Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers

Wei Shu and Min-You Wu

Department of Computer Science

State University of New York at Buffalo

Buffalo, NY 14260

shu,wu@cs.buffalo.edu

Abstract— Runtime Incremental Parallel Scheduling (RIPS) is an alternative strategy to the commonly used dynamic scheduling. In this scheduling strategy, the system scheduling activity alternates with the underlying computation work. RIPS utilizes the advanced parallel scheduling technique to produce a low-overhead, high-quality load balancing, as well as adapting to irregular applications. This paper presents methods for scheduling a single job on a dedicated parallel machine.

Index Terms— runtime load balancing, incremental scheduling, parallel scheduling, irregular and dynamic applications, distributed memory computers.

Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers

1. Introduction

One of the challenges in programming distributed memory parallel machines is to schedule work to processors. There are two types of application problem structures: problems with a predictable structure, also called *static problems*, and problems with an unpredictable structure, called *dynamic problems*. Solving dynamic problems is difficult since the number of tasks and the grain size of a task may not be known prior to execution. Many applications do have such dynamic and irregular features, and therefore fall into this difficult category.

There are two basic scheduling strategies: *static scheduling* and *dynamic scheduling*. Static scheduling distributes the work load before runtime, and can be applied to static problems. Most existing static scheduling algorithms are sequential, executed on a single processor system. Dynamic scheduling performs scheduling activities concurrently at runtime and applies to dynamic problems. Although dynamic scheduling can apply to static problems as well, static scheduling is usually used for static problems because it provides a more balanced load distribution than dynamic scheduling.

Static scheduling utilizes the knowledge of problem characteristics to reach a global optimal, or nearly-optimal, solution. It has recently attracted considerable attention among the research community. The quality of scheduling relies heavily on accuracy of weight estimation. Scalability of static scheduling is restricted since a large memory space is required to store the task graph. In addition, it is not able to balance the load for dynamic problems.

Dynamic scheduling has certain advantages. It is a general approach suitable for a wide range of applications. It can adjust load distribution based on runtime system load information. However, most runtime scheduling algorithms, when making a load balancing decision, utilize neither characteristics information of application problems nor global load information. Efforts to collect load information for a scheduling decision certainly compete resources with the underlying computation during runtime. System stability usually sacrifices both quality and quickness of load balancing.

It is possible to design a scheduling strategy that combines the advantages of static and dynamic scheduling. This scheduling strategy should be able to generate a well-balanced load without incurring large overhead. With advanced parallel scheduling techniques, this ideal scheduling

becomes feasible. In parallel scheduling, all processors cooperate together to schedule work. Some parallel scheduling algorithms have been introduced in [1, 2, 3, 4, 5]. Parallel scheduling is stable because of its synchronous operation. It uses global load information stored at every processor and is able to accurately balance the load. As an alternative strategy to the commonly used dynamic scheduling, parallel scheduling opens a new direction for runtime load balancing.

In this paper, we propose a new method, called *Runtime Incremental Parallel Scheduling (RIPS)*. RIPS is a runtime version of global parallel scheduling. In RIPS, the system scheduling activity alternates with the underlying computation work during *runtime*. Tasks are *incrementally* generated and scheduled in *parallel*. The RIPS system paradigm is shown in Figure 1. A RIPS system starts with a system phase which schedules initial tasks. It is followed by a user computation phase to execute the scheduled tasks, and possibly generate new tasks. In the second system phase, the old tasks that have not been executed will be scheduled together with the newly generated tasks. This process will repeat iteratively until the entire computation is completed. Note that we assume the Single Program Multiple Data (SPMD) programming model, therefore, we rely on a uniform code image accessible at each processor. In addition, we assume that jobs to be executed are computation-intensive and can be partitioned into medium grain-size tasks to ensure that the cost to migrate a task is less than its execution time.

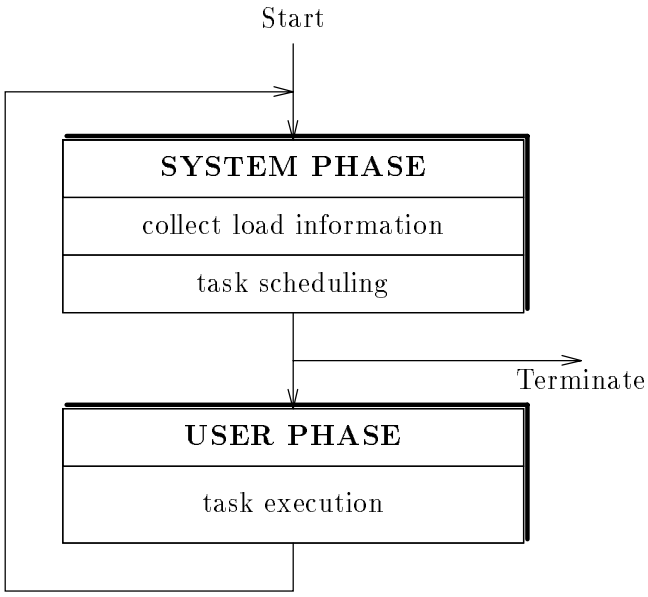


Figure 1: Runtime Incremental Parallel Scheduling (RIPS).

RIPS is a general approach. It can be used for a single job on a dedicated machine or a multiprogramming environment. It can be applied to both shared memory and distributed memory machines. Algorithms for scheduling a single job on a dedicated distributed memory machine are described in this paper. Previous works on runtime scheduling are simply reviewed

in the next section. Section 3 is devoted to the issues of incremental scheduling and Section 4 to the parallel scheduling algorithms. Experimental study and comparisons are presented in Section 5.

2. Previous Works

RIPS shares some common ideas with static scheduling [6, 7, 8, 9, 10]. Both of them utilize the systemwide information and perform scheduling globally to achieve high quality of load balancing. They also clearly separate scheduling and user computation. But, RIPS is different from static scheduling in three aspects. First, the scheduling activity is performed at runtime, therefore it can deal with dynamic problems. Second, the possible load imbalance caused by inaccurate grain-size estimation can be corrected by the next turn of scheduling. Third, it eliminates the requirement of large memory space to store task graphs, as scheduling is conducted in an incremental fashion. It then leads to better scalability for massively parallel machines and large size applications.

RIPS is similar to dynamic scheduling to some extent. Both methods schedule tasks at runtime instead of compile-time. Their scheduling decisions, in principle, depend on and adapt to the runtime system information. However, RIPS and dynamic scheduling are substantially different and should be considered as two separate categories. First, system functions and user computation are mixed together in dynamic scheduling, but there is a clear cutoff between system and user phases in RIPS, which can potentially offer easy management and low overhead. Second, placement of a task in dynamic scheduling is basically an individual action by a processor based on partial system information. Whereas, the scheduling activity in RIPS is always an aggregate operation based on global system information. The major characteristics of the three categories are summarized below.

	Static scheduling	Dynamic scheduling	RIPS
Load information	global	partial	global
Scheduling	compile time	runtime	runtime
Adaptive to system load	no	yes	yes
Scheduling & computation separation	yes	no	yes
Scheduling overhead	large	small	small
Storage requirement	large	small	small

Large research efforts have been directed towards the process allocation in distributed systems [11, 12, 13, 14, 15, 16, 17, 18, 19]. Eager *et al.* compared the sender-initiated algorithm with receiver-initiated algorithm [13]. A recent comparison study of dynamic load balancing strategies

on highly parallel computers is given by Willebeek-LeMair and Reeves [20]. The sender-initiated diffusion (SID), receiver-initiated diffusion (RID), and hierarchical balancing method (HBM) are also described in [20]. Work with a similar assumption includes the Gradient Model developed by Lin and Keller [21]. The randomized allocation algorithms developed by different authors are quite simple and effective [22, 12, 23, 24].

In the following, we describe three dynamic scheduling algorithms which will be compared to RIPS. A randomized allocation strategy dictates that each processor, when it generates a new task, should send it to a randomly chosen processor [22, 12]. The major advantages of this strategy are its simplicity and topology independence. No local load information needs to be maintained, nor is any load information sent to other processors. Statistical analysis shows that randomized allocation has a respectable performance. However, a couple factors may degrade the performance of the randomized allocation. First, the grain sizes of tasks may vary. Even if each processor processes approximately the same number of tasks, the load on each processor may be uneven. Secondly, lack of locality leads to large overhead and communication traffic. Only $1/N$ subtasks stay on the creating processor, where N is the number of processors in the system. Thus, most messages between tasks have to cross processor boundaries. The average distance traveled by messages is the same as the average interprocessor distance of the system. This leads to a higher communication load on large systems. Since the bandwidth consumed by a long-distance message is certainly larger, the system is more likely to be communication bound, as compared to a system using other load balancing strategies that encourage locality.

