Runtime Incremental Parallel Scheduling on Distributed Memory Computers

Wei Shu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
shu@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 advanced parallel scheduling techniques to produce a low-overhead, high-quality load balancing and adapts to applications of nonuniform structures.

1. Introduction

One of the challenges in programming distributed memory parallel machines is to schedule work to processors [13]. 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*. In the later case, it extends another level of difficulty for scheduling 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*. The static scheduling distributes the work load before runtime, and can be applyed to static problems. Most existing static scheduling algorithms are sequential, executed on a single processor system. The dynamic scheduling performs scheduling activities concurrently at runtime, which applies to dynamic problems. Although a dynamic scheduling can apply to static problems as well, we usually use static scheduling for static problems because static scheduling provides a more balanced load distribution than dynamic scheduling.

The static scheduling utilizes the knowledge of problem characteristics to reach a global optimal, or near optimal, solution with well-balanced load. It has recently attracted considerable attention among the research community [12, 25, 11, 26, 6]. However, static scheduling has not been widely used in real-world applications yet. The quality of scheduling heavily relies on accuracy of weight estimation. The requirement of large memory space to store the task graph restricts the scalability of static scheduling. 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 [9, 10, 21]. However, most runtime scheduling algorithms utilize neither the characteristics information of application problems, nor global load information for load balancing decision. Efforts to collect load information for a scheduling decision certainly compete the resource 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 well-balanced load without incurring large overhead. With advanced parallel scheduling techniques, this ideal scheduling becomes feasible. In a parallel scheduling, all processors are cooperated together to schedule work. Some parallel scheduling algorithms have been introduced in [18, 8, 5, 7, 24]. A 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. Parallel scheduling is scalable to massively parallel systems. The overhead is not larger than many commonly used dynamic scheduling algorithms. Global parallel scheduling opens a new direction for runtime load balancing. As an alternative strategy to the commonly used dynamic scheduling, it provides a high-quality, low-overhead scheduling.

In this paper, we propose a new method, named *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*. An overview of RIPS is given 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. RIPS System Overview

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 all the computations are completed.

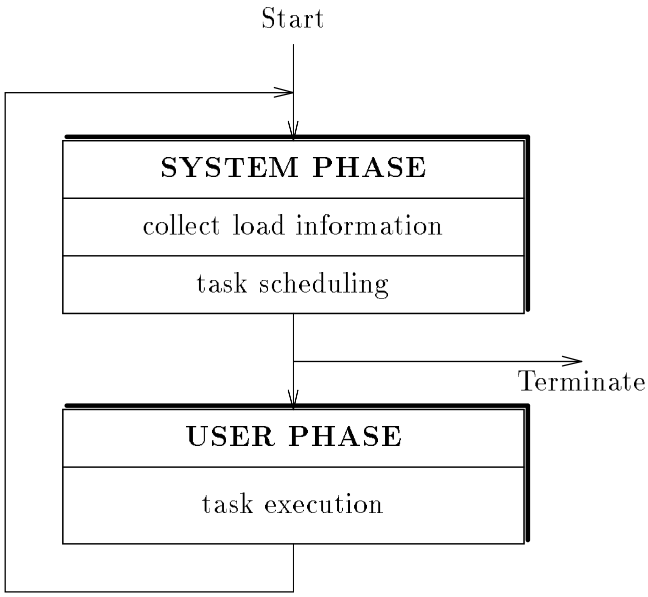


Figure 1: Runtime Incremental Parallel Scheduling (RIPS).

RIPS and the static scheduling share some common ideas. Both of them utilize the systemwise information and perform scheduling globally to achieve high-quality of load balancing. They also clearly separate the time to conduct scheduling and the time to perform computation. But, RIPS is different from the static scheduling in three aspects. First, the scheduling activity is performed at runtime. Therefore, it can deal with the 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 a better scalability for massively parallel machines and large size applications.

