Low-overhead scheduling of nested parallelism

by S. F. Hummel
E. Schonberg

Nested parallelism has the potential not only to permit more parallelism than non-nested parallelism, but to result in better load balancing. However, nested parallelism will not be profitable unless the overhead of scheduling nested parallel constructs can be made nonprohibitive. Previous implementations of nested parallel constructs have been fairly expensive and therefore have not been able to exploit fine-grained nested parallelism. In this paper, we describe a run-time system that schedules a large subset of nested parallel constructs—those that run until completion without blocking—with very little overhead. Our run-time system is built around a novel scheduling policy and work queue. The scheduling policy permits efficient stack-based local-memory storage allocation for task data, which is particularly efficient for multiprocessor architectures with both shared and local memory, such as the RP3. The shared, nonlocking work queue allows processors to obtain tasks in just a few instructions, without sacrificing load balancing.

1. Introduction

This paper describes a scheme for low-overhead, dynamic scheduling of nested parallelism in FORTRAN programs on multiprocessors with shared and local memory. We consider closed parallel constructs, such as the PARALLEL DO loop [1], in which the iterations of the loop are independent and can be executed in parallel. For example,

```
PARALLEL DO I = 1,N
   A(I) = F(B(I), C(I), A(I))
END DO
```

A *task* is a program-code sequence that is the unit of scheduling on a processor. A single iteration of the PARALLEL DO I loop above is a task in our study.

Nested parallelism arises from the nesting of parallel constructs, as shown here:

```
PARALLEL DO I = 1,N
   . . .
   PARALLEL DO J = 1,M
      . . .
   END DO
   . . .
END DO
```

Each PARALLEL DO I task splits into M tasks when it

reaches the PARALLEL DO J loop. This permits the distribution of inner parallel work to available processors. In this example, nested parallelism is potentially profitable when the number of processors P is greater than the number of outer iterations N (each processor might execute $M \times N/P$ tasks).

Even when P is less than N, nested parallelism may improve *speedup* (execution time for a single processor divided by the execution time for P processors) by eliminating or reducing load imbalance. (A *load imbalance* occurs whenever some processors are idle while other processors are busy.) When there is no nested parallelism, if the iterations of the outer PARALLEL DO I loop have different execution times, some processors may finish executing their assigned iterations earlier than other processors, resulting in a load imbalance. When nested parallelism is exploited, processors that finish early can assist slower processors by executing inner iterations, thereby improving load balancing. On the other hand, if the overhead of run-time scheduling is too high, fine-grained parallelism from many small tasks is not worthwhile. Therefore, an important design consideration is the trade-off between run-time overhead and load balancing.

Loop transformations performed by compilers, including loop collapsing and coalescing [2], eliminate nested parallelism by merging a set of nested loops into a single parallel loop, thereby reducing run-time overhead. However, an outer loop cannot be coalesced with an inner loop if the outer loop contains statements that are not in the inner loop. Vectorization is a compiler technique for mapping program parallelism onto processor vector units, but vectorization can be performed only on inner parallel loops; outer parallel loops must still be scheduled dynamically.

Typically, schemes for managing nested parallelism dynamically allocate resources from shared pools [3–5]. Task scheduling is performed by means of a shared queue. Whenever a processor becomes idle, it obtains another task from the shared queue until no tasks remain. This scheduling strategy, called *self-scheduling*, balances the workload because processors are idle only if there are no tasks on the queue. Similarly, task stack frames (the storage blocks for the private variables of tasks) are allocated from a shared-memory storage pool, so that inner-loop tasks can access the task stack frames of outer-loop tasks.

Shared resource pools improve overall load balancing at a cost [6]: Accessing shared memory is expensive, and the accompanying processor synchronization is a source of contention, particularly if locking is required. The shared scheduling queue is an obvious source of contention, so it is desirable to use a queue that reduces lock overhead. Many nonlocking queue algorithms for scheduling [7, 8] synchronize by using atomic read-modify-write instructions, such as *fetch&add*, *fetch&store* [9], and *compare&swap* [10], although none is designed specifically for nested parallelism.

To reduce the cost of memory allocation, schemes for static stack-frame allocation have been proposed in [4] and elsewhere. These schemes decrease the overhead of storage management by reducing the number of stack-frame allocation operations. However, storage estimates used for the allocations must be conservative, thus large, and the schemes still require a shared-memory storage pool for the stack frames. Furthermore, it is difficult to implement recursion by static stack-frame allocation. Thus, static allocation is not practical for languages with both parallelism and recursion.

We describe a scheme for scheduling nested parallelism suitable for architectures that have both local and shared memories, such as the RP3 parallel processor [11] and the BBN Butterfly architecture [12]. These machines are sometimes called NUMA (Non-Uniform Memory Access) machines. Our scheme reduces run-time overhead by means of a centralized scheduling policy that permits local-memory stack-frame allocation, and a novel, nonlocking "optimistic-queue" algorithm for scheduling. Thus, this scheme preserves the load-balancing properties of dynamic scheduling while reducing contention, by localizing storage management and eliminating locking. In particular, stack-frame allocation is a simple stack operation, and accessing the scheduling queue is usually no more expensive than manipulating a singly linked list.

This run-time management scheme has been implemented to execute PTRAN programs [13] on the RP3. PTRAN is a program analyzer, developed at the IBM Thomas J. Watson Research Center, that automatically parallelizes FORTRAN programs. The RP3 is a 64-way shared-memory multiprocessor, also designed and built at IBM, with a three-level memory hierarchy (cache, local, and global memory) and a network connecting all the processors [11, 14]. We present experimental results that verify the performance advantages of using local-memory stack-frame allocation and of using our nonlocking scheduling-queue algorithm on the RP3.

Before describing our run-time system, we present some background material. In the following section, we give an overview of dynamic run-time scheduling strategies in general, and in Section 3, we discuss the scheduling of nested parallelism in particular. In Section 4, we present our scheduling policy—which permits both dynamic scheduling and local-memory stack-frame allocations. Sections 5 and 6 describe details of stack-frame management and our scheduling-queue algorithm, respectively. The RP3 experiment is described in Section 7. Finally, after discussing potential enhancements to the run-time system in Section 8, we draw conclusions in

Section 9. Code for the scheduling queue is in the Appendix.

2. Overview of dynamic run-time scheduling

Parallel-program scheduling is typically performed by the run-time library rather than by the operating system [1, 6]. A run-time library can be precisely optimized for a specific programming model and semantics, while operating system kernel primitives must be general enough to accommodate a variety of programming models and languages [6, 8]. To perform scheduling by the run-time library, an application program, at the start of execution, invokes the operating system to create a set of operating system threads (lightweight processes), called *virtual processors*. The virtual processors schedule and execute the program tasks without further operating system assistance. Not all operating systems provide threads, in which case full-blown processes must be used for virtual processors. Operating system threads are preferable for virtual processors because threads share an address space, so that creating them is much cheaper than creating processes.

Some operating systems, including the RP3 operating system [14], allow parallel programs to *bind* virtual processors to physical processors, which prevents the operating system from suspending the execution of the virtual processors and moving them to different physical processors. Moving a virtual processor to a different physical processor is expensive, and suspending the execution of a virtual processor reduces the number of physical processors assigned to a program, hence, the amount of actual parallelism of the program. For the rest of this paper, we assume that virtual processors are bound to physical processors and refer to virtual processors simply as processors.

Table 1 shows the complexity of run-time scheduling necessary for different parallel language paradigms.

Scheduling can be performed for a simple loop with just two shared counters [4]: one counter (named *multiplicity*) to regulate assignment of iterations to processors, and the other counter (*number_left*) to determine when all iterations are complete. Variations of simple self-scheduling include schemes for distributing work in *chunks* of iterations, of either uniform [15] or decreasing size [16–18] in order to reduce overhead. The schemes described in [16–18] are designed to reduce the potential load imbalance of chunking.

Scheduling nested parallel constructs requires a more general mechanism, such as a queue, since the scheduling of separate inner parallel constructs is performed concurrently. The tasks of each individual parallel construct are scheduled with two counters, as described above. (Another approach, described in [3], uses a compiler-generated precedence table to control the self-scheduling order.)

Table 1 Scheduling complexity required for language features.

Language feature	*Complexity required*
Simple loop	Counters
Nested parallelism	Scheduling queue
No explicit synchronization	Run until completion
Explicit synchronization	Block/unblock facility
More operating system capability	Preemption, priorities

A large class of parallel constructs with no explicit synchronization operations can be scheduled using a "run-until-completion" paradigm wherein a task, once it begins to execute, may not be *blocked* (descheduled) before it finishes. The advantage of run-until-completion scheduling is its low overhead. For example, iterations from the same parallel construct scheduled on the same processor can use the same storage block for their task stack frames. As a result, the total amount of storage that is necessary for task stack frames is a function of the product of the maximum nested depth and the number of processors [4].

However, run-until-completion scheduling for more general parallel programs can lead to deadlock. For programs with explicit synchronization operations, the scheduler must be able to block tasks before they finish, and subsequently *unblock* (reschedule) them. Examples of explicit synchronization operations are *post event* and *await event*. A task that executes an *await event* operation cannot proceed until another task executes a corresponding *post event* operation. If tasks cannot be blocked and there are more parallel tasks than processors, a task executing an *await event* can wait forever for the task that executes the corresponding *post event* to be scheduled.[1] Reuse of task stack frames is no longer always possible when the potential for blocking exists, and the issues of task migration and local versus global queues become relevant.

More complex tasking models require further operating system functionality. For example, Ada scheduling requires priorities and preemption [8]. Even when additional operating system functionality is required, library scheduling is not as expensive as operating system scheduling.

The scheme described below implements run-until-completion scheduling for nested parallel constructs. Section 8 returns to the issue of explicit synchronization.

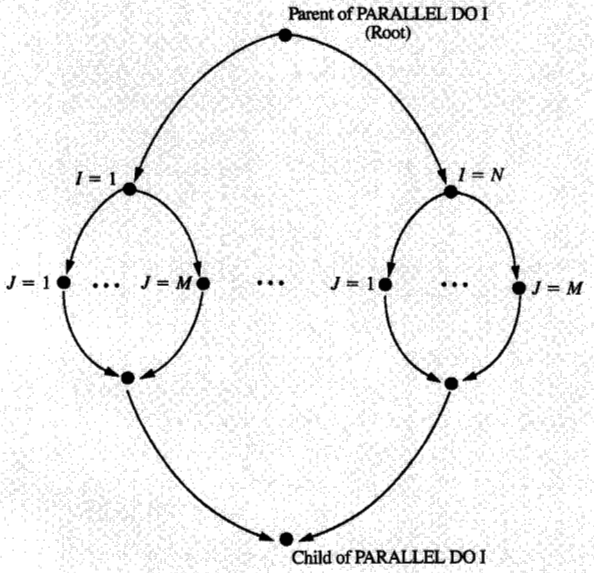
3. Nested parallelism

Run-time scheduling assigns processors to tasks in a *task-execution graph*, in which the nodes represent tasks and the edges represent task creation/termination operations. A

[1] In [19], compiler-generated precedence information prevents deadlock, so that run-until-completion scheduling is still possible.

```
PARALLEL DO I = 1,N
   PRIVATE X,Y
   . . .
   PARALLEL DO J = 1,M
      PRIVATE Z
      . . .
   END DO
   A(I) = X
   . . .
END DO
```

(a)



(b)

Figure 1

Example of parallel code with private variables: (a) code; (b) task-execution graph.

task-execution graph for the program in **Figure 1(a)** is shown in **Figure 1(b)**. The predecessor nodes (successor nodes) of a node are called the *parent tasks* (*child tasks*) of the node. In Figure 1(b), the task $I = 1$ is the parent task of its M child tasks $J = 1, \cdots, M$. [Note that the code in Figure 1(a) beginning with the statement `A(I) = X` is called a child task, with M parents corresponding to the M iterations of the `PARALLEL DO J` loop.]

A `PARALLEL DO` loop iteration may have private variables, which are declared inside the loop. A private variable of an iteration is shared among all iterations nested within the iteration. For example, a distinct copy of the private variables X and Y declared in Figure 1(a) is allocated to the iteration $I = 1$ in Figure 1(b), and this copy of X and Y is shared among the M children of $I = 1$.

