

Synchronizing Operations on Multiple Objects*

Tim Rühl and Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit, Amsterdam, The Netherlands

Abstract. Parallel programming on distributed memory systems is one of the most challenging research areas in parallel computing today. Object-based parallel programming languages are an important class of languages for such systems. Shared objects allow the programmer to deal with data partitioning, communication, and synchronization in a high-level manner. Synchronizing operations on a single object is well understood. Dealing with synchronization on multiple objects distributed over the available processors, however, is still an open issue. In this paper, we will present an abstraction, called *weavers*, that is used to synchronize operations on multiple objects, and show how weavers are applied in a runtime support system for atomic functions on multiple objects.

1 Introduction

Programming parallel computers with distributed memory is one of the most challenging research areas in parallel computing today. The interest in these systems is caused by the fact that such systems are relatively easy to build and cost less than parallel architectures that support shared memory in hardware. In particular, clusters of workstations receive much attention [1].

Writing parallel applications for distributed memory systems, however, is more difficult than for shared memory. To make distributed memory systems more attractive to program, some parallel programming systems provide the illusion of shared memory on top of a distributed memory system [2, 3, 6, 7, 11, 12, 17]. These systems differ in the programming interface they offer. Systems like Shasta and TreadMarks provide the same programming interface as a true shared memory (i.e., a flat address space). Other systems, such as Orca, DiSOM, and CRL, let the programmer specify objects, which contain associated data. The programmer defines operations that can be executed on these objects.

This paper discusses how to synchronize operations on *multiple* objects. This work is based on an extension to Orca, a parallel programming language that provides a shared memory abstraction based on shared objects. Shared objects are instances of an abstract data type, which defines the data that each object contains and the operations that can be invoked on this data. Processes can communicate and synchronize with each other by performing operations on shared objects.

* This research is supported by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O)

For efficiency, the Orca runtime system replicates objects that have a high read/write ratio on all processors that can access this object. Operations that change an object are sent to all processors that have a copy. Each of these processors then performs the operation on its local copy. A totally-ordered broadcast message is used to send this operation. The total ordering guarantees that all processors receive all messages in the same order, and therefore perform the updates in the same order. We have efficient implementations of totally-ordered group communication on a large number of systems [4]. By using a total ordering on the updates sequential consistency is preserved [13]. Read operations on replicated objects are performed on the local copy. Operations on remote single-copy objects are performed using point-to-point communication [9].

The original Orca system supports atomic operations only on a single object. This design decision was made to be able to build an efficient implementation. We have proposed an extension to Orca, called *atomic functions*, that allows the programmer to specify a sequence of statements that is executed atomically on a set of objects [16]. In this paper, we will discuss the problems this extension imposes on synchronization. We propose a generic abstraction, called *weavers*, that we have used to implement runtime support for atomic functions. This abstraction handles all synchronization requirements for performing operations on multiple objects.

In Section 2, we describe the synchronization requirements for performing operations on single Orca objects. Atomic functions and their synchronization problems are presented in Section 3. The weaver abstraction is presented in Section 4. In Section 5, we describe how the runtime support systems for Orca objects and atomic functions are integrated on top of weavers. In Section 7 we discuss related work, and in Section 8 we draw our conclusions.

2 Synchronization on Orca Objects

In parallel programming, two types of synchronization are important. The first type, *mutual exclusion*, guarantees that only a single process can be in a critical section. Other processes that want to access this critical section will be blocked until the first process has left it. Mutual exclusion on shared memory systems is supported by locks. By associating a lock with data and requiring that this lock must be granted before the data may be accessed, mutual exclusion on data access can be enforced.

The other type of synchronization is *condition synchronization*. With condition synchronization, processes depend on each other, i.e., the result of the computation of one process is needed for the computation of the other process to continue. A typical example is a bounded buffer, in which one process puts elements into the buffer and another process gets elements from this buffer. The generating process has to wait if the buffer is full, while the acquiring process has to wait if the buffer is empty. Therefore, each get and put operation can trigger the other process to continue. Condition synchronization on shared memory systems is supported by constructs such as semaphores, event counters, and condition variables.

```

object implementation BoundedBuffer;
  buffer: Bounded buffer data structure
  operation put(e: Element);
  begin guard not full(buffer) do
    Add element e to buffer
  od;
end;

  operation get(e: out Element);
  begin guard not empty(buffer) do
    Remove element from bounded buffer and return it in e
  od;
end;
end;

```

Fig. 1. Bounded buffer example showing condition synchronization in Orca.