In the gradient model [21], instead of trying to allocate a newly-generated task to other processors, the task is queued at the generating processor and waits for some processor to request it. A separate, asynchronous process on each processor is responsible for balancing the load. This process periodically updates the *state* function and *proximity* on each processor. The *state* of a processor is decided by two parameters: the *low water mark* and *high water mark*. If the load is below the *low water mark*, the state is *idle*. If the load is above the *high water mark*, the state is *abundant*. Otherwise, it is *neutral*. The proximity of a processor represents an estimate of the shortest distance to an idle processor. An idle processor has a proximity of zero. For all other processors, the proximity is one more than the smallest proximity among the nearest neighbors. If the calculated proximity is larger than the network diameter, it is in the saturation state and the proximity is set to be *network_diameter+1* to avoid unbounded increase in proximity values. If the calculated proximity is different from the old value, it is broadcast to all the neighbors. Based on the state function and the proximity, this strategy is able to balance the load between processors. When the state is abundant and not in the saturation state, the processor sends a task from its local queue to the neighbor with the least proximity.

The receiver-initiated diffusion (RID) algorithm is a near-neighbor diffusion approach which

employs overlapping balancing domains to achieve global balancing [20]. Load information is exchanged between neighbor processors. Whenever a load L_i increases to $(1/u)L_i$ or drops to uL_i, update messages are sent to its neighbors to update the load information, where u is called the *load update factor*. The balancing process is initiated by any processor whose load drops below a threshold L_{LOW}. Then, the average load of the processor itself and its neighbors, $\bar{L}_i$, is calculated. If a processor's load is below the average load by more than another threshold, $L_{threshold}$, it proceeds to implement the third phase of the load balancing process. Each neighbor k is assigned a weight h_k:

$$h_k = \begin{cases} L_k - \bar{L}_i, & if \;\; L_k > \bar{L}_i \\ 0 & otherwise \end{cases}$$

$$H_i = \sum_{k=1}^{K} h_k.$$

The amount of load requested by processor i from neighbor k is computed as:

$$\delta_k = (\bar{L}_i - L_i)\frac{h_k}{H_i}.$$

Load requests are sent to appropriate neighbors. Upon receipt of a load request, a processor will fulfill the request only up to an amount equal to half of its current load. In this algorithm, three parameters, L_{LOW}, $L_{threshold}$, and u, need to be adjusted for optimal performance. Thus, finding optimal parameter settings in general may be difficult.

There is another category of scheduling which carries out computation and scheduling alternately. It is sometimes referred to as *prescheduling* which is more closely related to RIPS. Prescheduling utilizes partial load information for load balancing. Fox *et al.* first adapted prescheduling to application problems with geometric structures [25, 1]. Some other works also deal with this type of problems [2, 26, 27]. The Parti project automates prescheduling for nonuniform problems [28]. The dimension exchange method (DEM) is a parallel scheduling algorithm applied to application problems without geometric structure [3]. It balances load for independent tasks with an equal grain size. The method has been extended by Willebeek-LeMair and Reeves [20] so that the algorithm can run incrementally to correct the unbalanced load due to the varied task grain sizes. However, the DEM scheduling algorithm generates redundant communications. It is designed specifically for the hypercube topology and is implemented much less efficiently on a simpler topology, such as a tree or a mesh [20]. RIPS uses optimal parallel scheduling algorithms, which is also applied to problems without geometric structures. RIPS minimizes the number of communications, as well as the data movement. Furthermore, RIPS is a general method and applies to different topologies, such as the tree, mesh, and k-ary hypercube. In this paper, we present a basic algorithm for the tree topology.

Optimal mapping of communications has been studied extensively. Static communication mapping minimizes the network conflicts at compile-time [7, 29, 30, 31]. Yu and Das have de-

veloped a task allocation algorithm for MIN-based multiprocessors to minimize conflicts [32]. In general, it is difficult, almost impossible, to obtain an optimal communication mapping for a problem with dynamic communication structures. In RIPS, communication in the system phase is conflict-free.

RIPS can be presented in its two major components: incremental scheduling and parallel scheduling. The incremental scheduling policy decides when to transfer a user phase to a system phase and which tasks are selected for scheduling. The parallel scheduling algorithm is applied in the system phase to collect system load information and to balance the load.

3. Incremental Scheduling

Typically, a runtime scheduling algorithm has four components: a transfer policy, a selection policy, a location policy, and an information policy [11]. The transfer policy determines whether a processor is in a suitable state to participate in a task scheduling. The selection policy determines which tasks should be scheduled. The location policy determines to which processor a task selected for scheduling should be sent. The information policy is responsible for triggering the collection of system load information. RIPS can also be described with these four policies. Here, the transfer policy determines when the next system phase should start. The selection policy determines a set of tasks that are to be scheduled. The location policy determines to which processor each task will be scheduled. The information policy determines when to collect the system load information. The selection policy and the transfer policy are the central components of incremental scheduling. The location and information policies are two major components of parallel scheduling, which will be discussed in the next section.

The transfer policy in RIPS includes two sub-policies: a local policy and a global policy. Each individual processor determines if it is ready to transfer to the next system phase based on its local condition. Then all processors cooperate together to determine the transfer from the user phase to the system phase based on the global condition.

We consider two local policies: *eager scheduling* and *lazy scheduling*. In the eager scheduling, every task must be scheduled before it can be executed. In the lazy scheduling, scheduling is postponed as much as possible. In this way, some tasks could be executed directly without being scheduled.

The eager scheduling is implemented with two queues in each processor. One is called *ready-to-execute (RTE)* queue, the other *ready-to-schedule (RTS)* queue. At the beginning of a user phase, all the RTS queues in the system are empty and the RTE queue of every processor holds almost the same number of tasks ready to be executed. During the user phase, new tasks can be

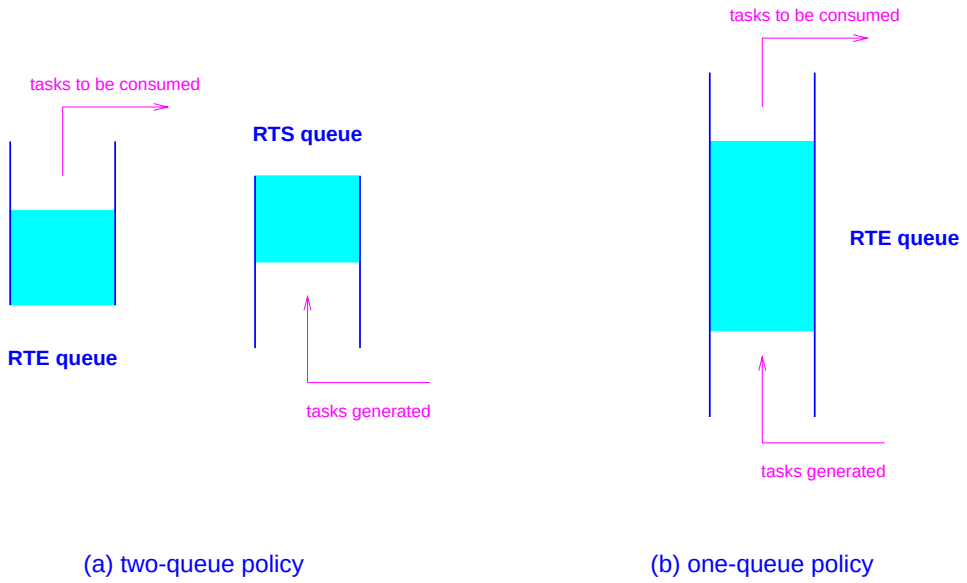


Figure 2: RTE and RTS queues

generated and entered into the local RTS queue, while the tasks in the RTE queue are consumed, as shown in Figure 2(a). When the RTE queue becomes empty, the processor is ready to transfer from this user phase to the next system phase. At the transfer, some of the RTE queues may be empty and others may have tasks left, because the consumption rate might not be the same due to unequal task grain sizes. In the beginning of the system phase, all the tasks left in the RTE queues, if any, will be moved back to the RTS queues and rescheduled together with the newly-generated tasks. The system phase schedules tasks in all RTS queues and distributes them evenly to the RTE queues. The tasks in RTS queues enter the local RTE queue or a remote RTE queue, depending on the scheduling results.

The lazy scheduling uses only a single queue, RTE, to hold all tasks, as shown in Figure 2(b). The tasks scheduled to the processor and the tasks generated at the processor are not distinguished. The newly-generated tasks enter the RTE queue directly. Some tasks may be generated and executed in the same processor without being scheduled. The transfer condition from a user phase to the next system phase is the same as the eager scheduling, that is, when the RTE queue becomes empty. In this way, only a fraction of tasks are scheduled and the number of total system phases could be reduced.

The two local policies correspond to the two queuing polices. They are illustrated as follows:

	Queuing policy	New tasks
Eager	two queues (RTE, RTS)	enter to RTS
Lazy	one queue (RTE)	enter to RTE

Two queue policies involve overhead for moving tasks between the queues. To avoid this overhead, we can implement the two queues as a single queue with a pointer that divides the single queue into two parts, as shown in Figure 3. The upper part is the RTE queue and the lower part is the RTS queue. During the user phase, the position of the pointer is fixed and only the tasks in the RTE queue can be executed. When transferred from a user phase to the next system phase, some tasks may be left in the RTE queue. Prior to system scheduling, the pointer moves up to cover all tasks in the RTS queue. In this way, the task copying from the RTE queue to the RTS queue can be eliminated. After scheduling, the pointer moves down to let the entire queue become the RTE queue. This implementation can easily adapt to the one queue policy, as long as the pointer points to the bottom of the queue during the user phase. This mechanism is useful for an adaptive algorithm that changes between a one-queue policy and a two-queue policy.