A task stack frame is a storage block containing the private variables of an iteration. In addition, each task stack frame contains a pointer to the stack frame of its parent task. During execution, the active stack frames form an inverted tree, called a *cactus stack* [20]. A cactus stack for Figure 1(b) is illustrated in **Figure 2** (at some instant when $M \times N$ processors are executing all of the `PARALLEL DO J` iterations concurrently). The root node is the task stack frame of the parent of the `PARALLEL DO I` loop, and the other nodes are task stack frames for the `PARALLEL DO I` and `PARALLEL DO J` iterations. The arrows point to parent stack frames.

The execution of a `PARALLEL DO` statement can be broken down into the following three actions:

- *Fork.* The parent task allocates a *parallel control block* (PCB), initializes it, and stores it on a shared queue. The PCB, which is used to schedule all iterations of the loop, contains the address of the parent-task stack frame, the address of the loop body code, the two counters needed to schedule the iterations (see the preceding section), and other information.
- *Context switch.* An idle processor obtains a task (an iteration) to execute from a PCB on the queue by decrementing the *multiplicity* counter of the PCB. To begin executing the task, the processor branches to the loop-body code address.
- *Initialization.* At the beginning of task execution, storage for the private variables of the iteration (its stack frame) is allocated, and a link to the parent stack frame (the cactus-stack link) is saved. (On architectures in which the cache is managed by software, such as the RP3, cache invalidations are also performed.)

When all iterations in a parallel construct have been started (*multiplicity* = 0), the corresponding PCB may be removed from the queue. When all iterations in the construct are complete (*number_left* = 0), the child task corresponding to the code that follows the construct is executed.

The cost of these three steps depends partly on the *scheduling policy*, which determines how processors are assigned to the tasks in the task-execution graph. Our scheduling policy is designed to reduce these costs, as described in the next section.

4. Scheduling policy

Our scheduling policy consists of three rules that determine what a processor should do at each task creation/termination point. The first two rules are common to several run-time systems [4, 5], while the third rule is

not. The third rule permits local-memory allocation of stack frames.

Let b designate the PCB of some parallel construct PC.

Rule 1 A processor that executes a task from b continues executing tasks from b while unassigned tasks remain. If none remain, it searches for work on the scheduling queue.

The advantage of this rule is that both context switching and initialization can be performed at low cost when a processor is able to obtain a sequence of tasks from the same PCB. For each task in the sequence after the first, the processor does not have to search the scheduling queue nor allocate a new task stack frame. The same stack frame is reused for all of the tasks the processor executes.

Let $b_1, \cdots, b_q$ be the PCBs on the queue at a given time, and let P_i be the number of processors assigned tasks from b_i $(1 \le i \le q)$ at that time. The performance advantages of Rule 1 are best realized for a given b_i if P_i is much smaller than the original number of tasks N_i in b_i. There is no reuse of the task stack frames if $P_i = N_i$, and there is no parallelism for b_i if $P_i = 1$. We say that the processors are *evenly distributed* at a given time if $|P_i - P_j| \le 1$, for $1 \le i, j \le q$, and if no processors remain idle when the queue is not empty. Generally, even distribution results in a high degree of stack-frame reuse without decreasing parallelism.

FIFO (first in, first out) scheduling does not achieve even distribution. For example, in Figure 1(b) strict FIFO ordering causes all N of the outer tasks to be scheduled before any of the $N \times M$ inner tasks, since the PCB for the outer parallel construct is first on the queue. Rule 2 below, which is not FIFO, achieves a more even processor distribution.

Rule 2 After a processor completes the fork operation that allocates b, it obtains its next task from b.

Thus, tasks from b are executed by the *owner* of b (the processor that allocates it) and perhaps by *helpers* (processors that find b on the queue). If no processors are available, the owner may complete all of the tasks in b without any helpers. With Rule 2, processor distribution is "even" when there is at most one helper processor, because (a) with no helper processors, each PCB on the queue has one processor, the owner, executing its tasks, (b) helper processors are assigned to the first PCB on the queue, (c) two or more helper processors give the first PCB three or more processors, while the other PCBs have only one. Section 8 discusses improvements for better processor distribution.

To implement Rule 2, the queue algorithm must allow task assignment from interior PCBs (PCBs that are neither the first nor the last one on the queue). Since each owner executes tasks from the PCB that it enqueues, all enqueued PCBs are being "consumed" concurrently, and

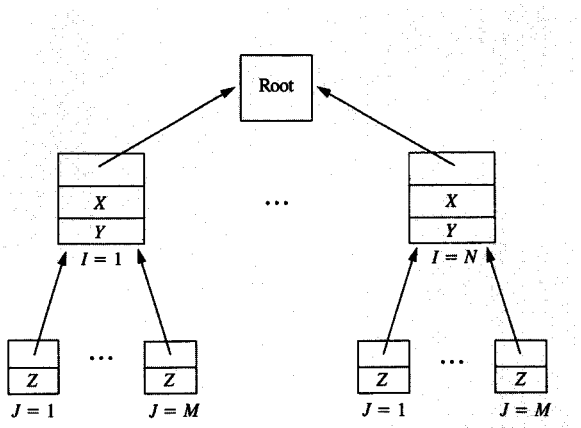


Figure 2

Cactus stack for the example of Figure 1.

some of these may become empty before becoming first on the queue. Queue algorithms that allow interior entries to become empty (perhaps removed from the queue) are more complex and typically more expensive than the more usual FIFO queue algorithms. Our queue algorithm, as described in Section 6, is designed to accommodate empty interior entries efficiently without removing them.

The third rule permits local-memory stack-frame allocation.

Rule 3 After all of the tasks in b are finished, the owner of b executes the child task of the iterations of the parallel construct PC—the code immediately after PC. (This may require the owner to wait, which is discussed below.)

The advantages of this rule are the following:

- The cactus stack can be implemented by P processor stacks, allocated in the local memories[2] of the P processors. (This is verified in the following section.) Stack frames are allocated by simple stack operations, with no synchronization or locking.
- PCB allocation can also be implemented by P individual processor stacks in global memory, again without synchronization or locking. (See the following section.)

Without Rule 3, both task stack frames and PCBs must be allocated from a shared-memory storage pool, which is more expensive. However, the disadvantage of this rule is that it may cause processors to have a load imbalance. The best load balancing is obtained when the processor that

[2] In the RP3 system, the local memory associated with any processor could be accessed by all other processors, although with greater delay.

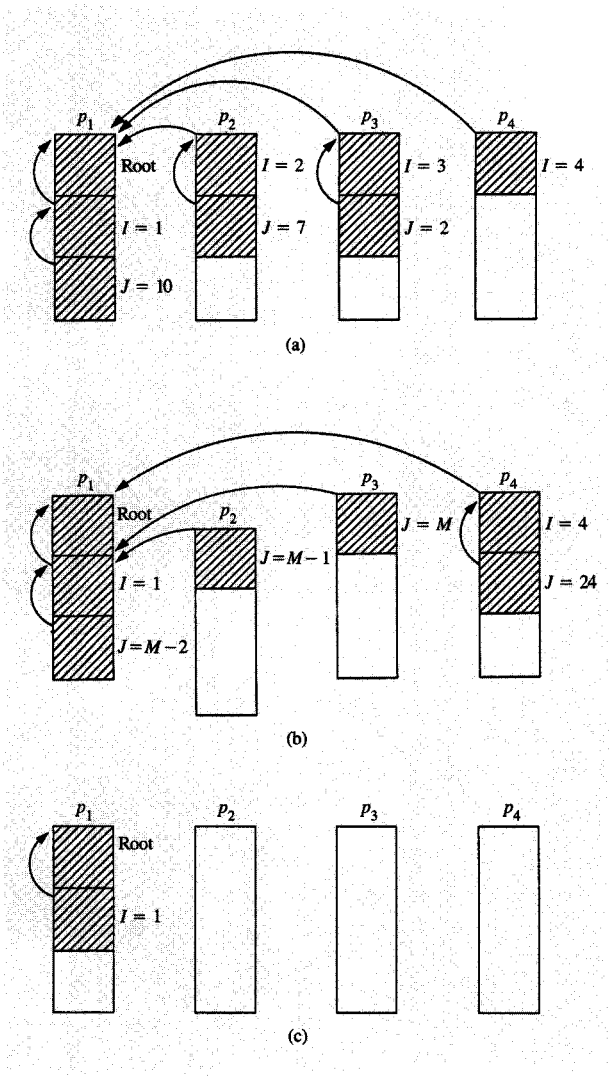


Figure 3

Three snapshots of four processor stacks, showing the cactus stack of the example of Figure 1 at different times.

executes the last task to finish from b executes the code after PC (the child task). Under Rule 3, the owner of PC must wait until all tasks from b are finished. However, the cost of this idle time must be weighed against the savings from local storage allocation. In Section 7, where we return to the issue of load balancing versus local-memory storage allocation, we argue that the savings outweigh the costs, in particular when P is large.

5. Cactus-stack management

While, in general, cactus-stack management requires a central storage pool and locking, it is possible to implement the cactus stack by means of P individual processor stacks because of Rule 3. We justify this claim as follows. First, the same processor that allocates a task stack frame always deallocates it. Therefore, each processor can maintain its own pool of task stack frames, and no locking is necessary. Furthermore, the stack frames themselves can be allocated in strict LIFO (last in, first out) order, since the owner processor of PC always deallocates all stack frames generated during the execution of PC by the time that PC completes. Because of the LIFO order, the local pool is, in fact, an ordinary stack, and allocation involves merely incrementing a stack pointer and setting a link to the parent stack frame.

To illustrate how the cactus stack works, we consider the abstract cactus stack of Figure 2 and the individual stacks of four processors $p_1, \cdots, p_4$ that are executing the program of Figure 1(a). **Figure 3** shows three snapshots of the stacks at different times during the execution of the PARALLEL DO loops. Figure 3(a) shows p_1 executing the inner $I = 1, J = 10$ task; p_2 executing the $I = 2, J = 7$ task; p_3 executing the $I = 3, J = 2$ task; and p_4 executing the $I = 4$ task. Since p_1 executed the parent of the PARALLEL DO I loop, its stack has a task stack frame for the root of the cactus stack, and all stack frames for PARALLEL DO I tasks have pointers to this root-task stack frame. In Figure 3(b), p_2 and p_3 have finished their PARALLEL DO I iterations and are helping p_1. Processor p_2 is executing the next-to-last iteration ($I = 1, J = M - 1$), and p_3 is executing the last iteration ($I = 1, J = M$). Since p_2 and p_3 are helpers, their task stack frames point to the $I = 1$ task stack frame. Processor p_4 is executing iteration $I = 4, J = 24$. Figure 3(c) shows p_1 executing the rest of the $I = 1$ iteration (the child) after the PARALLEL DO J loop has finished (and the other processors are idle because the queue is empty). Since p_1 executes both the parent and child task of the PARALLEL DO J loop, the task stack frame for $I = 1$, which is needed for the child task, is available on the p_1 stack after the PARALLEL DO J loop has finished.

Although there is one stack per processor, each stack must be accessible by all processors. For example, p_2 and p_3 must be able to access the p_1 stack while they are helpers [see Figure 3(b)]. On the RP3 system, the local memory of one processor can be addressed by other processors, so in fact the processor stacks can be allocated in local memory of the individual processors. For architectures on which the memory of a processor cannot be addressed by other processors, the processor stacks can still be allocated in local memory provided that each stack frame contains a copy of its parent stack frame instead of a pointer to the parent stack frame. This increases the cost of forking and initializing stack frames.

PCB allocation is similar to task stack-frame allocation. Each processor has its own stack of free PCBs, although

in global rather than local memory, since the degree of concurrent access is expected to be high. When a processor executes a fork operation, it increments its PCB stack pointer to allocate the next PCB on its stack. After all tasks in that PCB are finished, the processor decrements the stack pointer to deallocate it. Thus, each PCB is part of two data structures—a processor stack and the shared scheduling queue—and appropriate synchronization is required when accessing the PCB. Part of the effort in the queue-algorithm design was to make this synchronization efficient.

