Implicitly Synchronized Abstract Data Types: Data Structures for Modular Parallel Programming

Martin C. Rinard
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

ABSTRACT

Abstract data types are used to control the complexity of developing serial programs. They promote modular programming by encapsulating state and operations on that state. In parallel environments abstract data types must also encapsulate the synchronization required to correctly sequence the invocation of specific operations. An abstract data type is implicitly synchronized if it completely encapsulates the synchronization required for its use. Implicitly synchronized abstract data types promote modularity and help programmers manage the complexity of developing parallel software. This paper introduces the concept of implicitly synchronized abstract data types and shows how the implicitly parallel language Jade supports their development and use.

1. Introduction

Over the last decade, research in parallel computer architecture has lead to the development of many new parallel machines. Collectively, these machines have the potential to dramatically increase the resources available to solve important computational problems. The widespread use of these machines, however, has been limited by the difficulty of developing useful parallel software.

Programmers have traditionally developed software for parallel machines using explicitly parallel languages.[4,11] These languages provide constructs that programmers use to create parallel tasks. The tasks typically interact using synchronization operations such as locks, condition variables and barriers and/or communication operations such as send and receive. Explicitly parallel languages give programmers maximum control over the parallel execution, and they can exploit this control to generate extremely efficient computations.

The problem is that explicitly parallel languages present a complex programming environment. A major source of this complexity is the need to manage many of the low level aspects associated with mapping a computation onto the parallel machine. The programmer must decompose the program into parallel tasks and generate the synchronization operations that coordinate the execution of the computation. Because the synchronization code controls the interactions between all of the parallel tasks, the programmer must develop a comprehensive mental model of the global concurrency structure and keep that model in mind when writing the synchronization code.

In serial environments programmers use abstract data types to effectively manage the construction of complex programs. Each abstract data type encapsulates the complexity of implementing a specific piece of functionality. Building the application in terms of abstract data types allows programmers to control the complexity of building a given application.

In parallel environments abstract data types must encapsulate more than just the representation of state and the implementation of operations that manipulate that state. Each abstract data type must also encapsulate the concurrency generation and synchronization code required for its correct use and present an interface that requires no information about the global concurrency pattern. If an abstract data type satisfies these properties we say that it is *implicitly synchronized*.

Implicitly synchronized abstract data types allow programmers to control the complexity of building parallel applications. Programmers can build complex parallel applications by combining implicitly synchronized abstract data types, with the task of synchronizing the parallel program effectively decomposed into the task of synchronizing each abstract data type. But explicitly parallel environments undermine this software development strategy. The fundamental problem with abstract data types in explicitly parallel environments is that their behavior depends on the order in which their operations are performed. In general, when parallel threads invoke the operations of a given abstract data type, the operations must execute in a specific order for the program to generate the correct result. This order typically depends on the semantics of the abstract data type's client, not on the semantics of the abstract data type itself. The synchronization required to generate this order leaks outside the abstract data type, forcing the client to deal with the complexity directly and destroying the modularity of the resulting program.

In this paper we demonstrate how to develop implicitly synchronized abstract data types in the context of the implicitly parallel language Jade.[7,6] Instead of using synchronization operations to directly control the parallel execution, Jade programmers declaratively provide information about how parts of the program access data. The Jade implementation analyzes this data access information to automatically exploit the concurrency present in the application. This declarative approach promotes modular parallel programming by allowing programmers to build abstract data types that encapsulate all of the code required to correctly exploit the concurrency.

The rest of the paper is organized as follows. In Section 2 we introduce the basic concepts of Jade and sketch how they can be used to build implicitly synchronized abstract data types. In Section 3 we describe the basic Jade construct and show how it can be used to build a simple abstract data type. In Section 4 we show how to build layered abstract data types and in Section 5 we briefly describe the more advanced Jade constructs. In Section 6 we describe how the Jade implementation supports implicitly synchronized abstract data types. In Section 7 we analyze how well other parallel programming paradigms support the construction of abstract data types and discuss how programmers synchronize computations written in languages that do not support implicitly synchronized abstract data types.

2. Jade

Jade is an implicitly parallel language designed for the exploitation of coarse grain, task level concurrency. Jade programmers start with a serial program and use Jade constructs to specify how parts of the program access data. The Jade implementation dynamically analyzes this information to automatically extract the task level concurrency present in the application. For pragmatic reasons the current version of Jade is structured as an extension to C. Stable implementations of Jade exist for a wide range of computational platforms, included shared memory multiprocessors (the Stanford DASH machine), homogeneous message passing machines (the Intel iPSC/860) and heterogeneous networks of workstations. Jade programs port without modification between all of the platforms.

Jade is based on three fundamental concepts: shared objects, tasks and access specifications. Shared objects and tasks are the mechanisms that the programmer uses to specify the granularity of the data and the computation, respectively. The programmer uses access specifications to specify how tasks access shared objects. The next few sections describe how programmers can use these mechanisms to build implicitly synchronized abstract data types.

2.1. Shared Objects

Jade supports the abstraction of a single mutable memory that all parts of the computation can access. Each piece of data allocated in this memory is called a shared object. Programmers use shared objects to implement the mutable state encapsulated in each abstract data type. Outside the abstract data type each reference to a shared object is treated as an opaque handle passed to abstract data type operations. Inside the abstract data type the operations manipulate the encapsulated state by reading and writing the corresponding shared objects.