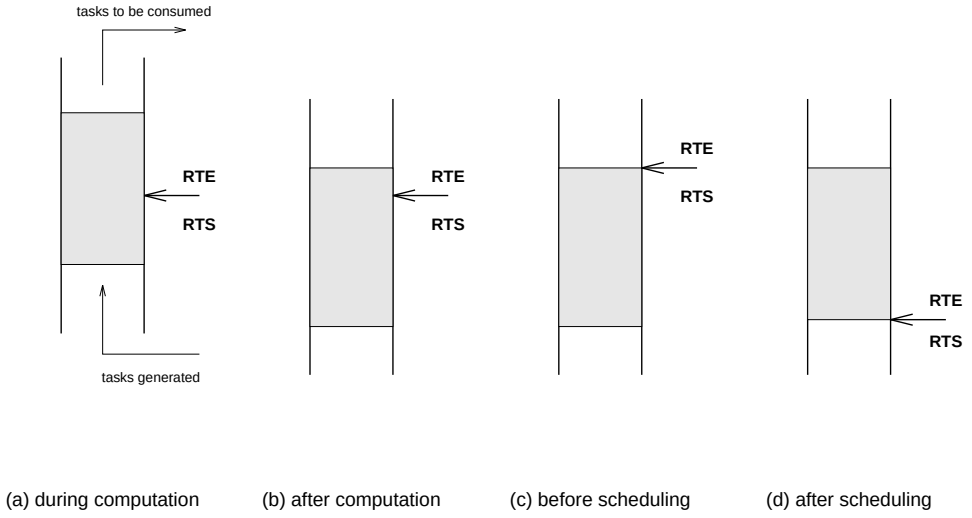


Figure 3: Implementation of RTE and RTS queues

Two possible global policies, called *ALL* and *ANY*, are listed as follows:

	Description
ALL	all RTE queues are empty
ANY	any RTE queue is empty

The ALL policy states that the transfer from a user phase to the next system phase will be initiated only when all the processors satisfy their local conditions. Whereas, with the ANY policy, as long as one processor has met its local condition, the transfer is initiated.

To test whether a transfer condition is satisfied, a naive implementation periodically invokes a global reduction operation. If the condition is satisfied, the system switches from this user phase to the next system phase; otherwise, it continues on the user phase. The time interval between

two consecutive global reduction operations should be carefully determined. An interval that is too short increases communication overhead; an interval that is too long may result in unnecessary processor idle. The optimal length of the interval is to be determined by empirical study.

Although the periodical reduction is a simple and general implementation, it may interfere with the underlying computation many times before the condition is satisfied. This overhead could be eliminated for some particular policy. The following method can be used for the ALL policy: a processor sends a *ready* signal to its parent when the local condition is satisfied and when a *ready* signal is received from each of its children. When the root processor satisfies the local condition and receives a *ready* signal from each of its children, the global ALL condition has been reached. The root processor will broadcast an *init* signal to all other processors to start the system phase. Some processors can be idle for a while before the global ALL condition is reached, as shown in Figure 4(a).

For the ANY policy, an alternative implementation allows any processor that satisfies the local condition to become an initiator and broadcast an *init* signal to all other processors. A processor, upon receiving the *init* signal, switches from this user phase to the next system phase. Because of communication delay, more than one processor could claim to be an initiator. Therefore, a processor may receive more than one *init* signal. A phase index variable is used to eliminate redundant *init* signals. As each *init* signal is tagged with a phase index variable, all the *init* signals with the same phase index variable, except the one received first, are considered to be redundant. Some machines provide a fast *or-barrier* synchronization, such as the *eureka mode* in Cray T3D [33]. Implementing the ANY policy can utilize this synchronization.

Note that when an idle processor initiates a phase transfer, other processors may still be executing tasks. The idle processor must wait until every processor finishes the current task execution, as shown in Figure 4(b). If the task grain size is very large, a preemptive strategy is encouraged to reduce the idle time. Although a phase transfer can cause idle, the idle time is not significantly large because phase transfers are not frequently invoked in many applications. Our performance study has shown that the idle time caused by phase transferring in our algorithms is much less than the idle time caused by load imbalance in other scheduling algorithms. Finally, if a co-processor can handle load balancing and communication activities, the idle time can be reduced. In this way, the main processor can continuously execute tasks when the co-processor initiates a load balancing process.

Now we consider the phenomenon that parallelism in a system is limited. If the number of tasks scheduled in a system phase is smaller than the number of processors, the following user phase is defined as a *low-parallelism* phase. In this case, some processors may have no task being scheduled. In the ANY policy, a processor that has not been scheduled a task will initiate the next system phase immediately by broadcasting an *init* message. If the *init* signal arrives at a

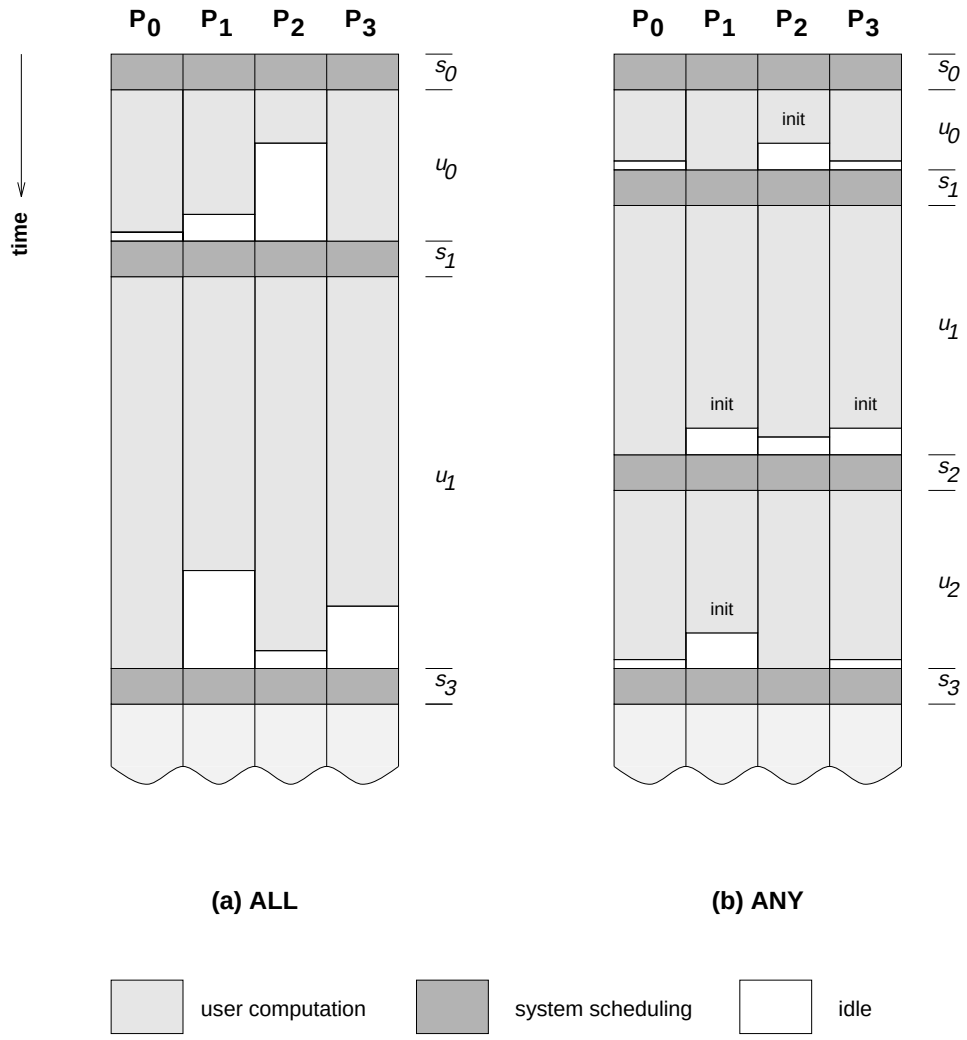


Figure 4: ALL and ANY policies

processor before the processor executes its first task, the processor will not be able to execute any task before the next system phase. In an extreme case, a user phase can transfer to the next system phase without any progress of task execution. To eliminate this problem, the eligibility of the initiator is defined for the ANY policy. That is, a processor is *eligible* as an initiator if and only if it has been scheduled at least one task in the previous system phase.

The second problem in a low-parallelism phase is the lack of awareness of low-parallelism caused by the lazy scheduling. With the ANY-Lazy policy, the processors that are eligible as initiators may keep generating and executing new tasks without knowing that other processors that are not eligible as initiators may starve at the same time. With the ALL-Lazy policy, the processors that have at least one task to execute may delay the global ALL reduction, leaving other processors starving. To solve this problem, a lazy scheduling is required to be adapted to the eager scheduling in a low-parallelism phase. In other words, the eager scheduling should always be applied to increase parallelism in a low-parallelism phase.

Transfer from the system phase to the user phase does not require a synchronization. Each processor terminates by itself and proceeds to the next user phase.

4. Parallel Scheduling

In this section we discuss *global parallel scheduling*, which is called *parallel scheduling* for short in the following context. Parallel scheduling is different from *dynamic scheduling*. With dynamic scheduling, processors exchange information and work load *concurrently*. While some processors are executing the user program, other processors may be executing the scheduling algorithm. Only partial load information is collected in consideration of scalability, and it is impossible to reach a fully-balanced load. Dynamic scheduling may be unstable and processor thrashing can occur.