6. Scheduling queue

The overhead of accessing a shared scheduling queue determines, to a large extent, the smallest task that is worth creating. Thus, it is important that the queue be, as far as possible, nonlocking. The larger the number of processors, the greater is the degradation due to locking. Accordingly, many queue algorithms have been designed to permit concurrent updating. These algorithms typically use atomic *read-modify-write* instructions, such as *fetch&*ϕ, which indivisibly carry out the following steps: read a variable, perform the (binary) operation ϕ on it, and store back the result. Read-modify-write instructions greatly simplify the coordination of multiple processors accessing a shared data structure such as a queue; therefore, they are provided on several multiprocessor architectures, including [11] and [12].

To schedule the nested parallel loops generated by PTRAN, we designed a new shared-queue algorithm. The queue is implemented as a singly linked list and uses the three read-modify-write instructions *fetch&increment* (*f&i*), *fetch&decrement* (*f&d*), and *fetch&store* (*f&s*), which respectively increment, decrement, and swap variables.

Our queue algorithm has many desirable properties: It has a very low overhead, in terms of both time and storage, it is nonlocking, and it permits empty interior entries in the following efficient manner. When all tasks in a PCB have been assigned to processors, we mark its storage block as empty, creating a hole in the queue, instead of removing the PCB; an empty PCB may be removed from the head of the queue only. Unfortunately, in the case of nested parallelism, leaving holes can result in long chains of empty PCBs. However, our algorithm *reuses* empty PCBs in place by reinitializing their storage blocks with new PCB data, so that these long chains of empty PCBs are not generated.

Our implementation achieves its efficiency by using an optimistic strategy wherein abnormal events (such as appending new PCBs to PCBs that are being removed) are allowed and corrected at a later time, rather than prevented. The design philosophy is to streamline common cases, while minimizing the adverse effect of unusual events. For example, because empty interior PCBs are not immediately removed and are sometimes reused, enqueuing, dequeuing, and updating PCBs are efficient operations. Rule 1 is another example of optimizing the common case. When a processor repeatedly obtains tasks from a particular PCB, the context-switch overhead is reduced.

The queue data structure and algorithm are described in detail later in this section. Below, we discuss related queue algorithms. The code implementing the queue is in the Appendix.

• *Related work*

Concurrent-access queue algorithms differ in several aspects, including the types of operations permitted, the number of different operations that can be performed concurrently, and storage/time requirements. Most concurrent-access queues are FIFO and do not allow interior-entry removals. Increased parallelism and reduced execution time can often be obtained by using more storage.

The underlying data structures of concurrent-access queues can be arrays, linked lists, and trees. Although tree-based queues have storage requirements proportional to the number of queue entries and operation-time logarithmic in the number of entries, the overheads of the basic operations are several orders of magnitude higher than for array-based or linked-list-based queues (thousands as opposed to tens of machine instructions [8]), so we do not pursue them further here.

A simple queue can be built using an array of pointers to queue elements and two counters, *first* and *last*, which are updated with *f&i* instructions modulo the array size [9, 21]. The number of insertion and deletion operations that can be performed concurrently on such queues is determined by the array size; unfortunately, so is the amount of storage that must be dedicated. Array-based queue algorithms that use *compare_double&swap* (conditional swap of two double-words) to update the counters and permit concurrent enqueues and dequeues are given in [22] and [23].

Queues based on linked lists are implemented by maintaining two pointers, *head* and *tail*, which are updated by an indivisible swap instruction, such as *f&s*, *compare&swap*, or *compare_double&swap*. Singly-linked-list algorithms for concurrent enqueuing and dequeuing are presented in [8, 24, 25]. However, these algorithms permit only one dequeue operation at a time. The storage overhead of linked-list-based queues is minimal (a few words).

Hybrid queues have been developed to exploit the high degree of concurrent access of array-based queues and the low storage requirements of linked-list-based queues. These queues use arrays of linked lists, where each list

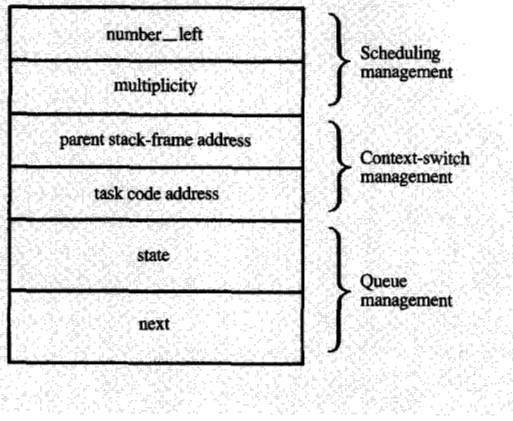


Figure 4

PCB data structure.

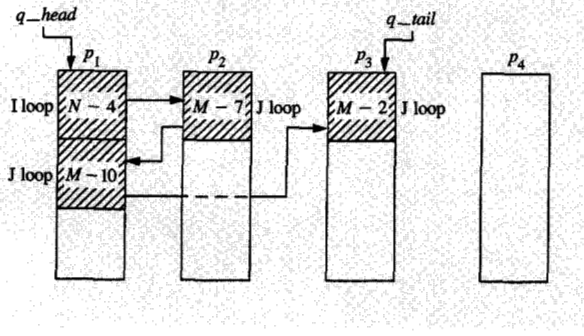


Figure 5

Snapshot of four PCB stacks with scheduling-queue links, at the same execution time as Figure 3(a). (The numbers in the PCB boxes are the values of corresponding *multiplicity* counters.)

must be locked while an operation is being performed on it [7], but operations can be performed on different lists concurrently. When the linked lists are implemented with indivisible swap instructions, only dequeue operations must lock a list [8].

In the case of our queue used for scheduling nested parallelism, each enqueued PCB is used to schedule multiple tasks (see Section 3). Queues whose entries represent more than one element are called *multi-item queues* [7]. By using a *f&d* operation to decrement the *multiplicity* counter in the PCB, processors can obtain work concurrently. Even if actual dequeue operations removing PCBs from the queue must be serialized, concurrent queue accesses to the first PCB are possible while it is not empty.

More concurrency is achievable when processors are allowed to obtain tasks from interior PCBs as well as the first PCB. However, interior PCBs may become empty. An obvious data structure for accommodating empty interior entries is a doubly linked list, since removing interior entries is straightforward. However, while interior-entry removal is being performed (updating the pointers of predecessor and successor entries), other operations are usually "locked out," in order that the list structure does not become corrupted. The nonlocking doubly-linked-list algorithms of which we are aware[3] have relatively high overheads and therefore have no advantage over a doubly linked list with locking.

For the implementation of Ada described in [8], the singly-linked-list queue algorithm permits multi-item and empty interior entries. Entries are not actually removed from the queue, but marked as empty and left as "holes." The storage block for an empty entry can be reused only after it has reached the head of the queue and been removed. If the enqueuer needs to enqueue some work prior to the entry removal, a new entry must be allocated. Our new queue algorithm avoids the need for this additional allocation by allowing entries left as holes to be "reused in place." Surprisingly, this improvement has led to the development of a completely nonlocking queue.

- *Queue data structure*

To implement our singly-linked-list multi-item queue, two pointers are maintained, *q_head* and *q_tail*, which point to the head and tail of this list. These pointers are changed using *f&s* operations. In addition to the fields described in Section 3, a PCB contains a field *next*, which is the queue-link field, and *state*, which is needed for coordination.

Figure 4 shows the fields of the PCB data structure. *Multiplicity*, *number_left*, and *state* are updated using *f&i* and *f&d* operations. The field *next* is updated using *f&s*. **Figure 5** shows a snapshot of four PCB stacks with scheduling-queue links corresponding to the execution state shown in Figure 3(a). The PCB stacks are associated with processors $p_1, \cdots, p_4$. Since p_1 executes the parent of the `PARALLEL DO I` loop, the PCB for the `PARALLEL DO I` loop, which is the queue head, is on the p_1 PCB stack. The next three PCBs on the queue represent the `PARALLEL DO J` loops of iterations $I = 2$, $I = 1$, and $I = 3$. For each PCB, the value of *multiplicity* is shown, which specifies the number of unassigned tasks.

The field *state* encodes stages in the processing of a PCB. *State* acts as an event counter, initialized to 0 and decremented/incremented after significant events. It is decremented whenever the PCB reaches the head of the queue, the PCB becomes empty, or the PCB is removed

[3] Symunix II Operating System, Ultracomputer Project, New York University.

Table 2 PCB-state transitions implemented by queue operations.

Queue operation	*Event*	*Operation performed on* state	*Fetched value of* state	*Action*
Produce	PCB is allocated	*f&i*	2	Reuse in place
			1,0	Wait until *state* is 1, then set *state* to 3 and perform *enqueue*
Enqueue	PCB is queue head	*f&d*	3	Finished
			2	Perform *dequeue*
Consume	PCB is empty	*f&d*	3	Look for work
			2	Perform *dequeue*
Dequeue	Removal complete	*f&d*	—	Continue traversal
Dequeue	PCB is queue head	*f&d*	3	Finished
			2	Remove queue head

from the queue. It is incremented to 3 after it is allocated from a PCB stack during a fork operation. The next section describes how *state* is used by the algorithm.

• *Queue implementation*

We identify four queue-related operations that processors perform:

- *Consume tasks.* A consumer looks for work, either on the queue or in a specific PCB.
- *Dequeue PCB.* A dequeuer removes empty PCBs from the head of the queue.
- *Produce tasks.* A producer updates a PCB (new or empty) with new task information during a fork operation.
- *Enqueue PCB.* An enqueuer appends a PCB onto the tail of the queue.

A producer sometimes turns into an enqueuer, and a consumer sometimes turns into a dequeuer. Producers and consumers do not alter the connectivity of the list, while enqueuers and dequeuers do.

Producing, consuming, and enqueuing can all be performed concurrently. Dequeuing can be performed concurrently with the other operations; however, there can be only one dequeue operation at a time. (This is verified in the subsection on correctness issues below.) Since a dequeue operation on an empty PCB is not performed when the PCB is reused in place, the number of dequeue operations, hence, the performance penalty from serialized dequeues, is small.

PCB-*state* transitions and processor actions based on PCB-related events are summarized in **Table 2.** A processor executing a queue operation detects a significant PCB event, performs a *f&i* or *f&d* operation on the *state* field of the PCB, and then performs the specified action based on the fetched value of *state*. The fetched value is the previous value of *state* before the *f&i* or *f&d* operation. This table is explained in more detail in the following subsections on queue operations; pseudocode for the operations is presented in the Appendix.

As an efficiency metric for each operation, we count the number of shared-data accesses required for the most common case of the operation. Shared-data-access time typically dominates the cost because of the distance from processors to global memory and because of contention among different processors accessing the same memory bank. To show the correctness of concurrent queue operations, race conditions must be ruled out. We systematically examine the interactions of all concurrent operations in the subsection on correctness issues to show that there are no harmful race conditions. This algorithm has been implemented and exhaustively tested, both by running the experiments presented in Section 7 and by inserting delays to force unusual events to occur.