RIPS is similar to the dynamic scheduling in a certain degree. 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, there exist substantial differences, making them appear as two separate categories. First, the 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 potentially offers 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 in RIPS, the scheduling activity 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

There is another category of scheduling which carries out computation and scheduling alternately. It is sometimes referred as *prescheduling* which is more closely related to RIPS. Fox *et al.* first adapts prescheduling to application problems with geographical structures [14, 18]. Some other works also deals with geographically structured problems [8, 4, 3]. The project PARTI automates prescheduling for nonuniform problems [19, 5]. The domain exchange method (DEM) is a parallel scheduling algorithm applied to application problems without geographical structure [7]. It balances load for independnt tasks with an equal grain size. The method has been extended by Willebeek-LeMair and Reeves [24] so that the algorithm can run incrementally to correct the imbalanced load due to varied task grain sizes. The DEM scheduling algorithm generates redundant communications, therefore leading to a large communication overhead. It is designed specifically for the hypercube topology and implemented much less efficiently on a simpler topology such as a tree or a mesh. RIPS uses optimal parallel scheduling algorithms, which is also applied to non-geographically structured problems. 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.

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

In a RIPS system, the system phase and the user phase are clearly separated and interleaved. It is easy to enumerate phases by an index variable k. Hence, the system is in progress as increment of the index variable k,

$$S_0, U_0, S_1, U_1, S_2, U_2, ...$$

where S_k represents the kth system phase, and U_k the kth user phase.

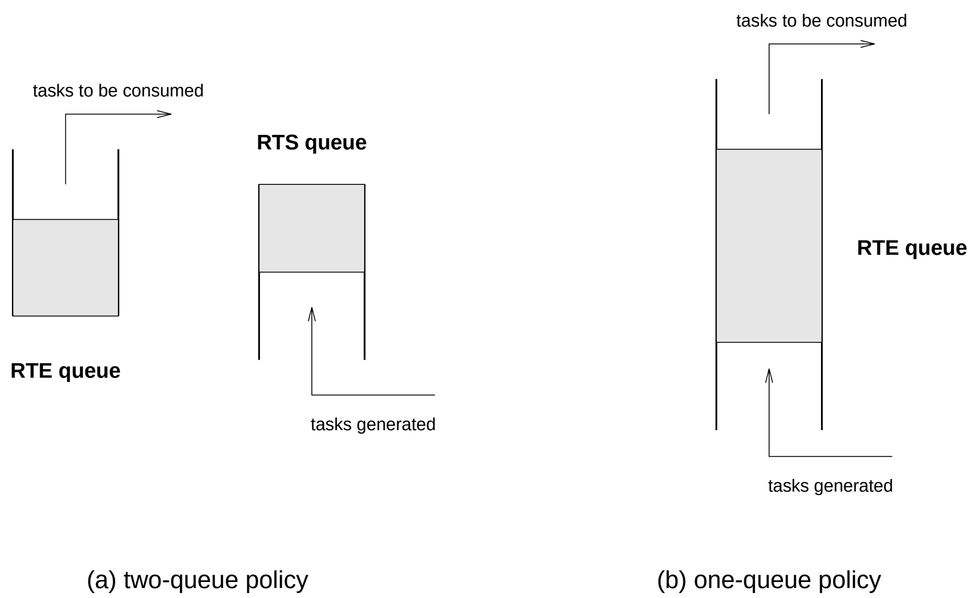


Figure 2: RTE and RTS Queues

There are two queues in each processor. One is called *ready-to-execute (RTE)* queue, the other *ready-to-schedule (RTS)* queue. A newly generated task could enter to the RTE or RTS queue, depending on different policies. In the policy that uses both RTE and RTS queues, the tasks in the RTE queue are consumed in the user phase, and the newly generated tasks enter the RTS queue, as shown in Figure 2(a). When the RTE queue becomes empty, the processor is ready for the next system phase. In the system phase, the old tasks left in the RTE queue, if any, are moved to the local RTS queue. Then, the tasks in every RTS queue participate in scheduling and enter to the local RTE queue or a remote RTE queue depending on the scheduling results. In the policy that uses only the RTE queue as shown in Figure 2(b), the newly generated tasks enter the RTE queue directly.