Parallel scheduling executes the scheduling algorithm *in parallel*. All processors cooperate together to collect load information and to exchange work load in parallel. With parallel scheduling, it is possible to obtain high-quality scheduling and scalability simultaneously. Furthermore, parallel scheduling is stable because it is a synchronous approach.

Different logical network topologies need different scheduling algorithms. The algorithms for the mesh and hypercube topologies can be found in [34]. Here we present a parallel scheduling algorithm for the tree topology, in which the communication network is tree-structured and each node of the tree represents a processor. The algorithm, called *Tree Walking Algorithm (TWA)*, is shown in Figure 5. The objective of scheduling is to schedule works so that each processor has the same work load, which requires an estimation of task execution time. The estimation can be

application-specific, leading to a less general approach. Sometimes, such an estimation is difficult to obtain. Therefore, in TWA, each task is presumed to require the equal execution time, and the objective of the algorithm becomes to schedule tasks so that each processor has the same number of tasks. Inaccuracy caused by grain-size variation can be corrected in the next system phase. An algorithm with estimated time of tasks could improve load balancing to some extent. However, the algorithm is more complex so that the scheduling overhead increases, which may overwrite this benefit. We will compare the performance of TWA to a modified TWA with estimated time to illustrate this trade-off.

The other objective of this algorithm is to minimize communication overhead of load balancing. Ideally, a task with a small communication/computation ratio should have a higher priority to migrate. Although the communication/computation ratio is difficult to predict, it has been observed that the communication/computation ratio does not change substantially in a single application. Thus, we can use the number of tasks migrated instead of the actual communication cost as the objective function.

In this algorithm, the first step will be executed only at the system setup time. Steps 2 and 3 collect the system load information. In step 2, the total number of tasks is counted with a global reduction operation. At the same time, each node records the number of tasks in its subtree and its children's subtrees, if any. In step 3, the root calculates the average number of tasks per processor and then broadcasts the number to every processor so that each processor knows if its subtree is over-loaded or under-loaded for step 4. If the number of tasks cannot be evenly divided by the number of processors, the remaining R tasks are evenly distributed to the first R processors so that they have one more task than others. In step 5, the work load is exchanged so that at the end of the system phase, each processor has the almost same number of tasks. This algorithm is deadlock-free because there is no loop in the tree topology.

Example 1:

An example is shown in Figure 6 and Table I. The nodes in the tree are numbered by preorder traversal. At the beginning of scheduling, each node has w_i tasks ready to be scheduled. Values of W_i are calculated at step 2. The root calculates the value of w_{avg} and R:

$$w_{avg} = 4,\ R = 5.$$

Then each node calculates the value of Q_i at step 4. The numbers of tasks to be exchanged between nodes are shown in Figure 6. The load exchange takes four communication steps to finish:

Tree Walking Algorithm (TWA)

1. Assign each node an *order* i according to the preorder traversal; and N_i, the number of nodes of its subtree, where $i = 0, 1, ..., N-1$, and $N = N_0$ is the number of nodes in the system.

2. Let w_i be the number of tasks in node i. Perform a *sum* reduction of w_i and let W_i denote the sum of w_i in the subtree of node i. Each node also keeps records of W_i of its children (if any).

3. The root calculates $w_{avg} = \lfloor W_0/N \rfloor$ and $R = W_0 \ mod \ N$, and broadcasts w_{avg} and R to all other nodes.

4. Each node calculates its quota q_i that indicates how many tasks are to be scheduled to the node:

$$q_i = \begin{cases} w_{avg} + 1 & if \ i < R \\ w_{avg} & if \ i \geq R \end{cases}$$

   Each subtree, rooted at node i, calculates its quota

$$Q_i = \sum_{all \ nodes \ in \ subtree \ i} q_j$$

   that indicates how many tasks are to be scheduled to the subtree. Q_i can be calculated directly as follows:

$$Q_i = w_{avg} * N_i + r_i$$

   where

$$r_i = \begin{cases} 0 & if \ i \geq R \\ N_i & if \ i \leq R - N_i \\ R - i & if \ R - N_i < i < R \end{cases}$$

   Each node keeps records of Q_i and Q_j, where node j is node i's child (if any).

5. Each node i waits for the incoming message from its parent if $W_i < Q_i$, and the incoming message from each of its children j if $W_j > Q_j$.

   Each node i, after it has received all incoming messages, sends $(W_i - Q_i)$ tasks to its parent if $W_i > Q_i$; and for each child j, if $W_j < Q_j$, sends $(Q_j - W_j)$ tasks to child j.

Figure 5: Parallel tree walking algorithm

1) node 3 to node 1, node 8 to node 6, node 4 to node 5
2) node 1 to node 0
3) node 0 to node 6
4) node 6 to node 7

At the end of scheduling, nodes 0–4 have 5 tasks each and nodes 5–8 have 4 tasks each.

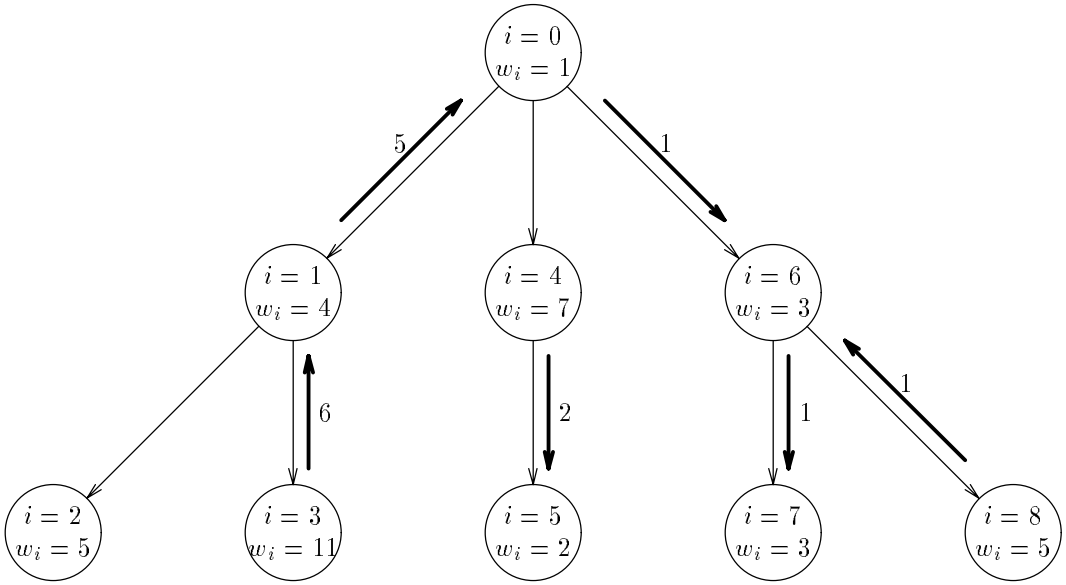


Figure 6: Example for the tree walking algorithm.

Table I: Example for the Tree Walking Algorithm

i	w_i	N_i	W_i	Q_i
0	1	9	41	41
1	4	3	20	15
2	5	1	5	5
3	11	1	11	5
4	7	2	9	9
5	2	1	2	4
6	3	3	11	12
7	3	1	3	4
8	5	1	5	4

In the following, we prove that the algorithm can

- achieve a well-balanced load;
- minimize the task migration and communication; and
- maximize locality.

Theorem 1: The number of tasks in each processor differs by at most one after executing the Tree Walking Algorithm.

Proof: Executing the algorithm, the number of tasks in processor i is

$$w_i^{'} = w_i + (Q_i - W_i) + \sum_{j \; is \; child \; of \; i} (W_j - Q_j).$$

Because

$$\sum_{j \; is \; child \; of \; i} W_j = W_i - w_i,$$

$$w_i^{'} = w_i + (Q_i - W_i) + (W_i - w_i) - \sum_{j \; is \; child \; of \; i} Q_j = Q_i - \sum_{j \; is \; child \; of \; i} Q_j = q_i.$$

Since q_i is either w_{avg} or $w_{avg} + 1$, the number of tasks in each processor differs by at most one. □

When the total number of tasks can be evenly divided by N, the number of processors, each processor has w_{avg} tasks. Otherwise, each of the first R processors have $(w_{avg} + 1)$ tasks and the rest of processors have w_{avg} tasks. In the following, we assume that the number of tasks T is evenly divided by N. When T cannot be evenly divided by N, the algorithm is nearly-optimal.

Theorem 2: The Tree Walking Algorithm minimizes the total number of communications and the total number of task-hops $\sum_j e_j$, where e_j is the number of tasks transmitted through edge j.

Proof: In TWA, there is one communication between subtree i and its parent if $Q_i \neq W_i$. When $Q_i \neq W_i$, at least one communication is required. Therefore, the total number of communications is minimized.