Consume tasks

A consumer looks for work either on the queue or in a specific PCB (Section 4, Rule 1). First, we consider a consumer looking for work on the queue, using a private pointer *my_pcb* to traverse the queue. Pseudocode for this function is given in the procedure *consume_q* in the Appendix. For each successive PCB *b*, the following code is executed to try to obtain an unassigned task *my_task*:

```
if (my_pcb→multiplicity ≥ 0) {
   my_task = f&d (my_pcb→multiplicity);
   if (my_task > 0) {
        /* Success, execute my_task */
   }
}
```

(Note that more than one processor can execute the first statement before any executes the second, and that, as a result, multiplicity can be decremented to reach some negative value.) The initial test prevents *multiplicity* of *b* from underflowing when *b* is empty. Without the test, underflow can occur if idle processors repeatedly decrement *multiplicity* of *b* while busy-waiting. If the

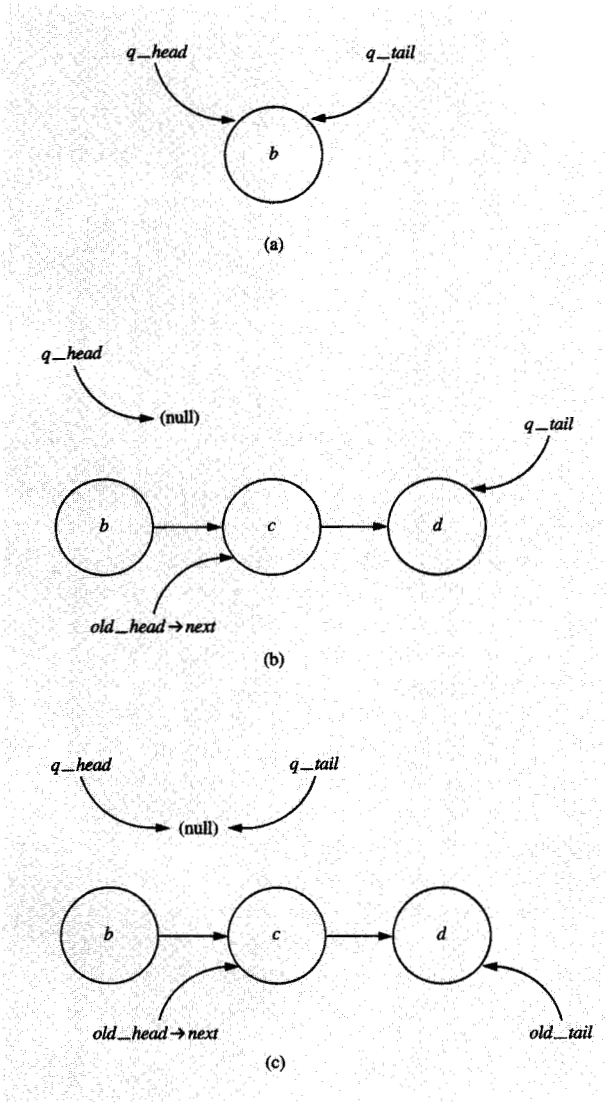


Figure 6

Three scheduling-queue configurations: (a) one-node queue; (b) *q_head* is null, and *c* and *d* are misplaced; (c) *q_tail* is null.

queue head is nonempty (the most common case), only three shared-data accesses are made: one for setting *my_pcb* to *q_head*, and two for accessing its *multiplicity*.

If *my_task* is found to be 0, *b* is empty, so *state* of *b* is decremented with a *f&d* operation. If the fetched previous value of *state* is 2, *b* is the queue head, so the consumer becomes a dequeuer.

Since each consumer uses its own private queue pointer, the queue can be traversed concurrently by more than one consumer, even while a dequeue operation is being performed. The following observation is important: While there may be many consumers concurrently decrementing *multiplicity* of *b*, only one consumer sets *my_task* to 0. Therefore, only one consumer decrements *state* of *b* and can become a dequeuer. We show that there can be only one dequeuer at a time and discuss other possible race conditions in the section on correctness issues.

Next consider a consumer looking for work in a specific PCB *b*. The procedure *consume_pcb* in the Appendix performs this operation. The consumer decrements *multiplicity* of *b* with a *f&d* operation, storing its previous value in *my_task*. If *my_task* is positive, the consumer has found a task to execute. Otherwise, if *my_task* is 0, the consumer decrements *state* of *b* and dequeues *b* if it is the queue head, as described above. In all cases, the consumer decrements *number_left* of *b*, the number of tasks from *b* that are still executing. When *number_left* is 0, the child of *b* can be executed. The most common case for *consume_pcb* (consumer finds work) requires only two shared-data accesses to decrement *multiplicity* and *number_left* of *b*.

Dequeue PCB

It is not obvious how to implement a singly-linked-list-based queue that permits concurrent dequeue and enqueue operations. When the list becomes empty, both the head and tail must be updated, and the queue is in an inconsistent state during these updates. PCBs enqueued during this time can therefore be lost. Our *fetch&store*-based solution to the empty-queue problem, in which there is only one dequeuer allowed at a time, is an extension of an algorithm described in [8].

A processor becomes a dequeuer when it discovers that a PCB is an empty queue head. The procedure *dequeue* in the Appendix implements the dequeue function. The dequeuer traverses the queue, removing PCBs by advancing *q_head*, until a nonempty PCB is found or the queue is empty. For each new queue head *b*, the dequeuer decrements *state* of *b*. If the previous value of *state* of *b* (fetched by the *f&d* operation) is 3, *b* is nonempty, so the dequeuer becomes a consumer again. Otherwise (the previous value is 2 and *b* is empty), the dequeuer sets *q_head* to *next* of *b*. Since *b* is now removed, *state* of *b* is decremented once again. At this point, *b* is fully dequeued, and the dequeuer continues its traversal.

In summary, a PCB *b* is completely dequeued after

- *state* of *b* has been decremented twice (once for being empty and once for being at the head of the queue).
- *q_head* has been set to *next* of *b*.
- *state* of *b* has been decremented a third time (for being removed).

Thus, the cost of actually dequeuing a PCB in the usual case is five shared-data accesses (including initially reading *q_head*).

If the dequeuer discovers that the queue is empty after updating *q_head* (*q_head* is `null`), special action must be taken for coordinating with possible concurrent enqueuers (see the subsection on enqueuing a PCB below). Enqueuers always update *q_tail* and update *q_head* only if the list is empty (*q_tail* is `null`). The following abnormal condition can arise.

Suppose *b* is the only PCB on the queue, so that both *q_head* and *q_tail* reference *b*, as shown in **Figure 6(a)**. The dequeuer removes *b* by setting *q_head* to *next* of *b*, which is `null` in this example. However, concurrent enqueuers append new PCBs to *b*, since *q_tail* still points to *b*, even though *b* is removed. These PCBs are "misplaced" and must be recovered.

To prepare for this possible abnormal event, the dequeuer saves a pointer to *b* in a variable *old_head* before updating *q_head*. After *q_head* is set to `null`, *old_head→next* is the head of the list of misplaced PCBs. **Figure 6(b)** shows misplaced PCBs *c* and *d* appended to *b*. To complete the dequeue operation, the dequeuer sets *q_tail* to `null`, saving its previous value in *old_tail* using a *f&s* operation [see **Figure 6(c)**]. Now *old_tail* is the tail of the list of misplaced PCBs and *q_head* and *q_tail* are in a consistent state. As a last step, the dequeuer appends the misplaced-PCB list to the queue.

This is an "optimistic solution" to the empty-queue problem. Rather than locking out enqueuers, to prevent misplacing PCBs while a dequeue operation is being performed, an inexpensive corrective action (enqueue operation) is taken so that misplaced PCBs are not lost and the integrity of the queue is maintained.

Produce tasks

During a fork operation, a producer allocates the next PCB *b* from its PCB stack, as described in Section 5, and initializes *number_left* and the context-switch-management fields. Our implementation guarantees that *b* is empty and no processors are executing tasks from *b* at the time it is allocated (see the subsection on correctness issues below). The procedure *produce* in the Appendix updates *state*, *multiplicity*, and *next* of *b*.

To indicate that *b* is no longer empty and to determine its status with respect to the queue, the producer increments *state* of *b* with a *f&i* operation, fetching its previous value (see Table 2): *b* is already enqueued if its previous value is 2; it is dequeued if its previous value is 0; and it is being dequeued if its previous value is 1.

If *b* is already enqueued, the producer simply sets *multiplicity* of *b* to the number of new tasks. Since *multiplicity* of *b* was previously nonpositive, this last step enables consumers traversing the queue to obtain tasks from *b*. Reusing a PCB that is already on the queue takes five shared-data updates, one for every PCB field except *next*.

Otherwise, the producer busy-waits until *state* of *b* is 1, which indicates that the dequeue of *b* is complete (*state* of *b* is not 0 because of the *f&i* operation performed by the producer described above). If *b* is already dequeued, the wait time is 0. Then it initializes *state* of *b* to 3, updates *multiplicity* of *b*, and enqueues *b*, as described in the next section. Once *multiplicity* is set, consumers may acquire tasks, even though the enqueue operation is not complete. However, this concurrency is not harmful, as discussed in the subsection on correctness issues below.

Enqueue PCB

An enqueuer appends a PCB *b* by swapping a local pointer to *b* with *q_tail* using a *f&s* operation. If the fetched value of *q_tail* (*previous*) is nonempty, *previous→next* is set to *b*. If *previous* is `null`, the queue is empty, so *q_head* is set to *b*. The field *next* of *b* has already been set to `null`.

The procedure *enqueue* in the Appendix appends a list of PCBs with head *my_head* and tail *my_tail* to the queue (*q_tail* is set to *my_tail* instead of *b*). This generality is necessary because *enqueue* calls *dequeue* to append a list of misplaced PCBs (see the subsection on dequeuing the PCB above). The cost of appending a list, which is the same as the cost of appending a single PCB, is three shared-memory accesses when the queue is nonempty.

If the queue is empty, the enqueuer decrements *my_head→state* with a *f&d* operation. Because there can be concurrent consumers removing tasks from *b* while it is being enqueued, *b* may already be empty. If *state* of *b* is 2 (*b* is empty queue head), the enqueuer becomes a *dequeuer*.

- *Correctness issues*

Since our queue is nonlocking, a PCB can be involved in multiple operations concurrently. While concurrency can improve performance, it introduces the possibility of erroneous race conditions, in which one processor makes an update to a queue data structure that causes another operation to function incorrectly, perhaps destroying the integrity of the queue. In this subsection, we show that no harmful race conditions can arise from concurrent operations.

Concurrent access to queue data structures is coordinated using the *state*, *multiplicity* and *number_left* fields of PCBs. Below, we give a detailed case-by-case analysis of concurrent queue operations, for all possible pairs of operation types.

Consume, consume: Consumers of a PCB *b* execute distinct iterations in the range $1, \cdots, N$; only one consumer discovers that *b* is empty.

A consumer successfully obtains an iteration by executing a *f&d* operation on *multiplicity* of *b*, which is initially *N*, and fetching a positive value. The consumer

that fetches 0 as its result discovers that b is empty, decrements *state* of b, and becomes its dequeuer (if the fetched value of *state* is 2). There can only be one such consumer. If the fetched value is negative, the PCB is empty, and the consumer is finished with b. Consumers of distinct PCBs do not interfere with each other.

Consume, dequeue: Consumers cannot obtain tasks from a PCB b that is being dequeued, and a consumer cannot become a dequeuer while another processor is dequeuing b.

A dequeuer never updates *multiplicity* of b, which is negative during a dequeue operation since b must be empty. Therefore, any consumer that attempts to obtain a task from b cannot succeed. If b is not being dequeued, it must be an interior PCB. If b becomes empty, no consumer can dequeue it, since a consumer can only decrement *state* of b once for being empty. The second decrement of *state* of b can be performed only when a dequeuer updates *q_head* to point to b. Therefore, b can be removed by the dequeuer only after b becomes the queue head.

Consume, produce: A producer cannot allocate a PCB b when it is nonempty, and a consumer cannot obtain a task from b while a producer is updating it with new task information.

Before reusing b, a producer (the owner of b) waits until *number_left* of b is 0, thereby ensuring that there are no consumers that can obtain a task from the previous use of b. Since *number_left* is decremented to 0 only after *multiplicity* has been decremented to 0, b must be empty. A producer updates all task-related fields of b before setting its *multiplicity*. (Setting *multiplicity* enables consumers to obtain tasks from b.) At this time, b is in a consistent state for task assignment. Consumers and producers of different PCBs do not interfere with each other.

Consume, enqueue: Tasks from a PCB b can be consumed safely while b is being enqueued.

It is possible for a consumer to obtain tasks from b while it is being enqueued, since *multiplicity* of b is set before *enqueue* is called (see *produce* in the Appendix). This is safe, however, since consumers decrement *multiplicity* and *number_left* of b and do not update *next* of b, and the enqueuer sets *next* of b and not the other two fields. Nevertheless, a potential conflict can arise between a concurrent enqueuer and consumer of b if the queue is empty when b is appended and all of the tasks in b have been consumed. As shown in *produce*, *state* of b is set to 3 before a consumer can obtain tasks and before the enqueue operation begins. If b becomes an empty queue head while it is being concurrently enqueued and consumed, *state* of b will be decremented twice: once by the current dequeuer and once by the consumer that obtains the last task. Whichever processor fetches 2 as the previous value of *state* will dequeue b.

Dequeue, dequeue: There can only be one dequeuer at a time.