In Orca, both types of synchronization are integrated within shared objects. Operations on objects are guaranteed to be executed atomically, i.e., while the operation is executed, no other operations can be executed that would change the result of the first operation. This guarantees mutual exclusion on the object.

Condition synchronization is provided by allowing the programmer to specify *guarded expressions* (see Figure 1). A guarded expression is a block of statements that is only allowed to execute if the associated guard condition is true. An operation can consist of multiple guarded expressions. If some of the guard conditions are true, one is selected and its statements are executed. While none of the guard conditions are true, the operation is *blocked*. In the bounded buffer example, the get operation has a guard condition specifying that an entry may only be removed from the buffer if at least one entry is available. While no entries are available, the get operation is blocked until another process has executed a put operation. The runtime support system guarantees that the get operation will be retried after one or more successful put operations.

3 Atomic Functions on Multiple Objects

The Orca model only supports atomic operations on a single object. This design decision was made to be able to build an efficient implementation [3]. Some applications, however, need higher-level constructs than those provided by the Orca model. Often an Orca application programmer wants to be able to specify atomic execution on *multiple* objects [18]. This section describes *atomic functions* and discusses the problems that arise while synchronizing operations on multiple objects. For conciseness reasons, we will not describe how this extension actually executes operations on multiple objects, but focus only on the synchronization aspects. More information on the execution can be found in [16].

An atomic function is a sequence of statements that is executed atomically on a set of objects. The programmer defines atomic functions like a normal Orca function, except that the keyword **atomic** is prepended to the function header.

```

process Worker(q: shared JobQueue; workers: shared IntObject);
    job: JobType;
begin repeat workers$inc(); # worker becomes active
    while q$GetJob(job) do
        # handle job and possibly add new jobs to queue
    od;
    workers$dec(); # worker becomes passive
until Terminate(q, workers);
end;

atomic function Terminate(q: shared JobQueue; workers: shared IntObject): boolean;
begin guard not q$Empty() do return false; od;
    guard q$Empty() and workers$value() = 0 do
        # q is empty and no worker is active, so terminate
        return true;
    od;
end;

```

Fig. 2. Global termination detection using an atomic function.

The function body consists of a sequence of statements, which may include operations on objects. Objects that are passed as *shared* (i.e., call-by-reference) parameter can be updated by the atomic function. Since Orca does not allow global variables or objects, an atomic function can only access those objects that are passed as parameters. An atomic function may consist of guarded expressions (as in Orca operations), in which the guard conditions may depend on the results of operations on those objects.

Typical usage of an atomic function is to perform operations on objects that are not related to each other most of the time, but sometimes need to be accessed atomically. An example application is termination detection in parallel applications based on a replicated workers paradigm. In such applications, the termination condition states that a process is finished if all processes are idle *and* the work queue is empty. During the execution of the application, processes extract jobs from the work queue, evaluate the job, and possibly add new jobs to the work queue (see Figure 2). When a process detects that the work queue is empty, it decrements the “active worker” counter (a shared object) and calls the atomic function *Terminate*. This atomic function returns true if the process may terminate, and false if the queue is not empty anymore. While neither guard conditions is true, the atomic function is blocked, and so is the invoking process.

Atomic functions must satisfy two synchronization requirements. The first requirement is that the atomic function is a critical section. No other operations that change the result of this atomic function may be executed during this critical section, i.e., no operations may be performed that access any of the objects passed as shared parameter while the atomic function is executing. The runtime support system must guarantee this mutual exclusion.

The second synchronization requirement is that atomic functions must allow condition synchronization. For this synchronization to be useful, the programmer must be able to specify synchronization conditions that depend on the results of

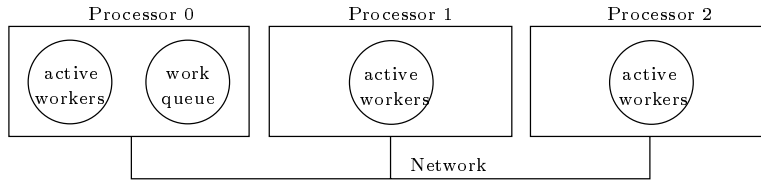


Fig. 3. Distribution of the workers and the queue object on three processors.

operations on objects. This implies that a blocked atomic function needs to be continued whenever one or more of these objects is updated. In the termination detection example, the atomic function *Terminate* must be re-evaluated after either the work queue is updated (since it may contain work) or the active worker counter is updated (to detect termination).