In each subtree, if $Q_i \geq W_i$, then it needs to receive from its parent $(Q_i - W_i)$ tasks, which is the minimum number of tasks to be transmitted to the subtree. Similarly, if $Q_i < W_i$, then it needs to send to its parent $(W_i - Q_i)$ tasks, which is the minimum number of tasks to be transmitted from the subtree. Therefore, the total number of task-hops $\sum_j e_j$ is minimized. □

This algorithm also maximizes locality. *Local tasks* are the tasks that are not migrated to other processors, and *non-local tasks* are those that are migrated to other processors. Maximum locality implies the maximum number of local tasks and the minimum number of non-local tasks. The following lemma gives the minimum number of non-local tasks.

Lemma 1: To balance the load, the minimum number of non-local tasks is

$$\sum_i \max(w_{avg} - w_i, 0).$$

Proof: Each processor with $w_i < w_{avg}$ must receive $(w_{avg} - w_i)$ tasks from other processors to balance the load. Therefore, a total of $\sum_i \max(w_{avg} - w_i, 0)$ tasks must be transmitted between processors. □

Another measure of locality is $\sum_k d_k$, where d_k is the distance that non-local task k has traveled. The following theorem shows that the TWA algorithm maximizes locality in both measures.

Theorem 3: The number of non-local tasks in the TWA algorithm is

$$\sum_i \max(w_{avg} - w_i, 0),$$

and the total number of distances $\sum_k d_k$ traveled by non-local tasks is minimized.

Proof: In TWA, each processor receives tasks before sending tasks. At any time when executing the TWA algorithm, the number of tasks in each processor is not less than $\min(w_i, w_{avg})$. Thus, in all processors, at least $\sum_i \min(w_i, w_{avg})$ tasks are local. Therefore, the number of non-local tasks is no more than

$$N \times w_{avg} - \sum_i \min(w_i, w_{avg}) = \sum_i (w_{avg} - \min(w_i, w_{avg})) = \sum_i \max(w_{avg} - w_i, 0).$$

As stated in Lemma 1, TWA minimizes the number of non-local tasks.

The total distance traveled by all tasks is equal to the total number of task-hops:

$$\sum_k d_k = \sum_j e_j,$$

which has been proved to be minimum by Theorem 2. □

In this algorithm, steps 2 and 3 spend $2m$ communication steps, where m is the depth of the tree. The communication steps in step 5 is that from a leaf node to another leaf node, which is at most $2m$. Therefore, the total communication steps of this algorithm is at most $4m$. With a balanced tree, $m = logN$, and the number of communication steps of this algorithm is $O(logN)$.

In step 5 of the Tree Walking Algorithm, each node must receive all incoming messages before sending out messages. This algorithm can be further optimized by relaxing this constraint, shown in Figure 7, where only the fifth step in the TWA has been modified and rewritten. We called this algorithm the Modified Tree Walking Algorithm (MTWA). In this algorithm, a node is able to send some messages out before it has received all incoming messages. The communication time and processor idle time can be reduced. It takes only two communication steps for Example 1:

1) node 3 to node 1, node 0 to node 6, node 6 to node 7,
   node 8 to node 6, node 4 to node 5
2) node 1 to node 0

The Modified Tree Walking Algorithm may have some negative impact in locality. In TWA, a node can keep the maximum number of local tasks and send non-local tasks to other nodes. But in MTWA, a node may send local tasks to other nodes and then receive tasks from others.

Modified Tree Walking Algorithm (MTWA)

The first four steps are the same as TWA algorithm.

5. For each node i, construct two sets: one is *InMsg* set including all nodes that are expected to send tasks to node i, and the other is *OutMsg* set consisting of all destination nodes to which node i needs to send tasks. The two sets are constructed as follows:

   If $W_i < Q_i$, add its parent node to the *InMsg* set; if $W_i > Q_i$, add tuple (*its parent node*, $W_i - Q_i$) into the *OutMsg* set.

   For each of its child node j, if $W_j > Q_j$, add its parent node to the *InMsg* set; if $W_j < Q_j$, add tuple (*its parent node*, $Q_j - W_j$) into the *OutMsg* set.

   Sort the tuples in the *OutMsg* set according to the number of tasks in an increasing order to form an *OutMsg list*.

   While neither the *InMsg* set nor the *OutMsg* list is empty
   while there is an incoming message arrived
   receive the message and update the *InMsg* set
   if there are adequate tasks to meet the first request in the *OutMsg* list
   send the message to the destination node and update the *OutMsg* list

Figure 7: Modified tree walking algorithm

Therefore, the decision on use of TWA or MTWA is a trade-off between scheduling time and locality.

5. Experimental Study

The RIPS system has been implemented on a 32-node TMC CM-5 machine. The network that connects the processors in CM-5 is a 4-ary *fat tree*. Different from the general tree topology used in the Tree Walking Algorithm, CM-5 has all processors as leaf nodes. The algorithm shown in Figure 8 is used in this implementation to map the scheduling tree to the fat tree topology. This algorithm is designed to minimize the communication distance and to distribute communication traffic. The tree used in a 32-node system is shown in Figure 9.

The system has been tested with three application problems. The first one, the exhaustive search of the N-Queens problem, has an irregular and dynamic structure. The number of tasks generated and the computation amount in each task are unpredictable. The second one, iterative deepening A* (IDA*) search is a good example of parallel search techniques [35]. The sample problem is the 15-puzzle with three different configurations. The grain size may vary substantially, since it dynamically depends on the currently estimated cost. In addition, synchronization at

Every node n in the fat tree has the following information in $node[n]$:

parent: its parent ID
numChild:the number of its children
child[4]: its children's ID
level: its level in the tree, the root has the highest level
a: being its parent's the ath child

There is a global table used during the tree construction:

$nonLeaf[maxLevel][maxNodeinLevel]$: the node ID in the fat tree

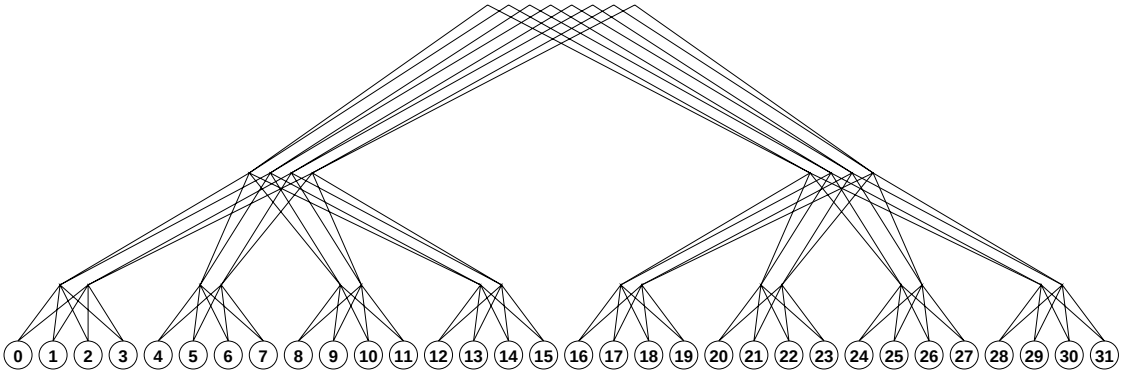
Scheduling tree construction algorithm for a 4-ary fat tree:

N is the number of nodes in the tree
L is the number of levels in the tree
K[i] is the number of nodes in the kth level

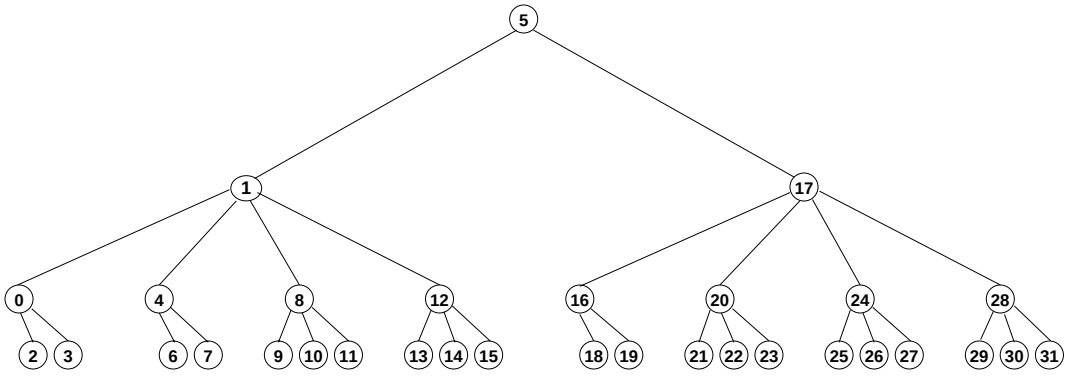
$$K[i] = \begin{cases} 1 + (N-1)/4^i & L \geq i \geq 1 \\ N - L - \sum_{j=1}^{L} (N-1)/4^i & i = 0 \end{cases}$$

for i = L downto 1
  for j = 0 to K[i] - 1
    the corresponding node ID is
      $n = j * 4^i + \sum_{k=0}^{i-2} 4^k$
    setup relevant information about the tree at the ith level:
      $nonLeaf[i][j] = n$
    setup information at node n:
      $node[n].level = i, \quad node[n].a = j \bmod 4$
    build up the parent and children relation:
      $p = node[n].parent = \begin{cases} nonLeaf[i+1][j/4] & i < L \\ -1 & i = L \end{cases}$
      $node[p].child[node[p].numChild] = n$
      $node[p].numChild = node[p].numChild + 1$
for level 0
  for $i = 0$ *to* $N - 1$, if $node[i].levelisunassigned$
    $p = node[i].parent = nonLeaf[1][i/4]$
    $node[i].a = node[p].numChild$
    $node[p].child[node[p].numChild] = i$
    $node[p].numChild = node[p].numChild + 1$

Figure 8: Mapping algorithm that maps 4-ary fat tree to scheduling tree



(a) a 4-ary fat tree



(b) scheduling tree

Figure 9: Mapping 4-ary fat tree to scheduling tree

each iteration reduces the effective parallelism. Performance of this problem is therefore not as good as others. Search problems provide a high degree of parallelism. Then the major issue of solving parallel search problems is to balance the load. Rao and Kumar have solved the 15-puzzle problem with the IDA* search using a simple load balancing algorithm [36]. That is, whenever a processor needs work, it sends a request to one of its neighbors. If the neighbor has work available above the cutoff depth, it sends the extra work to the requesting processor; otherwise it sends a reject message. This receiver-initiated, within-neighborhood algorithm can provide a fairly good performance. However, it is not able to balance the load as well as RIPS can do.