As argued above, if there are concurrent consumers accessing a PCB b when it becomes an empty queue head, only a single consumer can become its dequeuer (*consume*, *consume*). If a consumer and an enqueuer are concurrently accessing b when it becomes an empty queue head, only one of these processors becomes its dequeuer (*consume*, *enqueue*). There can be no other dequeue operation in progress, since *q_tail* is `null` at the time of the enqueue operation; therefore *q_head* must also be `null`. Finally, a consumer cannot become a dequeuer while another dequeue operation is being performed (*consume*, *dequeue*). Thus, there can only be one dequeue operation at a time.

Dequeue, produce: A PCB b cannot be reused by a producer while it is being dequeued, and a dequeue of b cannot be started while b is being reused in place by a producer.

A producer coordinates with a dequeuer of b by executing an initial *f&i* operation on *state* (see *produce*). If a dequeue operation has begun, the producer busy-waits until *state* is 1, indicating that the dequeuer has finished, before making b available again for task assignment by setting its *multiplicity*. Conversely, if b is enqueued and a dequeue operation has not begun, the initial *f&i* operation, which flags b as nonempty, reserves b for the producer. A dequeue operation cannot start before the producer sets *multiplicity*.

Dequeue, enqueue: Enqueuers and dequeuers coordinate so that an empty queue is initialized correctly and no misplaced PCBs are lost.

Our empty-queue solution is explained in detail above in the subsection on dequeuing (see also [25]).

Produce, produce: There is only one producer of a PCB b at a time.

The only producer of b is its owner, which waits for *number_left* of b to be decremented to 0 before it is reused. There are no conflicts among producers of different PCBs.

Produce, enqueue: A PCB b cannot be produced and enqueued at the same time.

The only producer and enqueuer of b is its owner, and an enqueue operation follows a produce operation. There are no conflicts among producers and enqueuers of different PCBs.

Enqueue, enqueue: Enqueue operations can safely be performed concurrently.

The only enqueuer of a PCB b is its owner, which enqueues it by a *f&s* operation to swap b with *q_tail*. Because *f&s* is atomic, all concurrent enqueuers fetch different values of *q_tail*. Thus, they do not conflict with each other.

- *Nonblocking and no-wait queues*

Although our queue algorithm is nonlocking, it is possible for an extremely slow processor to delay other processors for a long time. This arises from processors having to busy-wait. Algorithms, such as [21] and [23], that do not require busy-waiting are called *nonblocking* [21]. The three instances of busy-waiting in our algorithm are

1. When a producer waits for a PCB that is currently being dequeued to be fully dequeued (*state* = 1), so that it can be re-enqueued.
2. When a dequeuer waits for *old_head→next* to be set (misplaced PCBs are currently being enqueued), so that it can enqueue the list of misplaced PCBs (see the subsection on dequeuing the PCB).
3. When an owner processor waits for all helper processors to decrement *number_left* before executing the child task of a parallel construct.

Eliminating the first source of busy-waiting is straightforward: Rather than the owner of a PCB b busy-waiting until b is dequeued and subsequently enqueuing b, the dequeuer of b can detect that a reuse attempt has been made (by testing the value returned by the final *f&i* of *state*) and re-enqueue b. The second source of busy-waiting is more problematic, and we know of no nonblocking solution. The third source of busy-waiting is an unavoidable consequence of our scheduling policy. As described in Section 7, its cost is bounded by the execution time of a task, which can be reduced through use of a chunking strategy such as *factoring* to balance workloads (see Section 8).

Being *wait-free* is another desirable property of a scheduling queue [26]. An algorithm is wait-free if the number of instructions per operation is bounded. Hence, a wait-free algorithm is nonblocking, but not all nonblocking algorithms are wait-free. Wait-free algorithms are useful for real-time applications in which each task must start executing within a specified amount of time. Our scheduling-queue operations are not wait-free, but the queue does have the weaker property that at least one iteration of a parallel construct is started within a fixed number of instructions after it is enqueued: An enqueuer becomes a dequeuer only if the PCB that it enqueues is completely consumed by other processors. Thus, either the owner never calls the procedure *enqueue*, the owner returns from *enqueue* immediately after appending its PCB, or a helper processor starts executing the parallel construct before the second *if* statement in *enqueue* is reached.

7. Experimental results

Our run-time scheme has been implemented on the RP3 [11]. We present experimental results justifying two significant design choices: our scheduling policy, which makes it possible to use local memory rather than a shared-memory storage pool for stack-frame allocation, and our singly-linked-list queue algorithm which permits empty interior PCBs. For the sample programs used in the experiment, both of these design choices improved performance. Before describing the experiment, we give a brief overview of the RP3 system.

- *Architecture of the RP3*

The RP3 is an experimental machine whose 64 processors are interconnected by an omega network [11, 14]. Each processor has a cache and local memory. All nonlocal memory is accessible over the network. This hierarchy of cache, local, and remote memories has an access time ratio of 1:12:20. To reduce network contention, there is an (optional) address-translation scheme, called *interleaving*, wherein shared data with consecutive addresses are stored in consecutive memories rather than sequentially in the same memory (hence, accessing different pieces of data, such as individual elements of an array, does not cause contention at any one processor memory). Interleaved memory can be viewed as global, since delay in accessing interleaved data is, on average, the same for all processors.

The caches are write-through and two-way set-associative. There is no interprocessor cache consistency enforced by hardware, so the software must ensure cache consistency. In our run-time implementation, program data are cacheable, and a cache-invalidate instruction is executed by each processor when it completes a parallel construct. This is safe if there are no data dependencies among parallel tasks [27] (which PTRAN guarantees).

The RP3 operating system is a version of Mach, modified to allow threads to be bound to processors, and applications to run in single-user mode [14]. These extensions reduce timing differences from run to run of the same program.

- *Experiment*

Our experiment compared

- Placing processor stacks used for task frames in local memory with placing the stacks in global (interleaved) memory.
- Using our nonlocking singly-linked-list algorithm with using a locking doubly-linked-list algorithm that permits interior-PCB removals.

The doubly-linked-list implementation is highly optimized. Nonempty PCBs can be accessed concurrently, since consumers do not dequeue them. Locking is implemented using *fetch&add* operations as described in [9].

Two test programs were used in the experiment: a program that performs six-integer matrix multiplications and a loop nest taken from a Gauss–Jordan program for (back) solving a system of linear equations. A basic matrix multiplication nested-parallel loop has the form

```
PARALLEL DO I = 1,N
   PARALLEL DO J = 1,N
      DO K = 1,N
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
      END DO
   END DO
END DO
```

Our test program consists of the six different versions of the above nested loop created by reordering the three loops in all possible combinations. The sequential DO K loop is the outermost loop in two of the versions, it is the middle loop in two versions, and it is the innermost loop in two versions. Therefore, this program provides a variety of nesting patterns and loop granularities for comparing run-time overheads.

Our Gauss–Jordan test program is the loop nest

```
DO I = 1,N
   PARALLEL DO J = 1,N
      IF I .NE. J THEN
         PARALLEL DO K = I+1,N+1
            A(J,K) = A(J,K) - (A(J,I) * A(I,K)) / A(I,I)
         END DO
      END IF
   END DO
END DO
```

The granularity of parallelism in this loop nest is both very fine and independent of the problem size N, since a parallel task consists of a single iteration of the innermost loop calculating A(J,K).

Measurements for both test programs were made for all eight combinations of the following parameters:

- Matrices of dimension 150 × 150 or 300 × 300.
- Singly or doubly linked lists (i.e., nonlocking or locking).
- Task frames in local or global memory.

For the results described, we ran each program on 4–62 processors and repeated each run four times; we report the average running time. In most of the cases, the variation in running time (coefficient of variation) was less than 1%. We do not report running times for which the coefficient of variation was greater than 5%, which occurred in a few of the doubly-linked-list runs.

- *Results*

The minimum execution time for the 150 × 150 matrix multiplication program was obtained with 48 processors. This is illustrated in **Figure 7**. (Rather than studying execution time data, however, we have found the data presented in the format of Figures 8 to 11 to be more useful.) As additional processors were added, the program ran more slowly, for two reasons: The additional queue overhead exceeded the additional processing capability, and idle processors spin-waiting for the queue head increased memory contention. With 300 × 300 matrices, the execution time for the matrix multiplication program decreased as the number of processors increased, up to 62 processors. For the doubly-linked-list measurements, the coefficient of variation was greater than 10% for the 150 × 150 Gauss–Jordan program with more than 32 processors (so the results are not reported). The large variance occurred because the computation was dominated by competition for the queue lock. With 300 × 300 matrices, the coefficient of variance remained 5%, and the execution times continued decreasing up to 62 processors.

Total execution-time costs (execution time multiplied by the number of processors) are shown for the matrix multiplication and the Gauss–Jordan programs in **Figures 8** and **9**, respectively. Note that the horizontal scales of Figures 7 to 11 are not uniform. In the ideal situation, the cost remains constant as the number of processors changes. In all cases, the *sl* scheduler (singly linked, local) had the lowest cost. In addition, the cost of the *sl* scheduler rose relatively slowly as the number of processors increased, for most of the experiments.

When the tasks were fine-grained or there was only a small amount of work for each processor, the *sg* scheduler (singly linked, global) had the next lowest cost; for example, the Gauss–Jordan program (Figure 9) and the 150 × 150 matrix multiplication program when the number of processors was greater than 32 [Figure 8(a)]. Using a better queue algorithm was more important than using local memory, since the ratio of task-stack-frame accesses to queue operations was relatively small. When the tasks were more coarse-grained and there was more work available, the converse held. In Figure 8(b), the *dl* scheduler (doubly linked, local) consistently had the next lowest cost.

Both the *sg* scheduler and the *sl* scheduler showed decreasing execution times as the number of processors increased, up to 62 processors for the 150 × 150 Gauss–Jordan program, while execution times of both doubly-linked-list schedulers were highly variable and in general did not decrease after 32 processors. This indicates that the nonlocking, singly linked list is a more scalable queue algorithm than a locking list.

The largest costs were obtained with the *dg* scheduler (doubly linked, global). The percentage performance