The policy that uses two queues involves large 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 the scheduling, the pointer moves down to let the entire queue become the RTE queue, eliminating the task copying from the RTS queue to the RTE queue. On the other hand, 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

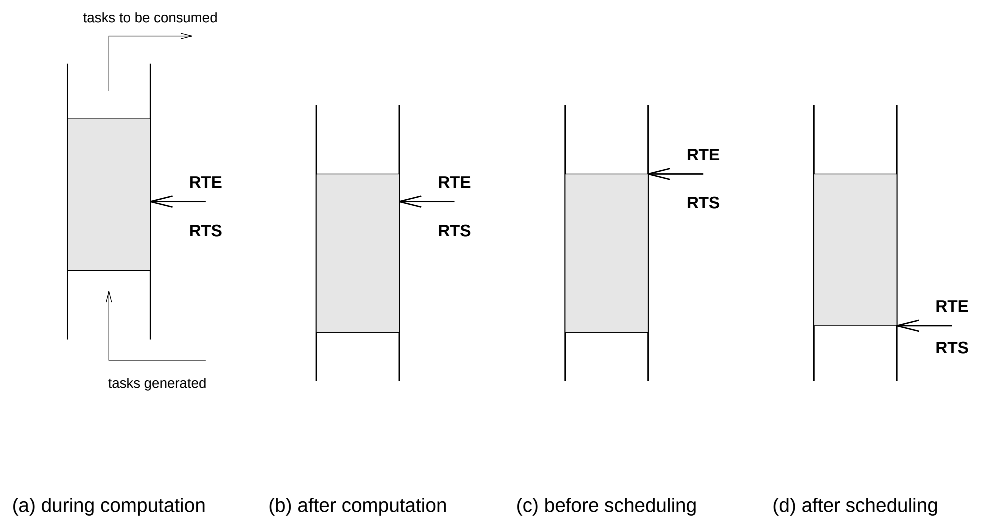


Figure 3: Implementation of RTE and RTS Queues

useful for an adaptive algorithm that changes between a one-queue policy and a two-queue policy.

Typically, a runtime scheduling algorithm has four components: a transfer policy, a selection policy, a location policy, and an information policy [21]. The transfer policy determines whether a processor is in a suitable state to participate in a task transfer. The selection policy determines which tasks should be transferred. The location policy determines to which processor a task selected for transfer 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 in RIPS depends on the queuing policy mentioned above. In this section, we emphasize the transfer, policy which is the central component of the incremental scheduling. The location policy and the information policy are two major components of the parallel scheduling, which will be discussed in the next section.

The transfer policy in RIPS includes two components: 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 tasks 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. In fact, these two local policies correspond to the two queuing polices. They are illustrated as follows:

	Queuing policy	Newly generated tasks
Eager	two queues, RTE and RTS	enter to RTS
Lazy	one queue, RTE	enter to RTE

The eager scheduling is implemented with the RTE queue and the 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 equal number of tasks ready to execute. During the user phase, new tasks can be generated and entered into the local RTS queue, while the tasks in the RTE queue are consumed. When the RTE queue is empty, the processor is ready to transfer from a 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 will be moved back to the RTS queues and rescheduled together with the newly generated tasks. The system scheduling phase schedules tasks in all RTS queues and distributes them evenly to the RTE queues.

The lazy scheduling uses only a single queue, RTE, to hold all tasks. The tasks scheduled to the processor and the tasks generated at the processor are not distinguished. 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.

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

	Condition	Description
ALL	ALL($L_{RTE}(i,t) = 0$)	all RTE queues are empty
ANY	ANY($L_{RTE}(i,t) = 0$)	any RTE queue is empty

where, $L_{RTE}(i,t)$ is the length of RTE queues in processor p_i at time t.