The third one, a molecular dynamics program named GROMOS, is a real application problem [37, 38]. The test data for GROMOS is the bovine superoxide dismutase molecule (*SOD*), which has 6968 atoms [39]. The kernel of the GROMOS code is a calculation of the forces between pairs of atoms. Since the forces decrease as the distances between the pairs of atoms increase, they are approximated by considering only pairs of atoms which are closer together than a predefined cufoff radius. In our test, the cutoff radius is predefined to 8 $\mathring{A}$, 12 $\mathring{A}$, and 16 $\mathring{A}$. GROMOS has a more predictable structure. The kernel of the GROMOS code contains a parallelizable outer loop, but has an inner loop for which the number of iterations varies between different iterations of the outer loop. Hanxleden and Kennedy ported GROMOS to CM-2 and DECmmp 12000 with a transformation called *loop flattening* to overlapping the executions of different iterations [38]. In our implementation of GROMOS, only the outer loop is partitioned and each task has several iterations of the outer loop. The number of tasks is known when the input data is given, but the computation density in each task varies. Thus, a load balancing mechanism is necessary.

We first compare four combinations of the transfer policies: ALL-Eager, ALL-Lazy, ANY-Eager, and ANY-Lazy. The application problems used in this comparison are the exhaustive search with 14-Queens and GROMOS with the cutoff radius of 8 $\mathring{A}$. Table II shows (1) the number of system phases; (2) the total number of tasks and the number of tasks that are scheduled; (3) the number of communications; (4) the average overhead time; (5) the average processor idle time; and (6) the total execution time. The execution time is the sum of the overhead time, idle time, and busy time, which is illustrated in Figures 10.

With the ALL-Eager policy, each system phase schedules tasks in one level of the tree and the next user phase consumes all of the scheduled tasks. Therefore, the number of phases is equal to the depth of the tree. Also, the number of scheduled tasks is equal to the total number of tasks, as each task is scheduled exactly once. With the ALL-Lazy policy, the number of phases could be less than or equal to the depth of the tree, since some tasks are executed locally without being scheduled.

The number of phases in the ANY policy can be larger than the depth of the tree. The number of tasks scheduled may exceed the number of tasks, which implies that some tasks may

Table II: Policy Comparison on 32-node CM-5

			# of phases	# of tasks	# of tasks scheduled	# of comm.	overhead (seconds)	idle time (seconds)	exec. time (seconds)
14-Queens	ALL	Eager	5	11166	11166	301	0.29	1.78	8.34
		Lazy	4	11166	170	248	0.14	1.11	7.52
	ANY	Eager	10	11166	15235	3948	0.67	0.09	7.03
		Lazy	8	11166	3997	1776	0.52	0.08	6.87
GROMOS (8 Å)	ALL	Eager	3	4986	4986	186	0.89	1.98	6.15
		Lazy	2	4986	1022	124	0.85	2.01	6.16
	ANY	Eager	10	4986	10688	3596	1.10	0.12	4.50
		Lazy	7	4986	6012	2322	0.60	0.11	3.99

be scheduled more than once. With the ANY-Lazy policy, the number of tasks scheduled could also be smaller than the total number of tasks, since some tasks may not be scheduled.

The number of communications listed in Table II includes only the communications to transfer from user phases to system phases, but not the communications in the system phases. In general, the number of communications with the ALL policy is less than that with the ANY policy. Most communications with the ANY policy come from many simultaneous *init* messages. The number of communications with the lazy scheduling is less than that with the eager scheduling, because the number of phases with the lazy scheduling is smaller than that with the eager scheduling.

The overhead time is the time spent on the bookkeeping, information collection, and load balancing. The idle time is the time the processor has no work to do. The overhead time and the idle time listed in Table II are the average time at one processor when averaged across all processors. The ALL policy involves fewer phases and communications so that the overhead is smaller. However, because all processors must wait until every ready task has been executed, processor idle time is large. On the other hand, the ANY policy, although it involves larger overhead, balances the load well and reduces the idle time. The total execution time of the ANY policy is consistently less than that of the ALL policy. The lazy scheduling is better than the eager scheduling since it involves less overhead and idle time.

In summary, the ALL policy is easy to implement and involves less communication. However, it has some potential drawbacks. It does not allow task rescheduling and therefore is not able to correct the load imbalance due to the grain size variation. A processor that has finished execution must wait for other processors to finish, resulting in processor idle. To utilize the resource of every processor, a system phase can be triggered earlier, determined by the ANY policy. The ANY policy results in more system phases and larger overhead, but reduces processor

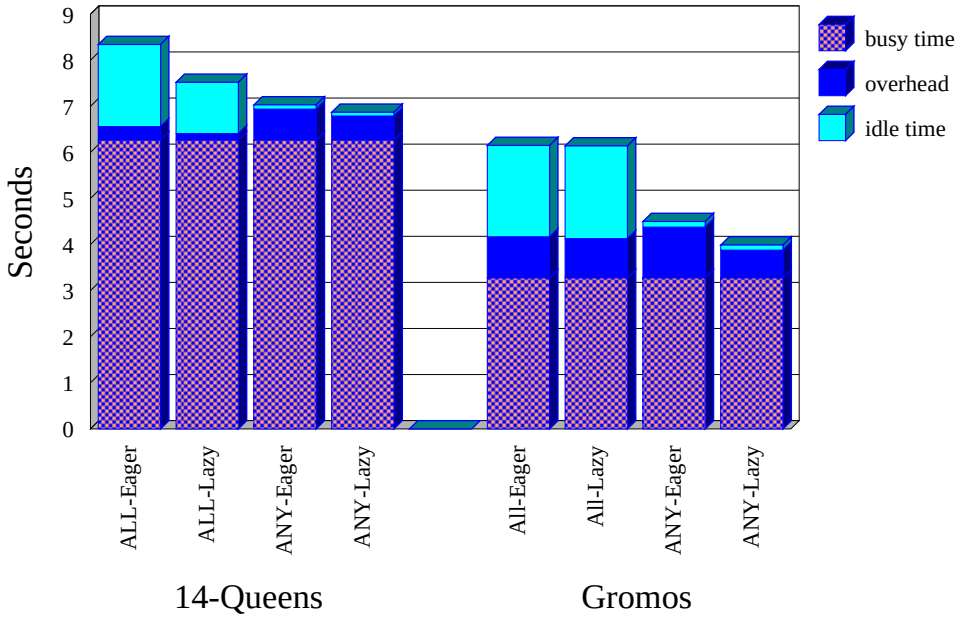


Figure 10: Policy comparison.

idle time. In the eager scheduling, each task must be scheduled before execution. In contrast, the lazy scheduling allows an unscheduled task to execute. The lazy scheduling outperforms the eager scheduling, since it reduces the number of tasks to be scheduled, as well as the number of scheduling phases.

The Modified Tree Walking Algorithm (MTWA) does not improve performance significantly compared to the Tree Walking Algorithm (TWA). While MTWA reduces the communication steps, its locality is not as good as TWA. In general, non-local tasks in MTWA are about 40 percent more than that in TWA. We did not conduct an extensive comparison since both algorithms perform about the same. Therefore, only the performance with the Tree Walking Algorithm is shown below.

For comparison purposes, we have also implemented a TWA algorithm with estimated time (TWAET). In this algorithm, each task is assigned a weight which is its estimated execution time. The estimation is obtained by taking the average of the execution time for the same type of tasks. The weights of all tasks in a processor are added up and then the total weight of tasks in all processors is obtained by a global reduction of the sum. The root calculates the average weight per processor and broadcasts it to all processors. Thus, each processor knows how much work load needs to be sent to other processors and executes a packing algorithm to select some tasks so that the total weight of these tasks is roughly the weight required to be sent. The tasks need to be packed before they are sent out. This algorithm is more expensive because of its message packing, buffer copying, and floating-point operations. The comparison is shown in Table III.

TWAET can balance the load better and the idle time is less than that of TWA. On the other hand, TWAET is more complex than TWA, resulting in a larger overhead. TWAET is faster for some instances and TWA is faster for other. Performance of both algorithms is roughly the same. Therefore, in the following comparison, we use only the TWA algorithm.