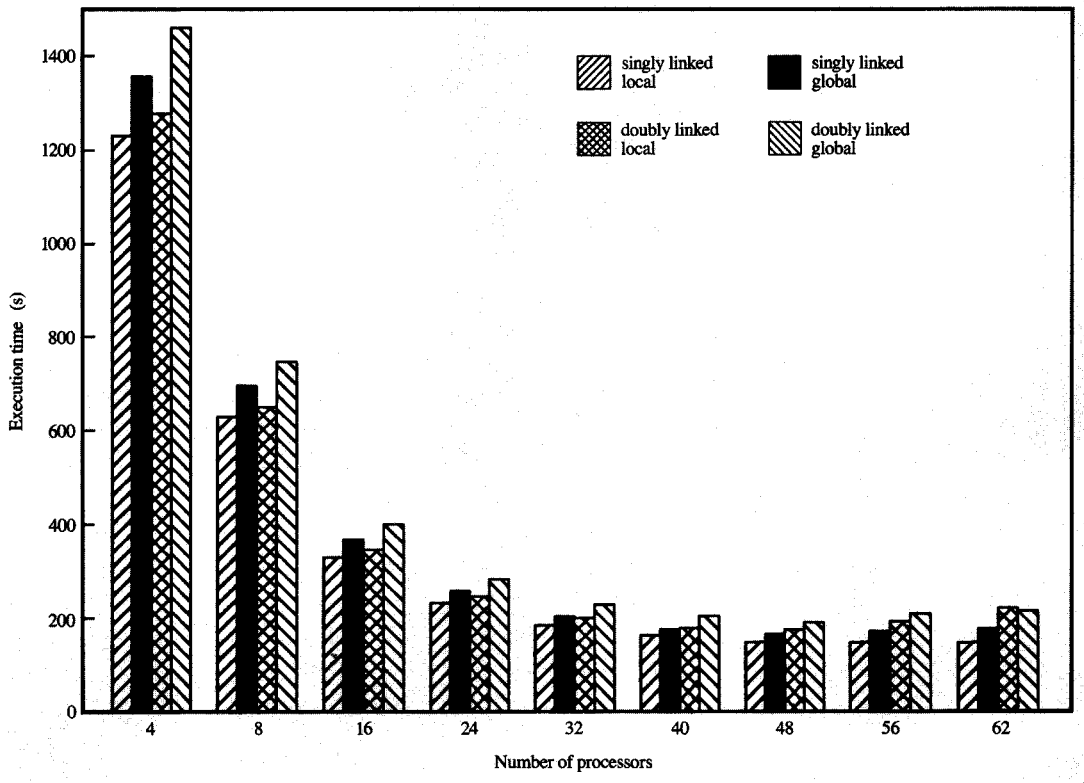


Figure 7

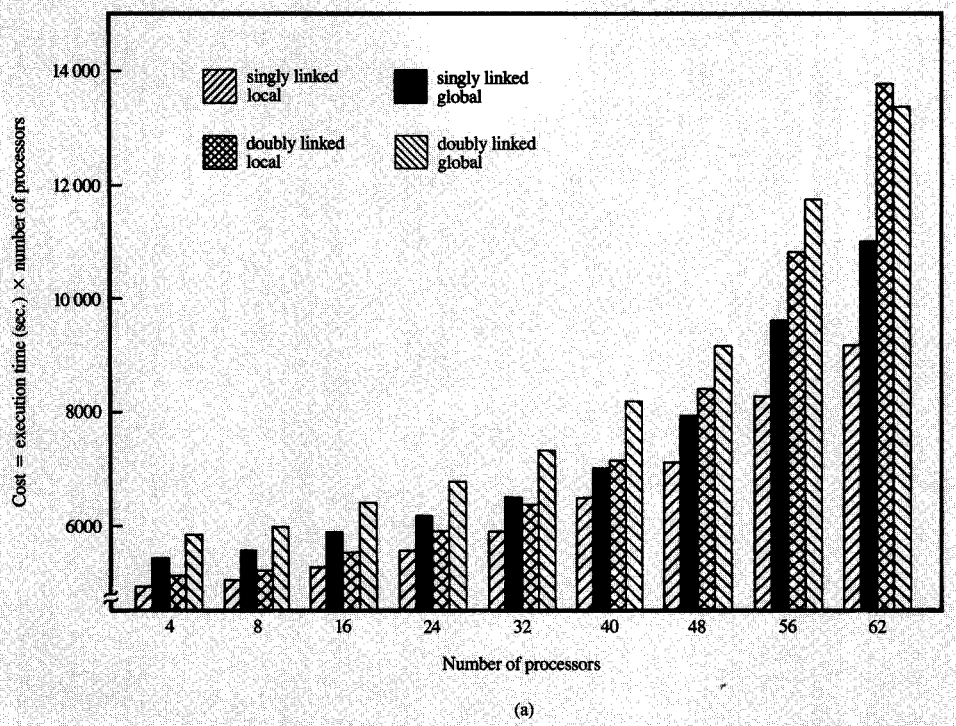
Execution times for the matrix multiplication program with 150 × 150 matrices.

improvement of the other schedulers over the *dg* scheduler is shown in **Figures 10** and **11** for the matrix multiplication program and Gauss–Jordan program, respectively. The *dl* scheduler improved execution times over the *dg* scheduler by −2% for 150 × 150 matrix multiplication on 62 processors [Figure 10(a)] to 27% for 300 × 300 matrix multiplication on 16 processors [Figure 10(b)]. There was only one case [Figure 10(a), with 62 processors] in which the *dl* scheduler performed worse than the *dg* scheduler. This kind of unexpected behavior is usually accounted for by subtle interactions between the application and the operating system. The *sg* scheduler improved execution times by 2% for 300 × 300 matrix multiplication on 56 processors [Figure 10(b)] to 31% for the 150 × 150 Gauss–Jordan problem on 32 processors [Figure 11(a)] over the *dg* scheduler. The *sl* scheduler, which included both optimizations, improved execution times by 15% [Figure 10(a)] to 37% [Figure 11(a)] over the worst case.

This experiment shows that there is benefit in using local memory for task stack frames, especially when task granularity is large. Therefore, our scheduling policy, which permits local-memory stack-frame allocation, is well-suited for nonuniform memory access (NUMA) architectures. Furthermore, even though the scheduling policy requires a queue algorithm that accommodates empty interior PCBs, our queue algorithm is highly efficient, even when tasks are very fine-grained.

- *Other costs*

All of the schedulers in the experiment were implemented with individual processor stacks; the experiment measured the difference between placing the stacks in local memory and placing them in global memory. However, without our scheduling policy, individual processor stacks are not possible, and a more expensive shared-memory storage-allocation algorithm must be used for the cactus stack. If our *sg* and *dg* schedulers paid the full cost of a general cactus-stack implementation, the performance improvements of the local-memory algorithm would be larger. We analyze the cost of general cactus-stack



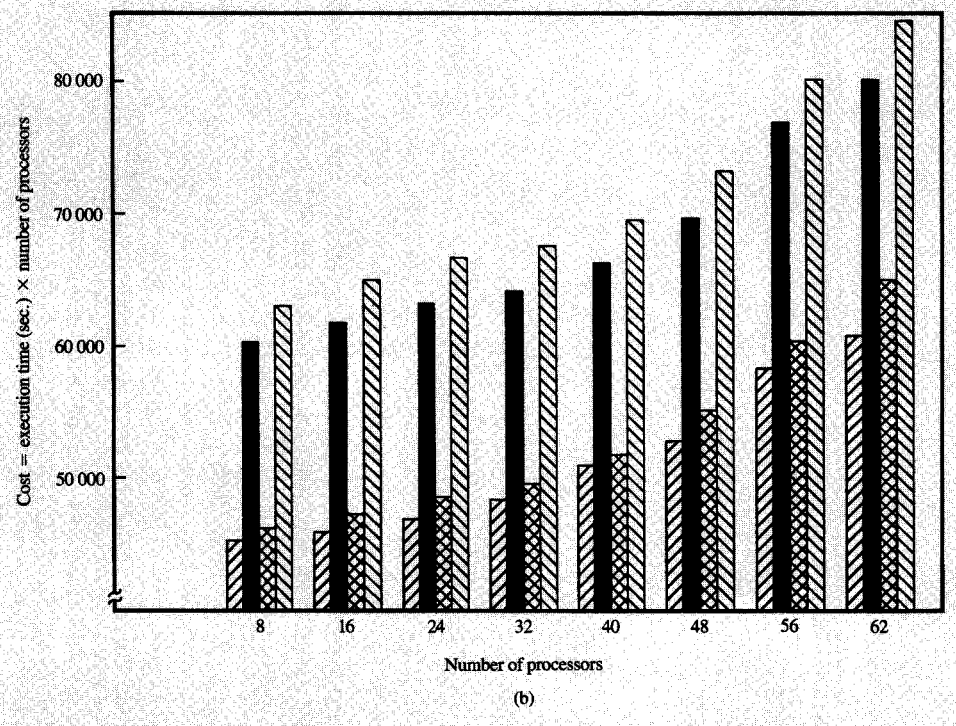
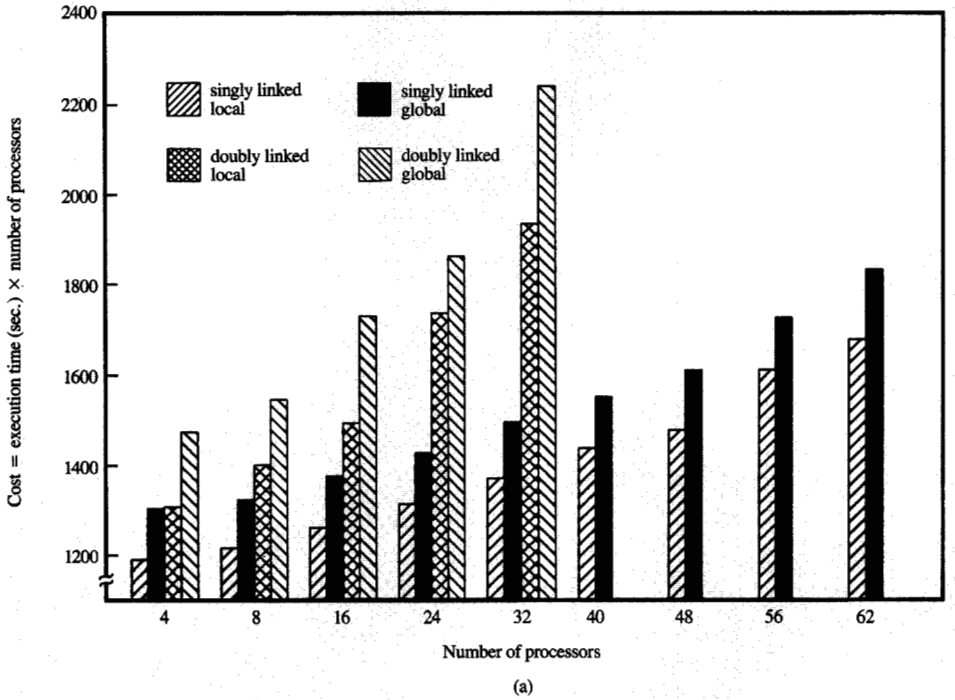
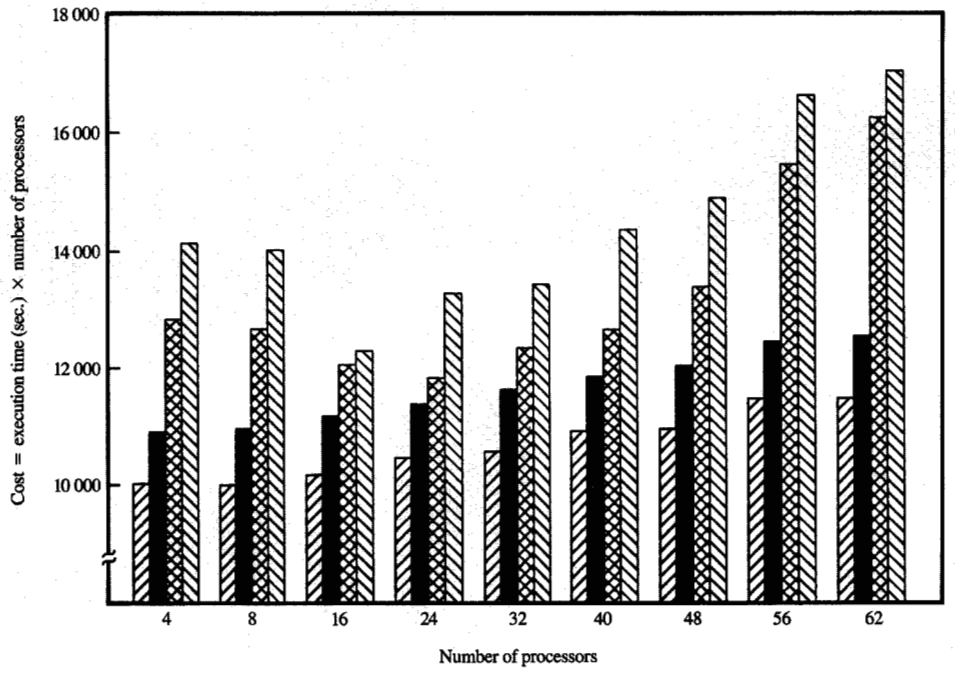


Figure 8

Costs for the matrix multiplication program with (a) 150×150 matrices and (b) 300×300 matrices.