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 user phase

U_k to system phase S_{k+1}; 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; and an interval that is too long may result in unnecessary processor idle. The optimal length of the interval is to be determined by experiment.

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 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 become an initiator to broadcast an *init* signal to all other processors. A processor, upon receiving the *init* signal, switches from a user phase to the next system phase. Because of communication delay, more than one processor could claim to be the initiator. As a result, a processor may received 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. Note that when an idle processor initiates a phase transfer, other processors may still be executing a task. 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.

Now we consider the phenomenon that the 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 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 unawareness of low-parallelism caused by the lazy scheduling. With the ANY-Lazy policy, the processors that are eligible as initiators may keep

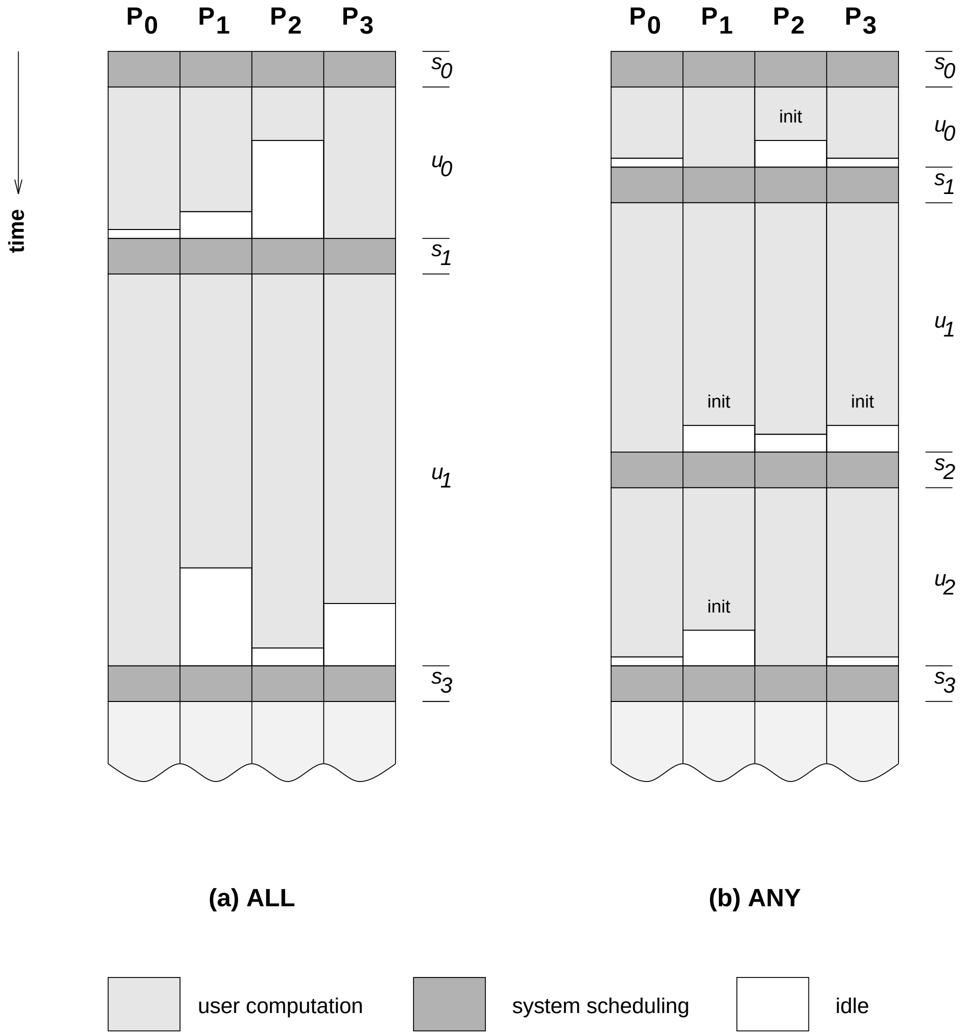


Figure 4: ALL and ANY policies

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. From another point of view, in a low-parallelism phase, the eager scheduling should always be applied to increase parallelism.