A major problem with the integration of mutual exclusion and condition synchronization is that copies of an object may only be available on a subset of the processors. Mutual exclusion requires that all processors that have a copy of any of the objects accessed in the atomic function are synchronized. When an atomic function blocks, some of these objects have to change before the atomic function can be retried. Therefore, the runtime system must allow operations on these objects while the atomic function is blocked.

The state change of an object, however, is only known on those processors that have a copy of the object, since only those processors perform the write operation. In the termination detection algorithm, the active workers object may be replicated, while the work queue object only is shared on processor 0 (see Figure 3). If the active workers counter is updated, all processors are aware of it, since they all have to update their local copy. After the update, all processors know that the atomic function *Terminate* has to be re-evaluated. If the work queue object is updated, however, only processor 0 performs the update, since it is the only processor with a local copy of the object state. This implies that communication is required to inform the other processors that the atomic function *Terminate* has to be re-evaluated.

In conclusion, implementing mutual exclusion and condition synchronization for atomic functions is much more difficult than for operations on a single object. In the following section, we present an abstraction that handles these synchronization requirements.

4 Weavers

In this section, we will present the *weaver* abstraction, which is used to implement runtime support for operations on multiple objects. A weaver is a *single* function (e.g., the code of an atomic function) that is executed on a *set* of processors. On each of these processors, a thread is started that will execute this weaver function. These weaver threads can communicate with each other to interlace their computations. The synchronization requirements for atomic functions can be mapped to synchronization operations on weavers.

A process can create a weaver by calling the *weaver_create* function, passing the destination processor set as parameter. During the execution of the weaver, the invoking process is blocked. Both input and output parameters can be passed to the weaver, and the results of the weaver invocation will be returned to the process after the weaver is finished.

Weavers are always executed in a total order. To guarantee this, all weavers that will execute threads on multiple processors are created by using a totally-ordered group message. These weavers will receive a global identifier, so that they can be identified by other processors. A weaver that has to run only on a remote processor (i.e., its processor set contains one processor) will be created with a unicast message, since this does not violate the total ordering. If the weaver only has to run locally (i.e., its processor set contains only the local processor), no message is sent. The total ordering on the execution of weavers together with program order (the invoking process is blocked until the weaver is finished) guarantees that weavers can be used to implement sequential consistency [8].

Weavers provide support to map the synchronization requirements for performing operations on multiple objects. Whenever a *weaver_create* message arrives, a new weaver thread is created and enqueued in the *weaver queue*. All processors manage a local copy of this weaver queue, which only contains entries for weavers that a processor has to participate in. Only the *first* runnable weaver in the weaver queue is executed. This guarantees mutual exclusion, since only a single weaver function at a time is allowed to be active on a processor. Other weaver invocations that arrive during the execution of a weaver are enqueued in the weaver queue and not executed yet. Only after all earlier weavers in the queue stopped running will an enqueued weaver function be executed. Since the *weaver_create* messages arrive in a total order, this guarantees that all weaver queues will contain enqueued weavers in the same order.

To implement condition synchronization on weavers, weavers can block by calling the function *weaver_wait*. Whenever a weaver blocks, it remains in the weaver queue, but its state is changed from *running* to *blocked*. Another weaver can execute while earlier weaver functions in the weaver queue are blocked. A weaver can wake up another blocked weaver by calling the *weaver_signal* function, which changes the state of the other weaver from *blocked* to *runnable*.

All threads of a weaver have to make the same calls to *weaver_wait* and *weaver_signal* to ensure the consistency of the weaver queues. Recall that all threads of a weaver execute the same function. However, this is not sufficient to ensure consistency of the queues. To handle condition synchronization on objects, all weaver threads must have the values of all variables used to compute the synchronization condition. If the value of such a variable is the result of an operation on an object, it might be necessary for weaver threads to communicate to send this value to the processors that have no local copy of this object. The weaver runtime system provides efficient communication primitives between weaver threads, which are used during the execution of the atomic function.