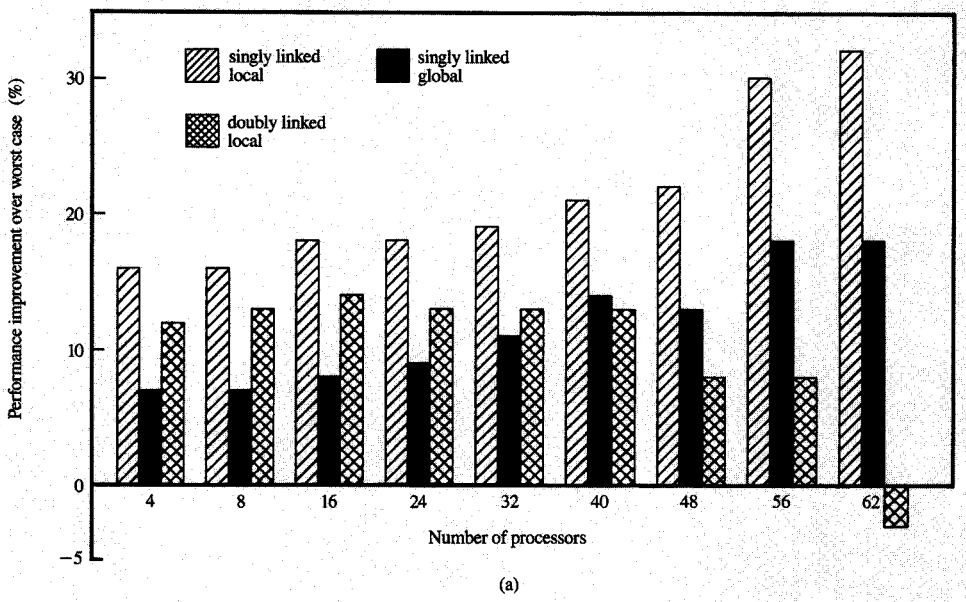
(a)



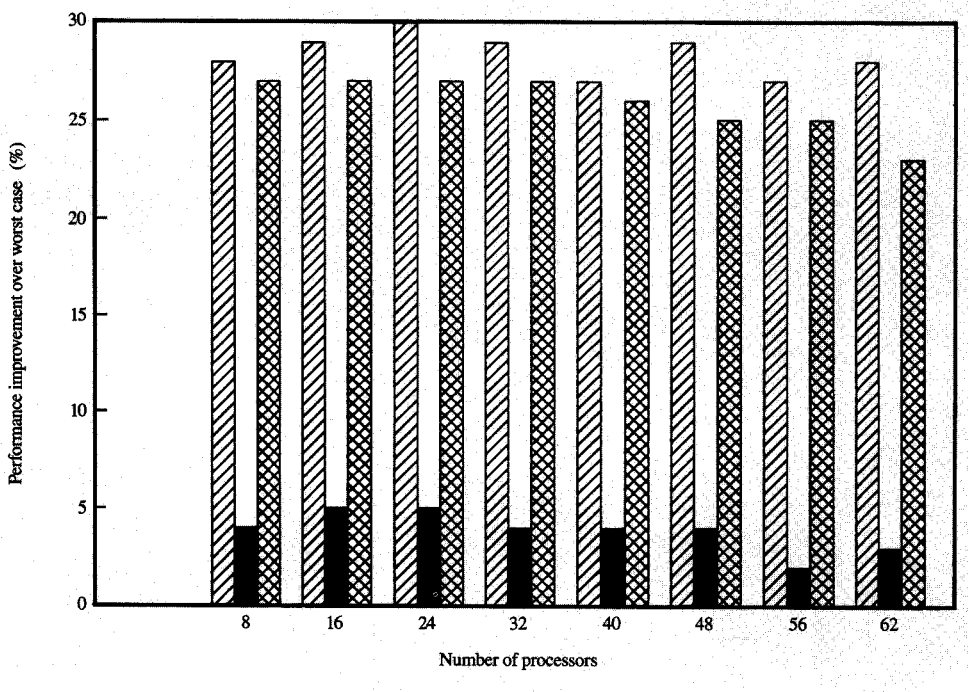
(b)

Figure 9

Costs for the Gauss–Jordan program with (a) 150×150 matrices and (b) 300×300 matrices.



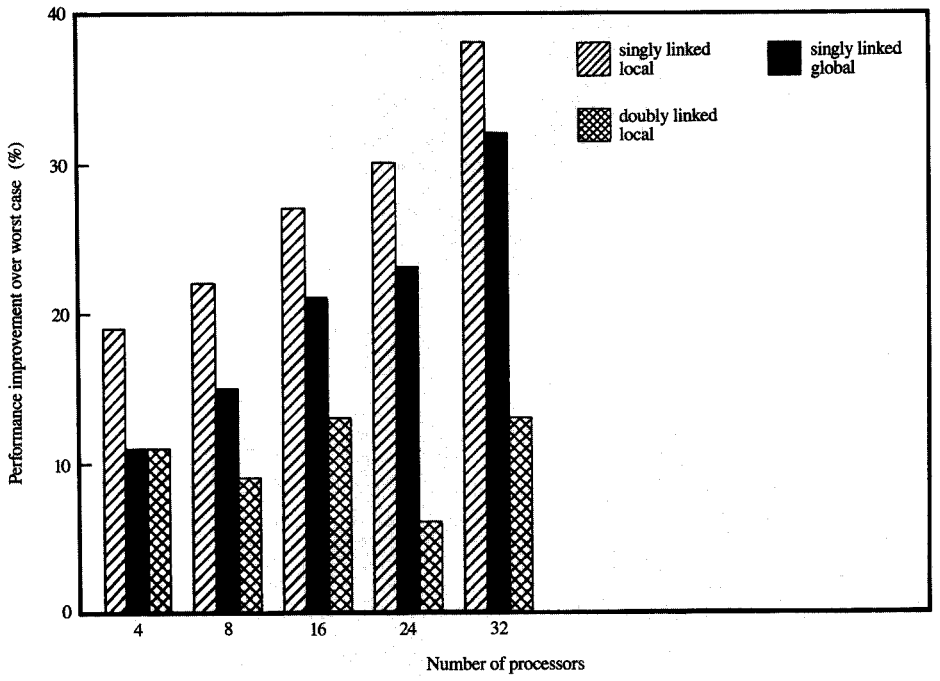
(a)



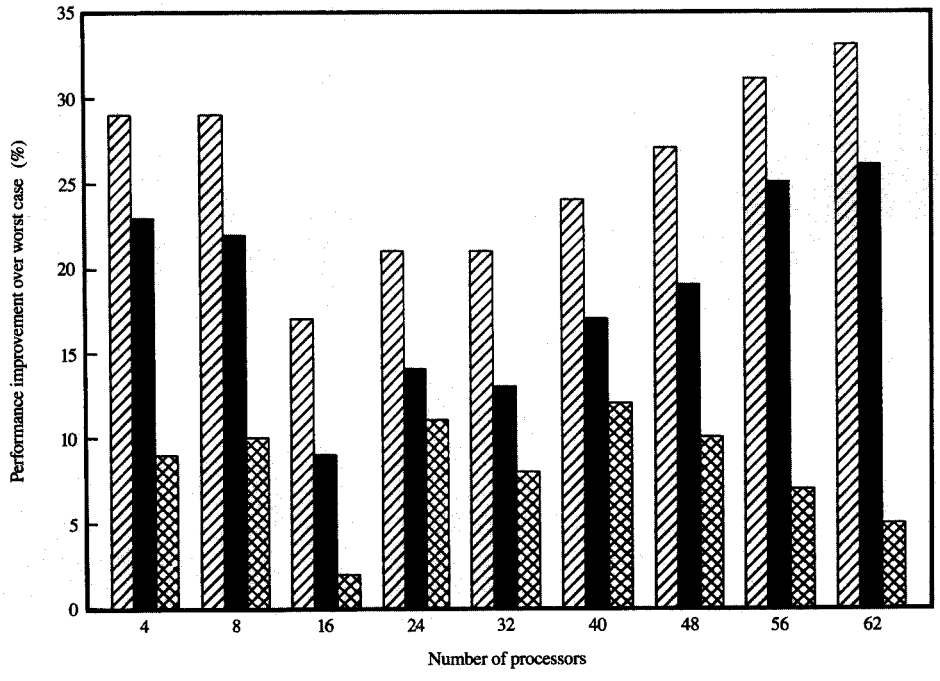
(b)

Figure 10

Performance improvement over worst case for the matrix multiplication program with (a) 150×150 matrices and (b) 300×300 matrices.



(a)



(b)

Figure 11

Performance improvement over worst case for the Gauss–Jordan program with (a) 150×150 matrices and (b) 300×300 matrices.

allocation for a parallel construct *PC* executed by *P* processors as follows. When *PC* is executed, *P* stack frames must be allocated. The best concurrent-access storage-allocation algorithm we are aware of (a version of the buddy-system algorithm) has a worst-case running time per request proportional to

$$\log(arena\_size) - \log(request\_size),$$

where *arena_size* is the size of the shared-memory storage pool, and *request_size* is the size of the storage requested [28]. In our case, *request_size* is *frame_size*, the size of a stack frame, and *arena_size* must be at least $P \times max\_nesting\_depth \times frame\_size$. Thus, the cumulative worst-case cost of the central allocation of all *P* stack frames is at least proportional to

$$\begin{aligned} P \times [&\log(P \times max\_nesting\_depth \times frame\_size) \\ &- \log(frame\_size)] \\ &= P \times \log(P \times max\_nesting\_depth). \end{aligned} \quad (1)$$

Thus, the cost of central allocation increases with *P*.

On the other hand, our scheduling policy has an additional cost: the idle time consumed by the processor that owns *PC* as it busy-waits for helper processors to finish their tasks (see Section 4). Since the owner itself executes tasks until the PCB is empty, this cost is bounded by the maximum running time of a task, which we denote as *max*. If *max* is small, the busy-waiting time is not significant. If *max* is large, it is possible to improve load balancing as follows. Instead of busy-waiting until *PC* is finished, the owner temporarily becomes a helper by obtaining a task from the queue. After the owner executes this task and *PC* is finished, the owner executes the child of *PC*.

8. Extensions

In this section, we propose several strategies that address performance issues raised in previous sections, such as even processor distribution and load balancing. We also discuss extensions that permit tasks that do not run until completion to accommodate explicit synchronization primitives (see Section 2). Since the overhead of complex schemes can outweigh their benefit, especially for fine-granularity parallelism [6], the extensions described here are fairly simple.

- *Reducing scheduling overhead*

A common technique to reduce the overhead of scheduling parallel loops is to assign iterations to processors in *chunks* rather than individually, so that each task consists of a set of consecutive iterations. With fixed-sized chunks of *K* iterations, the maximum task-execution time *max* (the expected amount of time an owner may have to busy-wait until all helpers are finished) is increased by a factor of *K*. However, a variable-sized chunking scheme called *factoring* has been proposed [17], in which the expected idle time is the maximum execution time of a single iteration, instead of a single task (chunk). By allocating tasks in decreasing-sized chunks, factoring reduces scheduling overhead without impairing load balancing. Factoring has been shown to outperform other chunking methods on loops with a wide range of iteration characteristics [18] by allocating tasks in decreasing-sized chunks. Because the scheduling overheads are lower when the tasks in each PCB are scheduled using factoring, more fine-grained nested parallelism can be exploited with factoring than with other chunking methods.

- *Even processor distribution*

With our scheduling scheme, processors searching the queue obtain work from the first nonempty PCB. However, if different processors look for work in different PCBs on the queue, there will in general be fewer processors assigned to each parallel construct, and the processor distribution will be more even (see Section 4). To address this problem, a scheme is proposed in [5] in which processors are allowed to scan the entire queue before deciding which task to obtain. Alternatively, we could add a third counter to PCBs that limits the number of processors that execute a parallel construct. This counter would be used instead of *multiplicity* to determine when to update *q_head*. The maximum number of processors desired for each construct could be specified by either the compiler or the run-time system.

- *Adaptive fork*

There is no benefit in performing a fork operation unless a processor is available or becomes available to help with the work. To reduce unnecessary fork operations, we can keep a count of the number of available processors and perform a fork operation only when the count is positive. Experimentation with such adaptive forks has shown generally good performance improvements, as much as 20% for the matrix multiplication program in Section 7. However, the execution time for different runs of the same version of the program varied by 10–20% with adaptive forking. We therefore disabled it in our experiment, in order to obtain repeatable measurements.

- *Block/unblock operations*

Explicit synchronization, such as posting and awaiting events, can lead to deadlock unless it is possible for the run-time system to block tasks (see Section 2). For example, if all scheduled tasks are waiting for unscheduled tasks, some scheduled task(s) must be suspended. When tasks may be blocked, a more expensive cactus-stack implementation is required, because tasks are not necessarily resumed in LIFO order. Alternatively, the

blocking and unblocking of tasks can be implemented in the run-time system by using a shared work queue, P local work queues, and P storage pools. Initially, tasks are allocated from the central work queue. Once a processor p begins a task, the task may not be blocked and reassigned to another processor, but it may be blocked and resumed on p. The stack frame for the task is allocated from the local-memory storage pool of p. If the task is blocked, it is placed on the local work queue of p. Since a task cannot be resumed on another processor, there is a trade-off of load balance and locality.