4. Parallel Scheduling

In this section we discuss *global parallel scheduling* algorithms, which is called *parallel scheduling* for short in the following context. In this category of scheduling, all processors execute a scheduling algorithm in parallel. The parallel scheduling algorithm utilizes the global information to provide a high-quality scheduling.

Parallel scheduling is different from the *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 the consideration of scalability, and no total load balance can be reached. A dynamic scheduling may be unstable and processor thrashing can occur.

Different from dynamic scheduling, 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 a high-quality scheduling and scalability simultaneously. Furthermore, parallel scheduling is stable because it is a synchronous approach.

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, named *Tree Walking Algorithm (TWA)*, is shown in Figure 5. Due to the unpredictable nature of the task grain size, each task is presumed to require equal computation time. Different processors accumulate different numbers of tasks. The goal of the algorithm is to schedule tasks so that each processor has the same number of tasks. The first step will be executed only at the system setup time, but not in every system phase. Steps 2 and 3 collect the system load information. In step 2, the total number of tasks is counted with a parallel 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 in 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.

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 4 communication steps to finish:

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.

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 achieves a well-balanced load and minimizes the data transfer and communication.

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

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

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

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: A Parallel Tree Walking Algorithm

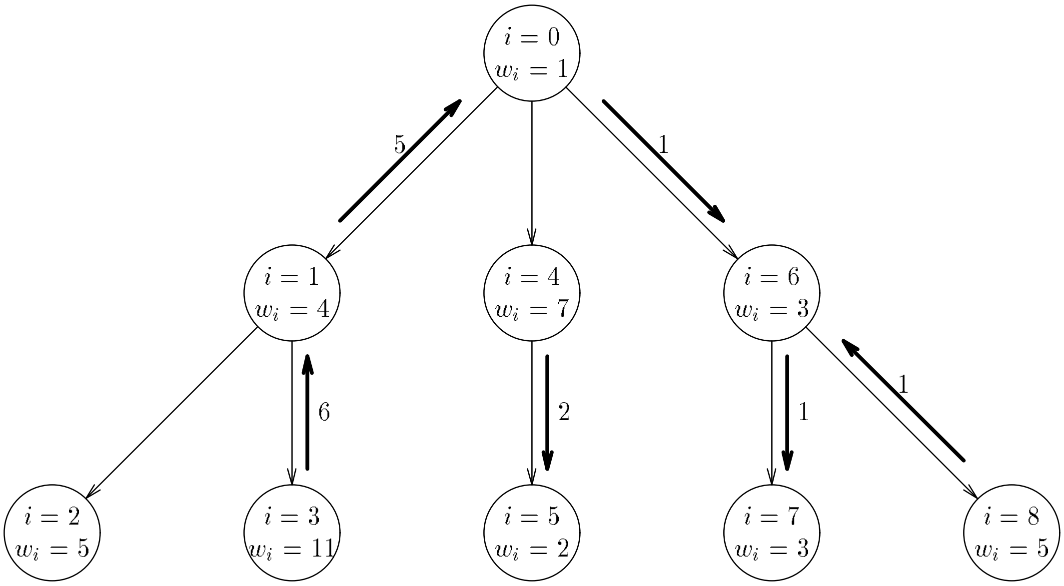


Figure 6: Example for the Tree Walking Algorithm.

Because

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

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

and

$$\sum_{j \; is \; child \; of \; i} Q_j = \sum_{j=i+1}^{i+N_i} q_j = w_{avg} * (N_i - 1) + r_i^*$$

where

$$r_i^* = \begin{cases} 0 & if \; i \geq R \\ N_i - 1 & if \; i \leq R - N_i \\ R - (i+1) & if \; R - N_i < i < R \end{cases}$$

$$w_i^{'} = w_i + (w_{avg} * N_i + r_i - W_i) + (W_i - w_i) - (w_{avg} * (N_i - 1) + r_i^*) = w_{avg} + r_i - r_i^*$$

Because $r_i - r_i^*$ is equal to 0 or 1, $w_i^{'}$ is equal to w_{avg} or $w_{avg} + 1$. □

Theorem 2: The Tree Walking Algorithm minimizes the total number of tasks to be transferred and the total number of communications.