Before a weaver blocks, it can add itself to a queue of pending weavers associated with an object. These queues are managed on the processors that have a

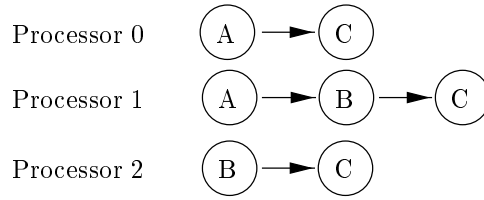


Fig. 4. Weaver queues on three processors containing weavers A, B, and C.

local copy of the object state. Other weavers that update an object look at this pending weavers queue and determine whether these weavers are still blocked. In the example described in Section 3, the weaver that invokes the atomic function *Terminate* and blocks adds itself to the pending weaver queues of the active workers object and the work queue object.

As described in Section 3, waking up a blocked weaver may or may not involve communication. The first situation occurs when a weaver runs only on processors that have a copy of the object on which the weaver is blocked. This situation is the easiest to handle, since all processors can locally change the state of the weaver from blocked to runnable. In the termination detection example, after an update on the active workers object all weavers blocked in atomic function *Terminate* can be signaled without communication.

After a processor changed the state of some of the weavers to runnable, it has to determine which weaver to execute next. Therefore, the weaver queue is traversed starting from the beginning, and the first runnable weaver can continue its execution. Since the weaver function that called *weaver_signal* was executed in a total order, the signals are also handled in total order. The recheck on the weaver queue after a weaver exits therefore guarantees that the total order of the execution of the weavers is preserved. Figure 4 shows an example situation in which weaver A is blocked on processors 0 and 1, weaver B is blocked on processors 1 and 2, and weaver C is running on processors 0, 1, and 2. Now if weaver C wakes up weavers A and B, processor 1 will first run weaver A before running weaver B, since this is the total order in which the weaver functions were created. The threads of weaver A can now communicate with each other, and either exit or block again. Communication to and from the thread of weaver B on processor 1 will be blocked until this thread is running.

Communication is required to signal a blocked weaver if this weaver also runs on processors that do *not* have a copy of the object. In this case, a (totally ordered) *signal message* is required to wake up this blocked weaver. In the termination detection example, after an update on the work queue object communication is required to wake up all weavers blocked in atomic function *Terminate*.

In order to preserve the total order of the execution of the weavers, all updates to the state of the weavers in the weaver queue must be kept consistent. A major difficulty is that the threads of a weaver that perform operations on more than one object cannot always know which other weavers will be signaled. For example, suppose a weaver updates object X and object Y, and object X is only

```

proc weaver_queue_scheduler() ≡
  while runnable weavers do
    entry := first runnable entry
    if entry = signal message then perform signals locally
    else execute weaver in entry
  if marked weavers then send signal message
end

```

Fig. 5. Weaver queue scheduler algorithm.

on processor 0 while object Y is only on processor 1. Since the pending queues associated with an object exist only on processors that have a copy of the state of the object, a processor cannot know which weavers the other processor will signal. This makes it very inefficient to block until all signal messages have arrived, since it would require that all weaver threads communicate with each other before the weaver queue can be rechecked for runnable weavers.

This problem is solved by assigning one of the processors that have a local copy of the object as the *master* processor. When an object is updated, all processors that have a copy of the object perform the local signals (i.e., signals to weavers that belong to the first case). Only the master processor, however, is responsible to send a *signal message* to wake up weavers that belong to the second case. To reduce the number of messages a master processor has to send, it marks these weavers with a *wakeup marker*. When this master processor finds that all weavers in the weaver queue are blocked, it collects the weaver identifiers of the marked weavers, and broadcasts these identifiers in a single signal message.

In this scheme, multiple processors (i.e., all master processors with locally marked weavers) can send signal messages. To preserve the order of the weavers, these signal messages are not executed directly when they are received, but instead are also enqueued in the weaver queue. When no runnable weavers precede the signal message in the weaver queue, all weavers in the signal message that still exist are signaled locally (see Figure 5). The weaver queue is then rechecked, so that the first runnable weaver will be executed. Since the signal broadcast uses the same total ordering as the broadcast that is used to create weavers, the weavers are still executed in order.