Table III: Comparison of TWA and TWAET Algorithms

		overhead (seconds)	idle time (seconds)	exec. time (seconds)	efficiency
Exhaustive search 13-Queens	TWA	0.29	0.05	1.36	75%
	TWAET	0.40	0.04	1.46	72%
Exhaustive search 14-Queens	TWA	0.52	0.08	6.87	91%
	TWAET	0.59	0.06	6.90	90%
Exhaustive search 15-Queens	TWA	0.97	0.23	42.0	97%
	TWAET	0.97	0.16	41.8	97%
IDA* search config. #1	TWA	0.53	0.13	2.04	66%
	TWAET	0.80	0.09	2.24	60%
IDA* search config. #2	TWA	1.17	0.41	7.98	80%
	TWAET	1.47	0.28	8.13	78%
IDA* search config. #3	TWA	2.04	0.52	30.2	91%
	TWAET	3.01	0.37	30.9	89%
GROMOS (8 $\AA$)	TWA	0.60	0.11	3.99	82%
	TWAET	0.62	0.10	4.00	82%
GROMOS (12 $\AA$)	TWA	1.16	0.36	11.4	87%
	TWAET	1.12	0.24	11.2	88%
GROMOS (16 $\AA$)	TWA	0.95	0.65	22.4	93%
	TWAET	0.97	0.61	22.4	93%

In Table IV, we compare RIPS to three dynamic load balancing strategies: random allocation, gradient model and RID. The performance of RIPS shown in Table IV is with the ANY-Lazy policy. The comparison is done with (1) the number of tasks that are sent to other processors, which is a measure of locality; (2) the overhead time, which includes all system overhead; (3) the idle time, which is a measure of load imbalance; (4) the execution time; and (5) the efficiency. Here, the efficiency is defined as $\mu = \frac{T_s}{T_p * N}$, where N is the number of processors, T_s is the sequential execution time, and T_p is the parallel execution time. Although the randomized allocation does not have good locality, it can balance the load fairly well. The gradient model does not show good performance for the N-Queens problem. However, it performs fairly on the less irregular, highly parallel GROMOS program. Generally speaking, it cannot balance the load well, since the load is spread slowly. In addition, the system overhead is large since information and tasks are frequently exchanged. The RID parameters should be selected as a function of the number of tasks

Table IV: Comparison of Four Scheduling Algorithms on 32-node CM-5

		# of tasks	# of non-local tasks	overhead (seconds)	idle time (seconds)	exec. time (seconds)	efficiency
Exhaustive search 13-Queens	Random	7579	7342	0.40	0.08	1.50	68%
	Gradient Model	7579	4255	0.97	0.88	2.87	36%
	RID	7579	2597	0.41	0.10	1.53	67%
	RIPS	7579	314	0.29	0.05	1.36	75%
Exhaustive search 14-Queens	Random	11166	10832	0.62	0.25	7.14	88%
	Gradient Model	11166	6305	1.28	5.45	13.0	48%
	RID	11166	4218	0.57	0.27	7.11	88%
	RIPS	11166	645	0.52	0.08	6.87	91%
Exhaustive search 15-Queens	Random	15941	15459	1.10	1.50	43.4	94%
	Gradient Model	15941	9058	2.10	30.5	73.4	56%
	RID	15941	7103	0.78	1.42	43.0	95%
	RIPS	15941	925	0.97	0.23	42.0	97%
IDA* search config. #1	Random	2895	2804	0.34	0.63	2.31	58%
	Gradient Model	2895	2010	0.85	1.70	3.89	34%
	RID	2895	619	0.95	2.57	4.86	28%
	RIPS	2895	203	0.53	0.13	2.04	66%
IDA* search config. #2	Random	3382	3277	0.65	1.97	9.02	71%
	Gradient Model	3382	2184	2.34	7.25	16.0	40%
	RID	3382	382	3.20	14.8	23.4	27%
	RIPS	3382	257	1.17	0.41	7.98	80%
IDA* search config. #3	Random	29046	28138	1.33	4.65	33.5	82%
	Gradient Model	29046	19285	2.28	22.6	52.3	52%
	RID	29046	3038	2.35	6.83	36.7	75%
	RIPS	29046	1138	2.04	0.52	30.2	91%
GROMOS (8 $\AA$)	Random	4986	4831	0.26	0.56	4.10	80%
	Gradient Model	4986	2311	0.52	0.73	4.53	72%
	RID	4986	528	0.43	0.36	4.07	81%
	RIPS	4986	494	0.60	0.11	3.99	82%
GROMOS (12 $\AA$)	Random	4986	4833	0.25	1.76	11.9	83%
	Gradient Model	4986	2184	0.51	1.40	11.8	84%
	RID	4986	547	1.01	0.90	11.8	84%
	RIPS	4986	556	1.16	0.36	11.4	87%
GROMOS (16 $\AA$)	Random	4986	4832	0.26	3.94	25.0	83%
	Gradient Model	4986	2363	0.54	5.26	26.6	78%
	RID	4986	485	1.79	2.01	24.6	84%
	RIPS	4986	582	0.95	0.65	22.4	93%

per processor and the granularity of these tasks [40]. Three parameters, L_{LOW}, $L_{threshold}$, and u are adjusted to their optimal values: 2, 1, and 0.4, respectively. RID shows a better performance than the randomized allocation in most cases. However, it does not perform well for IDA* because of its low parallelism. It is known that a receiver-initiated approach does not do well in a lightly-loaded system [13]. When the problem size becomes large, such as the configuration #3, RID's performance is improved. In RIPS, the Tree Walking Algorithm can balance the load very well and the incremental scheduling is able to correct the load imbalance. Many people may expect large overhead from this accurate load-balancing algorithm. A surprising observation is that the overhead of RIPS is slightly larger than that of the randomized allocation and much smaller than that of other dynamic scheduling algorithms, such as the gradient model. It is partly due to the fact that many tasks are packed together for transmission, which reduces communication overhead, whereas, in dynamic scheduling, tasks are distributed individually.

Next, we use the randomized allocation as a baseline algorithm and show the relative performance of other scheduling algorithms. First, an *optimal efficiency* is calculated assuming (1) optimal scheduling; and (2) no overhead. The optimal efficiency is the best possible efficiency that can be obtained for a given problem on an ideal system. The optimal efficiencies for different problem sizes are shown in Table V. A measure used to determine the effectiveness of scheduling algorithm g is the *normalized quality factor*:

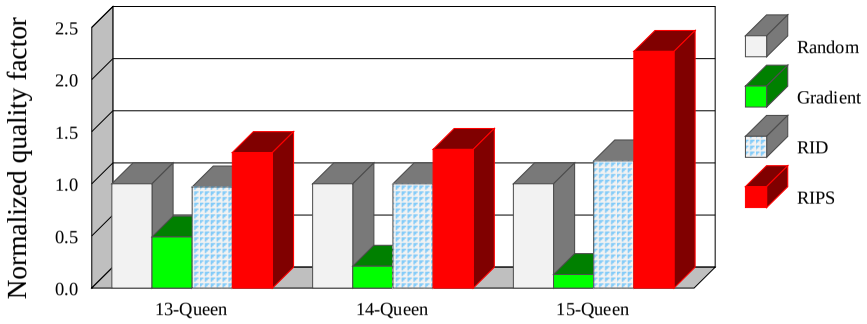
$$\mathcal{F} = \frac{\mu_{opt} - \mu_{rand}}{\mu_{opt} - \mu_g},$$

where μ_{opt} is the optimal efficiency, μ_{rand} is the efficiency of the randomized allocation algorithm, and μ_g is the efficiency of algorithm g. The factor of the randomized allocation algorithm is equal to 1. If the algorithm performs better than the randomized allocation, its value is larger than 1. Otherwise, it is smaller than 1. The normalized quality factors of these test problems are shown in Figure 11. For small problem sizes, the system overhead dominates the factor, whereas for large problem sizes, the factor is dominated by scheduling quality because the system overhead is relatively small. Therefore, the difference between scheduling algorithms can be easily recognized when the problem size is large.