Proof: For 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 transferred 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 transferred from the subtree.

For each subtree, if $Q_i \neq W_i$, then there is at least one communication between the subtree and its parent, which is the minimum number of communications. If $Q_i = W_i$, then there is no communication between the subtree and its parent. □

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 complexity 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		

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 a *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

The Modified Tree Walking Algorithm may have a negative impact in locality. In TWA, a node can keep the maximum possible 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. 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 the 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 distribute the communication traffic. The tree used in a 32-node system is shown in Figure 9.

The system has been tested with two application problems. The first one, the N-queen problem in exhausted search, is of a very irregular and dynamic structure. The number of tasks generated and the computation amount in each task are unpredictable. The second one, a molecular dynamics program named GROMOS, is a real application problem [23, 22]. The test data for GROMOS is the bovine superoxide dismutase molecule (*SOD*), which has 6968 atoms [20]. The cutoff radius is predefined to 8 Å, 12 Å, and 16 Å. GROMOS has a more predictable structure. The number of tasks is known with the given input data, but the computation amount in each task varies. Thus a load balancing mechanism is necessary. We use these problems to test the system behavior from completely irregularly to slightly irregularly structured problems.

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 is 14-Queen and GROMOS with the cutoff radius of 8 Å. 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.

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.

In general, the number of phases in the ANY policy is larger the depth of the tree. Note that

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:

$$node[n].level = i, \quad node[n].a = j \; mod \; 4$$

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 $node[i].level is unassigned$

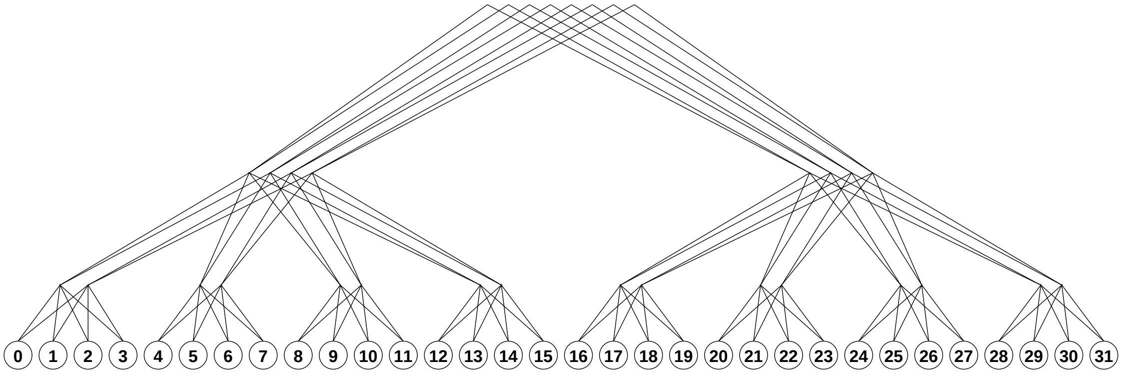
$$p = node[i].parent = nonLeaf[1][i/4]$$