Since object masters can be on different processors, more than one processor may decide that a signal message is needed to wake up a single weaver. To avoid these spurious signal messages, the following optimization is used. Whenever a signal message arrives for a marked weaver, the mark is removed. For example, suppose both processor 0 and processor 1 want to send a signal message to wake up weaver A. If the signal message from processor 0 is received by processor 1 before all weavers are idle, the signal message will be enqueued and be executed before processor 1 has sent its signal message. Since the signal message removes the wakeup marker from weaver A, the signal message from processor 1 will not be sent.

5 Runtime Support using Weavers

We have implemented a runtime system based on weavers that integrates Orca objects and atomic functions. The runtime system has been implemented on a portable virtual machine for implementing parallel programming systems, called Panda [15], and has been tested on a collection of 64 Pentium Pros connected by Myrinet [5]. Panda provides threads, point-to-point communication and totally-ordered group communication. Below, we will show how weavers have been used to implement this runtime support system. First, we will map Orca objects, which only involve operations on single objects, on top of weavers. After that, the usage of weavers in the runtime support for atomic functions is presented.

5.1 Orca Objects

Operation invocations on Orca objects can be implemented directly using weavers. First, the runtime system determines whether the operation is a read operation, i.e., whether the operation does not always change the object during the execution. If so, one processor that has a copy of the object is selected (the local processor is selected if it has a copy) and a weaver is created on this single processor, passing the object identifier, the operation identifier, and the operation arguments as parameters. When the operation succeeds, the results are returned to the invoking process, after which this process can continue. If the operation would want to write the object, an error code is returned to the invoking process, telling it to retry the operation as a write operation.

To execute an operation that writes the object, a weaver is created that executes on *all* processors that contain a copy of the object. Since all these processors now run a weaver thread, they can all update their local state while preserving sequential consistency.

If the operation blocks, each weaver thread adds itself locally to the pending weaver queue of the object. Since a write operation is always executed on all copies, waking up a pending weaver after a successful write operation can be done locally (i.e., case one in Section 4).

5.2 Atomic Functions

To execute an atomic function, the runtime system computes the union of all processors that have a copy of any of the objects passed as parameter to the atomic function. A weaver is created that runs on this union set of processors, with the atomic function and the atomic function arguments (including the object pointers) as parameter. The weaver threads all invoke the atomic function, which starts to perform its statements.

Whenever the atomic function wants to perform an operation on an object, only those processors that have a local copy perform the operation, while the other processors skip the operation. However, at some point even those processors that did not have a copy of the object need to know the results of these operations to compute the result of the guard condition. Our runtime system handles these

	Remote	Replicated (8 processors)
Standard Orca	40.6 μs	64.7 μs
Weaver implementation	62.0 μs	90.2 μs

Table 1. Orca object invocation latencies.

data dependencies and tries to minimize the number of messages and the amount of data that needs to be sent [16]. Since this is part of the execution environment, we will not discuss this mechanism here. Sufficient to know is that weaver threads communicate so that all threads can evaluate the guard conditions.

If the atomic function completes its execution, all operations have been performed on all copies of the objects. Since this includes write operations and since the weaver is executed in a total order with respect to the other weavers, all objects contain the same state after the atomic function is finished. Therefore, sequential consistency is preserved.

If the atomic function blocks, the weaver adds itself to all objects that were passed as parameter. If any of these objects changes, the weaver is signaled. Note that since the weaver may be running on a superset of the processors that have a copy of the object that is updated, communication is required to restart the atomic function. This is the second case that is discussed in Section 4.

6 Performance

In this section, we will evaluate the performance of our runtime system based on weavers. First, we will look at the performance for Orca objects. Then, we will compare the performance of an application that uses either only Orca objects or Orca objects combined with atomic functions. All measurements have been performed on a system consisting of 200 MHz Intel Pentium Pros connected by Myrinet [5] and FastEthernet.

Table 1 shows the performance numbers for invoking an operation on a remote Orca object and on a replicated Orca object, both for the standard Orca runtime system as for our runtime system based on weavers. As can be seen, the performance of our implementation is up to 50% slower than the original Orca version. Part of this difference is the overhead introduced by the weaver queue and the extra software layering. Another part is caused by the fact that our prototype implementation needs more performance tuning. Furthermore, the communication latency is very low on Myrinet (the roundtrip latency is about 36 μs). If FastEthernet is used, the performance difference is less than 10%.

