 50 %, so the scalability of this strategy, like any other centralized strategy, is limited. The overhead costs associated with distributed scheduling also increase as the number of processors increases, thus limiting the scalability of this approach as well. As expected, the centralized scheduler saturates with far fewer processors than the centralized mediator or distributed strategy. Thus, the centralized mediator or distributed strategies are best suited to scale with the number of processors, but either would need to be implemented in a hierarchical manner for sufficiently large numbers of processors.

Finally, the scheduling strategies for centralized scheduling and centralized mediation were validated and verified with actual parallel implementations. These comparisons indicated that the simulation experiments are a reliable indicator of the performance of the scheduling algorithms.

It is important to mention that we believe that centralized mediation is likely to be an easier strategy to implement and debug than the distributed receiver strategy. First, the distributed receiver strategy requires that processors receiving task messages be interrupted to determine in a timely fashion whether there is work to give away. This is not necessary in the centralized mediation strategy since the mediator is dedicated to processing this type of information. Second, because the new task, task request, and computation status messages in the centralized mediation are sent to a centralized location, it is easier to isolate programming bugs than in the distributed strategy, where any processor may receive these messages. Thus we feel that the centralized mediation strategy may be a promising scheduling strategy for dynamic, adaptive parallel computations.

References

[1] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN. 37831-6367, September 1991.

[3] M.J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 54:484–512, 1984.

[4] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Conference on Parallel Programming: Experience and Applications*, pages 1 – 9, July 1988.

[5] 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.

[6] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *Computing Surveys*, 10(3):225–261, September 1978.

[7] J. J. Dongarra and D. C. Sorenson. Schedule: Tools for developing and analyzing parallel fortran programs. In D. B. Gannon, L.H. Jamieson, and R. J. Douglass, editors, *Characteristics of Parallel Algorithms*, pages 363 – 394. MIT Press, 1987.

[8] 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.

[9] D. L. Eager, E.D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[10] R. Finkel and U. Manber. Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9:235–256, 1987.

[11] J.Boyle, R.Butler, T.Disz, B.Glickfeld, E.Lusk, R.Overbeek, J.Patterson, and R.Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, Winston, Inc., New York, NY, 1987.

[12] L. V. Kale and W. Shu. The chare-kernel language for parallel programming: A perspective. Technical Report UIUCDCS-R-88-1451, Department of Computer Science - University of Illinois at Urbana-Champaign, August 1988.

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

[14] J. A. Kapenga and E. De Donker. A parallelization of adaptive task partitioning algorithms. *Parallel Computing - North-Holland*, 7:211–225, 1988.

[15] G. Kindervater. *Exercises in Parallel Combinatorial Computing*. PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1989.

[16] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, New Jersey, 1984.

[17] 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*, pages 329–336, August 1986.

[18] P. Maybee. Grail: A system for parallel distributed programming. Technical report, University of Colorado, Boulder, Colorado 80309-0430, 1989.

[19] 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, Denver, CO 80204, October 1988. To appear in *Parallel Computing.*

[20] R. Mirchandaney, D. Towsley, and J. A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.

[21] M. Reiser and S. S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of the ACM*, 27(2):313–322, April 1980.

[22] 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.

[23] A. Ross and B. McMillin. Experimental comparison of bidding and drafting load sharing protocols. In *The Fifth Distributed Memory Computing Conference*, pages 968–974, April 1990.

[24] S. L. Smith. *Adaptive, Asynchronous Distributed Algorithms for Parallel Computation.* PhD thesis, University of Colorado, Boulder, Colorado 80309-0430, December 1991.

[25] S. L. Smith, Elizabeth Eskow, and Robert B. Schnabel. Adaptive, asynchronous stochastic global optimization algorithms for sequential and parallel computation. In T. F. Coleman and Y. Li, editors, *Proceedings of the Workshop on Large-Scale Numerical Optimization*, pages 207 - 227. SIAM, Philadelphia, 1989.

[26] D. A. Tanqueray and D. F. Snelling. A distributed self-scheduler for partially ordered tasks. *Parallel Computing - North-Holland*, 8:267–273, 1988.

[27] H.W.J.M. Trienekens. *Parallel Branch and Bound Algorithms.* PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1990.