$$node[i].a = node[p].numChild$$

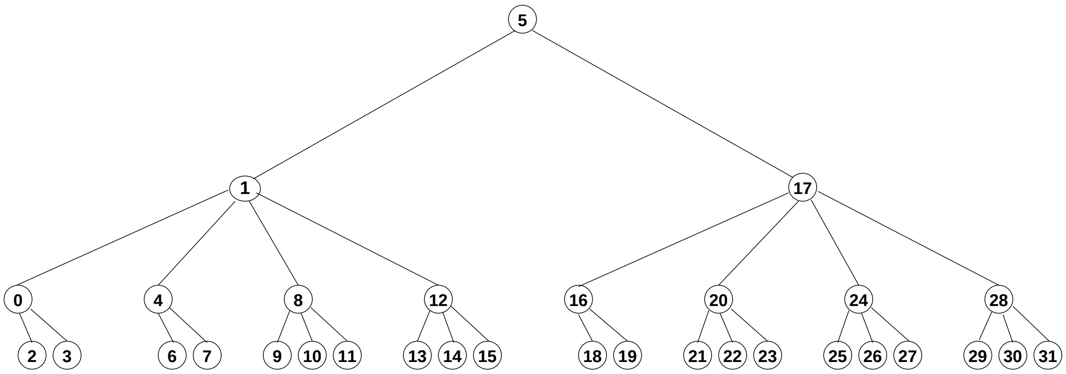
$$node[p].child[node[p].numChild] = i$$

$$node[p].numChild = node[p].numChild + 1$$

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

the number of tasks scheduled may excess the number of tasks, which means that a task may be scheduled more than once. With the ANY-Lazy policy, since some tasks may not be scheduled, the number of tasks scheduled could be larger or smaller than the total number of tasks.

Table II: Policy Comparison

			# of phases	# of tasks	# of tasks scheduled	# of comm.	overhead (seconds)	idle time (seconds)	exec. time (seconds)
14-Queen	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 $\mathring{A}$)	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

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 in the ALL policy is less than that in the ANY policy. Most communications in the ANY policy come from many simultaneous *init* messages. The number of communications in the lazy scheduling is less than that in the eager scheduling, because the number of phases in the lazy scheduling is smaller than that in 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 times in one processor. The ALL policy involves less phases and less 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, as 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 balance due to the grain size variation. A processor that 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 a larger overhead, but reduces processor 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 do an extensive comparison since both algorithms perform about the same. Therefore, only the performance with the Tree Walking Algorithm is shown below.

In Table III, we compared RIPS to two other dynamic load balancing strategies: random allocation and gradient model. A randomized allocation strategy dictates that each processor, when it generates a new task, should send it to a randomly chosen processor [1, 2, 9]. One advantage of this strategy is simplicity of implementation. 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 few 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. Second, the 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, compared to a system using other load balancing strategies that encourage locality.

In the gradient model [17, 16, 15], 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.

Table III: Comparison of Three Scheduling Algorithms

		# of tasks	# of non-local tasks	overhead (seconds)	idle time (seconds)	exec. time (seconds)	efficiency
13-Queen	Random	7579	7342	0.40	0.08	1.50	68%
	Gradient Model	7579	4255	0.97	0.88	2.87	36%
	RIPS	7579	314	0.29	0.05	1.36	75%
14-Queen	Random	11166	10832	0.62	0.25	7.14	88%
	Gradient Model	11166	6305	1.28	5.45	13.0	48%
	RIPS	11166	645	0.52	0.08	6.87	91%
15-Queen	Random	15941	15459	1.10	1.50	43.4	94%
	Gradient Model	15941	9058	2.10	30.5	73.4	56%
	RIPS	15941	925	0.97	0.23	42.0	97%
GROMOS (8 Å)	Random	4986	4831	0.26	0.56	4.10	80%
	Gradient Model	4986	2311	0.52	0.73	4.53	72%
	RIPS	4986	494	0.60	0.11	3.99	82%
GROMOS (12 Å)	Random	4986	4833	0.25	1.76	11.9	83%
	Gradient Model	4986	2184	0.51	1.40	11.8	84%
	RIPS	4986	556	1.16	0.36	11.4	87%
GROMOS (16 Å)	Random	4986	4832	0.26	3.94	25.0	83%
	Gradient Model	4986	2363	0.54	5.26	26.6	78%
	RIPS	4986	582	0.95	0.65	22.4	93%

The performance of RIPS shown in Table III 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 balance; (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. The randomized allocation, although it does not have a good locality, can balance the load fairly well. The gradient model does not show good performance for the N-Queen problem. However, it produces a fairly good performance for 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. In RIPS, the Tree Walking Algorithm can balance the load very well as it evenly distributes the tasks 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 because many tasks are packed together for transmission, reducing communication overhead, whereas, in dynamic scheduling, tasks are distributed individually.