To evaluate the performance overhead of using atomic functions, we look at an application that uses a work queue, namely the Travelling Salesman Problem. Two versions will be compared using our runtime system: one that uses only a single Orca object (the work queue), and another that uses two Orca objects (a work queue and an active workers counter) and the atomic function *Terminate* (see Figure 2). For the single object implementation, a flag is added to the work queue object that is used to inform the worker processes that no more work will be generated.

Since we are interested in the performance of the atomic function, and not in the application performance, we ran the application with a small problem size (14 cities) and 60 processors. No speedup is achieved then, but the performance of the termination detection becomes a critical factor. If the work queue object and the active workers object are both single-copy objects on processor 0, the atomic function version performs only 6% worse than the single Orca object version. If the active workers object is replicated (as in Figure 3), the performance decrease is 12%. In this case, the atomic function is executed on all processors, which requires extra communication to propagate the boolean result of the *empty* operation.

7 Related Work

Atomic functions resemble the general atomic transaction model [10] (as used for databases), which allows atomic actions on arbitrary sets of data. Deadlocks, atomic commit, concurrency control, and fault tolerance are complicated and expensive to handle if arbitrary data can be accessed in a transaction. Moreover, for parallel programming, such a general mechanism is hardly ever required. Atomic functions differ from atomic transactions in that the objects to be accessed must be known when the function is started. Also, we do not deal with fault tolerance. These restrictions allow for an efficient implementation.

Jade provides *tasks* that are performed atomically [14]. As in atomic functions, the programmer has to specify the shared objects that a task can access. Every task is executed on a single machine. Therefore, a remote object is migrated to or replicated on the executing machine. In our model, all objects are updated in place, even objects that are replicated. Therefore, our atomic functions use threads on all processors that have access to any of the objects that will be accessed. Our model also allows multiple processes to invoke atomic functions and to synchronize with each other, while the Jade programming model is based on a sequential execution.

8 Conclusions

In this paper, we have described the weaver abstraction, a generic mechanism for implementing runtime support for operations on multiple objects. Weavers have been used to implement runtime support for Orca objects and atomic functions. All the synchronization requirements of this runtime system can be mapped onto two synchronization primitives provided by weavers, *signal* and *wait*. The weaver implementation takes care of keeping the execution of weavers in total order and of the synchronization between weavers.

We have implemented a prototype implementation of the weaver abstraction, and also prototype implementations of runtime support systems for Orca objects and atomic functions. Apart from synchronization, this runtime system also deals with the communication between weaver threads that is required to resolve data dependencies.

References

1. Thomas E. Anderson, David Culler, David A. Patterson, and the NOW team. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.
2. H. E. Bal. *Programming Distributed Systems*. Prentice Hall International, Hemel Hempstead, UK, 1991.
3. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed system. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
4. Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Cerial Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Portability in the Orca shared object system. Submitted for publication, 1997.
5. Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.
6. John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.
7. Miguel Castro, Paulo Guerdes, Manuel Sequeira, and Manuel Costa. Efficient and flexible object sharing. In *Proc. 1996 Int. Conf. on Parallel Processing*, volume I, pages 128–137, Bloomingdale, IL, Aug. 1996.
8. W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, 1992.
9. Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. In *Proc. 15th Int. Conf. on Distributed Computing Systems*, pages 439–449, Vancouver, Canada, May 1995.
10. J. Gray and A. Reuter. *Transaction Processing: Techniques and Concepts*. Morgan Kaufmann, San Mateo, CA, 1992.
11. Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proc. 15th ACM Symp. on Operating Systems Principles*, pages 213–228, Copper Mountain, CO, Dec. 1995.
12. Pete Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, Dec. 1994.
13. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
14. Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.
15. Tim Rühl, Henri Bal, Raoul Bhoedjang, Koen Langendoen, and Gregory Benson. Experience with a portability layer for implementing parallel programming systems. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, Aug. 1996.
16. Tim Rühl and Henri E. Bal. Optimizing atomic functions using compile-time information. In *Proc. 1995 Programming Models for Massively Parallel Computers*, pages 68–75, Berlin, Germany, Oct. 1995.
17. Daniel Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, Oct. 1996.
18. Gregory V. Wilson and Henri E. Bal. Using the Cowichan problems to assess the usability of Orca. *IEEE Parallel and Distributed Technology*, 4(3):36–44, fall 1996.