9. Conclusion

We have designed, implemented, and assessed a run-time supervisor that schedules nested parallel loops on multiprocessors that have both local and shared memory. A multiprocessor scheduler must optimally trade off processor load imbalances with overhead. For example, exploiting nested parallel loops can improve load balancing, but the run-time realization is more complex than for simple loops. In general, achieving balanced work loads requires centralized resource allocation, and the increased overhead arises either from the loss of locality or from contention. By pairing local stacks with a global work queue, our system attempts to minimize the cumulative performance loss from load imbalances and overhead.

With our scheduling policy, the parent of a parallel loop enqueues it, executes iterations until it is empty, and then waits for any helper processors to finish with their iterations. The policy permits private variables for the iterations to be allocated locally from individual processor stacks. Locality is further exploited by reusing stack frames: Once a processor executes an iteration, rather than searching the global queue for new work, it continues executing iterations from the same loop. Iterations from the same loop can use the same stack frame. Thus, the overhead of executing subsequent iterations is reduced.

The global work queue is implemented as a singly linked list, with *fetch&increment*, *fetch&decrement*, and *fetch&store* operations used for synchronization. The queue mechanism uses an "optimistic algorithm" in that it is highly efficient for the usual case, and operations are not "locked out" when the queue is in an inconsistent state. Inconsistencies are detected and subsequently corrected rather than prevented. As an example of the efficiency of queue operations, we note that when the queue is nonempty, a processor can execute a fork operation and begin executing a task from a parallel construct after fewer than six shared-memory accesses.

A consequence of our scheduling policy is that the queue may have holes—that is, interior entries whose loop iterations have all been executed. Our queue accommodates such empty interior entries by leaving them as place holders and allowing them to be reused in place. A state variable is included in an entry to coordinate its reuse, enqueue, and dequeue. By using the state variable to delay the reuse of a PCB while it is being dequeued, our algorithm permits a high degree of concurrent queue operations.

To assess our run-time system, we compared, on the RP3, our scheme to a more traditional scheme that uses global-memory stack allocation and a doubly linked work queue (which allows PCBs to actually be removed). Two programs were tested, one with coarse-grained iterations and the other with fine-grained iterations. The number of processors was varied from 4 to 62 to test the scalability of the schemes. The experiment revealed that the local-stack-allocation optimization was particularly effective (up to 27% performance improvement) when the granularity was coarse, since there were more opportunities for reusing data, and that the singly-linked-list optimization was particularly effective (up to 31% performance improvement) when the granularity was fine, since there was greater contention for the scheduling queue. These effects became more pronounced as the number of processors was increased. Measured together, the two optimizations improved performance by as much as 37%.

While our scheme works best on machines with local memory, it can be implemented on machines without local memory by placing the individual processor stacks in global memory. The allocation of stack frames is still contention-free and, hence, less expensive than using a centralized storage pool for allocation.

Appendix: Code for scheduling queue

```
/* Look for work on the queue */
consume_q() {

   pcb_pointer my_pcb;
   integer my_task;

   while true {
      /* Traverse queue until a nonempty PCB is found   */
      my_pcb = q_head;
      while (my_pcb ≠ null) {
         if (my_pcb→multiplicity ≥ 0) {
            my_task = f&d(my_pcb→multiplicity);
            if (my_task > 0)    /* Found a task       */
               return [my_pcb, my_task];
            if (my_task = 0) {
               /* my_pcb is empty, so decrement state */
               if (f&d(my_pcb→state) = 2) {
                  dequeue(my_pcb);
                  my_pcb = q_head;
                  continue;
               }
            }
         }
         my_pcb = my_pcb→next;
      }
   }
}
```

```
/* Look for work in PCB my_pcb */
consume_pcb(pcb_pointer my_pcb) {

   integer my_task, previous_state;

   my_task = f&d(my_pcb→multiplicity);
   if (my_task ≠ 0) {
       f&d(my_pcb→number_left);

       /* Test whether a task was obtained */
       if (my_task > 0) return my_task;
       return failure;
   }

   /* my_pcb is empty, so decrement state */
   previous_state = f&d(my_pcb→state);
   f&d(my_pcb→number_left);
   if (previous_state = 2)
      dequeue(my_pcb);

   return failure;
}
```

```
/* Remove empty entries at the head of the queue */
dequeue(pcb_pointer head) {
   /* head is a private copy of q_head  */

   pcb_pointer old_head, old_tail, first_missed;

   /* Advance q_head until the queue is empty or
      a nonempty PCB is found */
   while true {
      old_head = head;
      head = old_head→next;
      q_head = head;

      if (head = null) break;

      f&d(old_head→state);  /* Removal complete */

      /* Update and test state of the new queue head */
      if (f&d(head→state) ≠ 2) return;
   }
   /* The queue is empty. Set q_tail to null so that q_tail
      and q_head are consistent */

   old_tail = f&s(q_tail, null);

   /* Take care of misplaced PCBs */
   if (old_tail = old_head) {
      /* No misplaced PCBs   */
      f&d(old_head→state);  /* Removal complete */
      return;
   }

   /* Otherwise, must append misplaced PCBs to the queue */

   /* Busy_wait for possible enqueue in progress to complete */
   while (old_head→next = null) {}

   first_missed = old_head→next;
   f&d(old_head→state);        /* Removal complete */

   enqueue(first_missed, old_tail);
   return;
}
```

```
/* Initialize state, multiplicity, and next of PCB */
produce(pcb_pointer my_pcb, integer multiplicity)  {

   /* my_pcb is allocated, so increment state  */
   if (f&i(my_pcb→state) = 2) {
      /* my_pcb is already enqueued */
      my_pcb→multiplicity = multiplicity;
      return;
   }

   /* Wait for dequeue in progress to complete */
   while (my_pcb→state ≠ 1) {}

   /* Enqueue PCB */
   my_pcb→state = 3;
   my_pcb→next = null;
   my_pcb→multiplicity = multiplicity;
   enqueue(my_pcb, my_pcb);
   return;
}
```

```
/* Enqueue a list of  PCBs */
enqueue(pcb_pointer my_head, pcb_pointer my_tail)  {

   pcb_pointer previous;

   previous = f&s(q_tail, my_tail);
   if (previous ≠ null) {
      previous→next = my_head;
      return;
   }
   /* Queue is empty */
   q_head = my_head;
   if (f&d(my_head→state) = 2)
      /* Queue head is empty */
      dequeue(my_head);
   return;
}
```


Acknowledgments

We thank all the members of the PTRAN and RP3 groups for making their systems available for experimentation, and Michael Burke and Frances Allen for their support of this work. We also thank Janice Stone, James Lipkis, and Eric Freudenthal for many helpful discussions, the referees for their useful comments, and Christina Meyerson for producing our figures.

References

1. L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon, "IBM Parallel FORTRAN," *IBM Syst. J.* **27**, No. 4, 416–435 (1988).
2. C. D. Polychronopoulos, "Loop Coalescing: A Compiler Transformation for Parallel Machines," *Proceedings of the IEEE International Conference on Parallel Processing*, August 1987, pp. 235–242.
3. Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi-Zhu, "Dynamic Processor Self-Scheduling for General Parallel Nested Loops," *IEEE Trans. Computers* **39**, 919–929 (July 1990).
4. Michael Weiss, C. R. Morgan, P. Belmont, and Zhixi Fang, "Dynamic Scheduling and Memory Management for Parallel Programs," *Proceedings of the IEEE International*

Conference on Parallel Processing, August 1988, pp. 161–165.
5. David Bernstein, "PREFACE-2, Supporting Nested Parallelism in Fortran," *Research Report RC-14160*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1988.
6. Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *Technical Report 88-09-04*, University of Washington, Seattle, September 1988.
7. James M. Wilson, "Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add," Ph.D. Thesis, New York University, New York, 1988.
8. Susan Flynn Hummel, "SMARTS—Shared-Memory Multiprocessor Ada Run Time Supervisor," Ph.D. Thesis, New York University, New York, 1988.
9. A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Trans. Program. Lang. & Syst.* **5**, No. 2, 164–189 (1983).
10. *IBM System/370 Principles of Operation*, Order No. GA22-7000; available through IBM branch offices.
11. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the IEEE International Conference on Parallel Processing*, August 1985, pp. 764–771.
12. *Inside the Butterfly GP1000*, BBN Systems and Technologies Corporation, 10 Moulton St., Cambridge, MA 02238, June 1985.
13. Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *J. Parallel & Distr. Computing* **5**, No. 5, 617–640 (1988).
14. R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg, "Operating System Support for Parallel Programming on RP3," *IBM J. Res. Develop.* **35**, No. 5/6, 617–634 (1991, this issue).
15. Clyde Kruskal and Alan Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Trans. Software Eng.* **SE-11**, No. 10, 1001–1016 (October 1985).
16. David Kuck and Constantine Polychronopoulos, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," *IEEE Trans. Computers* **C-36**, 1425–1439 (December 1987).
17. L. E. Flynn and S. Flynn Hummel, "Scheduling Variable-Length Parallel Subtasks," *Research Report RC-15492*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1990.
18. S. Flynn Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," *Proceedings of Supercomputing 91*, ACM/IEEE, 1991, pp. 610–619. (Also to appear in *Commun. ACM*, August 1992.)
19. Peiyi Tang, Pen-Chung Yew, Zhixi Fang, and Chuan-Qi-Zhu, "Deadlock Prevention in Processor Self-Scheduling for Parallel Nested Loops," *Proceedings of the IEEE International Conference on Parallel Processing*, August 1987, pp. 11–19.
20. E. A. Hauk and B. A. Dent, "Burroughs B6500/B7500 Stack Mechanism," *AFIPS Proc. Spring Joint Conference*, 1968, pp. 245–251.
21. M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. & Syst.* **12**, No. 3, 463–492 (July 1990).
22. J. M. Stone, "A Simple and Correct Shared-Queue Algorithm Using Compare-and-Swap," *Proceedings of Supercomputing '90*, ACM/IEEE, 1990, pp. 495–504.
23. J. M. Stone, "A Nonblocking Compare-and-Swap Algorithm for a Shared Circular Queue," *Proceedings of the IMACS/IFAC Symposium on Parallel and Distributed Computing in Engineering Systems*, Corfu, Greece, June 1991.
24. J. M. Mellor-Crummey, "Concurrent Queues: Practical Fetch-and-Φ Algorithms," *Technical Report 229*, University of Rochester, Rochester, NY, 1987.
25. J. M. Mellor-Crummey and M. L. Scott, "Synchronization Without Contention," *Proceedings of Architectural Support for Programming Languages and Operating Systems*, ACM, April 1991, pp. 269–278.
26. M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures," *Proceedings of Principles and Practice of Parallel Programming*, ACM, June 1990, pp. 197–205.
27. Alexander Veidenbaum, "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proceedings of the IEEE International Conference on Parallel Processing*, August 1986, pp. 1029–1036.
28. E. Freudenthal, "A Short Note on Parallel Power-of-Two Buddy Allocation for Shared Memory MIMD Machines," *Ultracomputer System Software Note 66*, New York University, New York, February 1992.

Received October 24, 1990; accepted for publication January 29, 1991

Susan Flynn Hummel *School of Computer Science and Electrical Engineering, Polytechnic University, 6 Metrotech Center, Brooklyn, New York 11201.* Dr. Flynn Hummel received a B.A. in mathematics (minor in computer science) from McGill University in 1980 and a Ph.D. in computer science from New York University in 1988. She currently has a visiting position at the Ecole Nationale Supérieure des Mines de Paris, while on leave from her position as Assistant Professor of Computer Science at Polytechnic University in New York. She has previously held visiting positions at the IBM Thomas J. Watson Research Center in New York and the Vrije Universiteit in Amsterdam. Her main research interest is scalable software and hardware systems.

Edith G. Schonberg *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Schonberg received her Ph.D. in computer science in 1980 from New York University. From 1980 to 1983, she conducted research in database management systems at Bell Laboratories, Murray Hill, New Jersey. From 1983 to 1989 she was a Research Scientist at New York University in the Ultracomputer Project, working on parallel operating systems and the debugging of parallel programs. Dr. Schonberg joined IBM in 1989 as a Research Staff Member in the Parallel Software Department at the IBM Thomas J. Watson Research Center. Her current research interests include compiler optimizations, run-time systems, and environments for multiprocessor architectures.