RIPS has been ported to a large CM-5 machine. Table IV shows a speedup comparison of three scheduling algorithms. The randomized allocation and RIPS scale well. However, RIPS generates a better performance. The gradient model does scale well because it spreads the load slowly.

Table IV: Speedup Comparison on Large CM-5

		Number of Processors			
		64	128	256	512
15-Queen	Random	57.0	107	208	361
	Gradient Model	29.1	38.0	39.1	38.8
	RIPS	60.3	116	225	402
GROMOS (16 Å)	Random	50.6	97.3	189	355
	Gradient Model	41.6	41.8	41.8	40.1
	RIPS	55.7	109	216	387

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 transferred and the number of communications. Furthermore, in a dynamic system, when it indents 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 transferred 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. Experiments showed that RIPS outperforms two other scheduling strategies, the randomized allocation and the gradient model.

Further research goes in two directions: reducing the idle time and reducing the system overhead. To reduce the idle time further, a more accurate load balancing algorithm needs to be designed. The Tree Walking Algorithm used in RIPS assumes that every task is of the same grain size. When tasks are not of the same grain size, this inaccurate scheduling can be corrected in the next system phase. Since the processor that assigned less work will trigger the next system phase, more system phases will result. An algorithm that schedules tasks with different grain sizes should be considered. To reduce system overhead, a number of techniques, such as conditional packing and locality optimization, are to be investigated. Also, an algorithm should be studied to reduce the communication steps and processor suspension, and to maintain the locality.

Acknowledgments

We are very grateful to Reinhard Hanxleden for the GROMOS program, Terry Clark for the SOD data, and M. Feeley for the elegant N-queen program. This research was partially supported by NSF grants CCR-9109114.

References

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

[2] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, August 1988.

[3] S. B. Baden. Dynamic load balancing of a vortex calculation running on multiprocessors. Technical Report Vol. 22584, Lawrence Berkeley Lab., 1986.

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

[5] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience, accepted for publication*, 1991.

[6] Y.C. 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*, November 1992.

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

[8] 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*, pages 583–590, 1989.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, SE-12(5):662–674, May 1986.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Eval.*, 6(1):53–68, March 1986.

[11] H. El-Rewini and H. H. Ali. Scheduling conditional branching using representative task graphs. *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 1991.

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

[13] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.

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

[15] L. V. Kale. Comparing the performance of two dynamic load distribution methods. In *Int'l Conf. on Parallel Processing*, pages 8–12, August 1988.

[16] F. C. H. Lin. *Load Balancing and Fault Tolerance in Applicative Systems*. PhD thesis, Dept. of Computer Science, University of Utah, August 1985.

[17] F. C. H. Lin and R. M. Keller. Gradient model: A demand-driven load balancing scheme. In *Int'l Conf. on Distributed Computing System*, pages 329–336, 1986.

[18] J.K. Salmon. Parallel hierarchical N-body methods. Technical report, Tech. Report, CRPC-90-14, Center for Research in Parallel Computing, Caltech, 1990., 1990.

[19] J. Saltz, R. Mirchandaney, R. Smith, D. Nicol, and K. Crowley. The PARTY parallel run-time system. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1987.

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

[21] Niranjan G. Shivaratri, Phillip Krieger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.

[22] Reinhard v. Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. Technical Report CRPC-TR92207, Center for Research on Parallel Computation, Rice University, April 1992.

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

[24] Marc Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel and Distributed System*, 9(4):979–993, September 1993.

[25] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[26] 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.