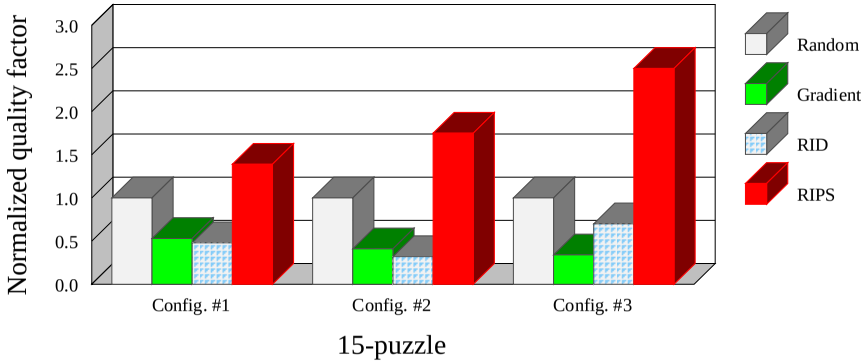
Table V: Optimal Efficiencies for Test Problems

Exhaustive Search			IDA* Search			GROMOS		
13-Queens	14-Queens	15-Queens	config. 1	config. 2	config. 3	8 Å	12 Å	16 Å
98.8%	99.2%	99.4%	85.3%	91.7%	97.2%	98.9%	98.9%	98.9%

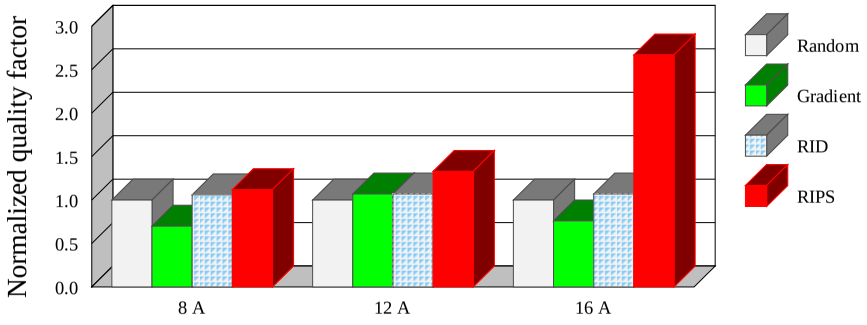
RIPS has been ported to a large CM-5 machine. Table VI shows a speedup comparison of four scheduling algorithms. Both the randomized allocation and RID scale up well and RIPS



(a) Exhaustive Search



(b) IDA* Search



(c) GROMOS

Figure 11: Normalized quality factor.

Table VI: Speedup Comparison on Large CM-5

		Number of Processors			
		64	128	256	512
Exhaustive search 15-Queens	Random	57.0	107	208	361
	Gradient Model	29.1	38.0	39.1	38.8
	RID	57.5	104	186	323
	RIPS	60.3	116	225	402
IDA* search config. #3	Random	47.1	71.8	103	139
	Gradient Model	30.4	41.3	52.7	59.6
	RID	28.0	27.3	24.0	20.2
	RIPS	54.3	84.8	123	176
GROMOS (16 Å)	Random	50.6	97.3	189	355
	Gradient Model	41.6	41.8	41.8	40.1
	RID	48.8	75.6	92.9	110
	RIPS	55.7	109	216	387

performs even better. The gradient model does not scale well because it spreads the load slowly. RID performs well for the N-Queens problem, but not for IDA* search because configuration #3 does not have enough parallelism on large systems. The value of u needs to be adjusted for low parallelism on large systems. For IDA*, it has been adjusted to 0.7, and for other two problems, it remains at 0.4.

6. Concluding Remarks

It has been widely believed that a scheduling method that collects load information from all processors in the system is neither practical, nor scalable. This research has demonstrated a scalable scheduling algorithm that uses the global load information to optimize load balancing. At the same time, this algorithm minimizes the number of tasks to be scheduled and the number of communications. Furthermore, in a dynamic system, when it intends to quickly and accurately balance the load, the system could become unstable. In RIPS, a synchronous approach eliminates the stability problem and is able to balance the load quickly and accurately.

RIPS combines the advantages of static scheduling and dynamic scheduling, adapting to dynamic problems and producing high-quality scheduling. It balances the load very well and effectively reduces the processor idle time. Tasks are packed to be sent to other processors which reduces significantly the number of messages. Its overhead is comparable to the low-overhead

randomized allocation. It applies to a wide range of applications, from slightly irregular ones to highly irregular ones.

Acknowledgments

We are very grateful to Reinhard Hanxleden for the GROMOS program, Terry Clark for the SOD data, and Marc Feeley for the elegant N-Queens program. This research was partially supported by NSF grants CCR-9109114 and CCR-9505300. The performance data was gathered on a 32-node CM-5 at NPAC, Syracuse University, and on a 512-node CM-5 at NCSA, University of Illinois.

References

[1] J. Salmon, "Parallel hierarchical N-body methods," tech. rep., CRPC-90-14, Center for Research in Parallel Computing, Caltech, 1990.

[2] K. M. Dragon and J. L. Gustafson, "A low-cost hypercube load balance algorithm," in *Proc. of the 4th Conf. on Hypercube Concurrent Computers and Applications*, pp. 583–590, 1989.

[3] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. of Parallel Distrib. Comput.*, vol. 7, pp. 279–301, 1989.

[4] I. Ahmad and Y. Kwok, "A parallel approach to multiprocessor scheduling," in *Int'l Parallel Processing Symposium*, pp. 289–293, Apr. 1995.

[5] M. Y. Wu, "Parallel incremental scheduling," *Parallel Processing Letters*, 1995.

[6] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 330–343, July 1990.

[7] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, June 1990.

[8] T. Yang and A. Gerasoulis, "PYRROS: Static task scheduling and code generation for message-passing multiprocessors," *The 6th ACM Int'l Conf. on Supercomputing*, July 1992.

[9] Y. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Supercomputer '92*, Nov. 1992.

[10] I. Ahmad, Y. Kwok, and M. Wu, "Performance comparison of algorithms for static scheduling of DAGs to multiprocessors," in *Second Australasian Conference on Parallel and Real-time Systems*, Sept. 1995.

[11] N. G. Shivaratri, P. Krieger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, vol. 25, pp. 33–44, Dec. 1992.

[12] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662–674, May 1986.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Eval.*, vol. 6, pp. 53–68, Mar. 1986.

[14] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Networks*, vol. 8, pp. 199–217, June 1984.

[15] T. L. Casavant and J. G. Kuhl, "Analysis of three dynamic distributed load-balancing strategies with varying global information requirements," in *Int'l Conf. on Distributed Computing System*, pp. 185–192, May 1987.

[16] Y.-T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.*, vol. C-34, pp. 204–217, Mar. 1985.

[17] Z. Lin, "A distributed fair polling scheme applied to parallel logic programming," *International Journal of Parallel Programming*, vol. 20, Aug. 1991.

[18] W. Shu, "Adaptive dynamic process scheduling on distributed memory parallel computers," *Scientific Programming*, vol. 3, pp. 341–352, 1994.

[19] M. Y. Wu, "Symmetrical hopping: A scalable scheduling algorithm on distributed memory machines," *Concurrency: Practice and Experience*, 1995.

[20] M. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. Parallel and Distributed System*, vol. 9, pp. 979–993, Sept. 1993.

[21] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE Trans. Software Engineering*, vol. 13, pp. 32–38, Jan. 1987.

[22] W. C. Athas, *Fine Grain Concurrent Computations*. PhD thesis, Dept. of Computer Science, California Institute of Technology, May 1987.

[23] R. M. Karp and Y. Zhang, "A randomized parallel branch-and-bound procedure," *Journal of ACM*, vol. 40, pp. 765–789, 1993.

[24] S. Chakrabarti, A. Ranade, and K. Yelick, "Randomized load balancing for tree structured computation," in *IEEE Scalable High Performance Computing Conference*, pp. 666–673, 1994.

[25] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, vol. I. Prentice-Hall, 1988.

[26] M. Berger and S. Bokhari, "A partitioning strategy for non-uniform problems on multiprocessors," *IEEE Trans. Computers*, vol. C-26, pp. 570–580, 1987.

[27] S. B. Baden, "Dynamic load balancing of a vortex calculation running on multiprocessors," Tech. Rep. Vol. 22584, Lawrence Berkeley Lab., 1986.

[28] J. Saltz, R. Mirchandaney, R. Smith, D. Nicol, and K. Crowley, "The PARTY parallel runtime system," in *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1987.

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

[30] H. Shen, "Self-adjusting mapping: a heuristic mapping algorithm for mapping parallel programs onto transputer networks," *The Computer Journal*, vol. 35, pp. 71–80, Feb. 1992.

[31] A. Kavianpour, "Systematic approach for mapping application tasks in hypercubes," *IEEE Trans. Computers*, vol. C–42, pp. 742–746, June 1993.

[32] C. Yu and C. R. Das, "Disjoint task allocation algorithm for MIN machines with minimal conflicts," *IEEE Trans. Parallel and Distributed System*, vol. 6, pp. 373–387, Apr. 1995.

[33] R. Koeninger, M. Furtney, and M. Walker, "A shared memory MPP from Cray research," *Digital Technical Journal*, vol. 6, no. 2, pp. 8–21, 1994.

[34] M. Y. Wu, "On runtime parallel scheduling," Tech. Rep. 95-34, Dept. of Computer Science, State University of New York at Buffalo, Apr. 1995.

[35] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, Sept. 1985.

[36] V. N. Rao and V. Kumar, "Parallel depth-first search, part I: Implementation," *Int. J. Parallel Programming*, vol. 16, no. 6, pp. 479–499, 1987.

[37] W. F. van Gunsteren and H. J. C. Berendsen, "GROMOS: GROningen MOlecular Simulation software," tech. rep., Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[38] R. v. Hanxleden and K. Kennedy, "Relaxing SIMD control flow constraints using loop transformations," Tech. Rep. CRPC-TR92207, Center for Research on Parallel Computation, Rice University, Apr. 1992.

[39] J. Shen and J. A. McCammon, "Molecular dynamics simulation of superoxide interacting with superoxide dismutase," *Chemical Physics*, vol. 158, pp. 191–198, 1991.

[40] M. Willebeek-LeMair, "Personel communication," 1995.