2.2. Tasks

Jade programmers explicitly decompose the serial computation into tasks by identifying the blocks of code whose execution generates a task. In many parallel programming languages tasking constructs explicitly generate parallel computation. Because Jade is an implicitly parallel language with serial semantics, Jade programmers ostensibly use tasks only to specify the granularity of the parallel computation. The implementation, and not the programmer, then decides which tasks execute concurrently.

Programmers typically create a task to perform the computation associated with each operation on an abstract data type. The tasking construct itself is encapsulated inside the implementation of the abstract data type. If there is concurrency available within the operation, the task creates child tasks that cooperate to concurrently perform the independent parts of the operation.

2.3. Access Specifications

Each task has an access specification that declares how it will access individual shared objects. It is the responsibility of the programmer to provide an initial access specification for each task when that task is created. As the task runs, the programmer may dynamically update its access specification to more precisely reflect how the remainder of the task accesses shared objects.

The key to building implicitly synchronized abstract data types in Jade is developing a programming methodology that encapsulates all access specifications inside the definition of the abstract data type. The simplest abstract data types bundle the access specifications with the encapsulated tasking construct. Abstract data types that are used in more sophisticated contexts export operations that encapsulate access specifications.

2.4. Parallel and Serial Execution

The Jade implementation analyzes access specifications to determine which tasks can execute concurrently. This analysis takes place at the granularity of individual shared objects. The dynamic data dependence constraints determine the concurrency pattern. If one task declares that it will write an object and another declares that it will access the same object, there is a dynamic data dependence between the two tasks and they must execute sequentially. The task that would execute first in the serial execution of the program executes first in all parallel executions. If there is no dynamic data dependence between two tasks, they can execute concurrently.

This execution strategy preserves the relative order of reads and writes to each shared object. If a program only declares read and write accesses, the implementation guarantees that all parallel executions preserve the semantics of the original serial program and therefore execute deterministically. More sophisticated access specifications may allow the implementation to further relax the sequential execution order. For example, if two tasks declare that their accesses to a given object commute, the implementation has the freedom to execute the tasks in either order. In this case the implementation determines the execution order dynamically, with different orders possible in different executions.

This parallelization strategy correctly synchronizes the execution of programs that use abstract data types. The client simply invokes abstract data type operations and each operation generates a task to perform its computation. The Jade implementation analyzes the access specifications to determine which operations can execute concurrently. If operations must execute serially the implementation automatically generates the synchronization required to enforce the appropriate precedence constraints. The resulting parallel execution preserves the semantics of both the client and the abstract data type.

3. The Tasking Construct

We start our discussion of how to build implicitly synchronized abstract data types in Jade by presenting the `withonly` construct. This construct allows the programmer to identify a task and specify how it accesses shared objects. Each abstract data type operation uses this construct to generate a task to perform its computation. Figure 1 contains the general syntactic form of the construct.

```
withonly { specification } do (parameters) { body }
```

Figure 1: The `withonly` Construct

The `body` section contains the serial code executed when the task runs. The `parameters` section contains a list of variables from the enclosing environment. When the task is created, the implementation copies the values of these variables into a new environment; the newly created task will execute in this environment.

When a task is created, the Jade implementation executes the `specification` section to generate an initial access specification for the task. This section is an arbitrary piece of code containing *access declaration statements*. Each such statement declares how the task will access a given shared object; the task's access specification is the union of all such declarations. For example, the `rd(o)` (read) statement declares that the task may read the object `o`, and the `wr(o)` (write) statement declares that the task may write `o`. The `specification` section may contain dynamically resolved variable references and control flow constructs such as conditionals, loops and function calls. The programmer is therefore free to use information available only at run time when generating a task's access specification. The Jade implementation dynamically checks each task's accesses to ensure that it respects its access specification. If a task attempts to perform an access that it did not declare, the implementation will detect the violation and generate a run time error identifying the undeclared access.

3.1. An `index` *Example*

```
typedef struct { int key, data; } entry;
typedef entry shared *index;
```

Figure 2: Index Data Structure Declarations

We demonstrate how to develop implicitly synchronized abstract data types in Jade by presenting the implementation of a simple `index` abstract data type. This data type implements a mapping from keys to data items. The implementation of the `index` uses a closed hash table implemented with an array. Figure 2 defines the hash table data structure. The programmer uses the `shared` keyword to declare that the hash table is a shared object.

```
void lookup(index h,
  int k, int shared *d) {
 withonly {
  rd(h); wr(d);
 } do (h, k, d) {
  j = hash(k);
  first = j;
  while (TRUE) {
    if (h[j].key == k) {
      *d = h[j].data;
      break;
    }
    j++;
    if (j == SIZE_HASH)
      j = 0;
    if (j == first) {
      /* not found */
      *d = 0;
      break;
    }
  }
 }
}
```

Figure 3: Index Lookup Operation

```
void insert(index h,
  int k, int d) {
 withonly {
  cm(h);
 } do (h, k, d) {
  j = hash(k);
  first = j;
  while (TRUE) {
    if (h[j].key == 0) {
      h[j].key = k;
      h[j].data = d;
      break;
    }
    j++;
    if (j == SIZE_HASH)
      j = 0;
    if (j == first) {
      /* table full */
      break;
    }
  }
 }
}
```

Figure 4: Index Insert Operation

The index exports two operations: `lookup`, which retrieves the data item indexed under a given key, and `insert`, which inserts a data item into the index under a given key. Figure 3 contains the definition of the `lookup` operation. This operation takes three parameters: `h`, which points to the shared object containing the hash table, `k`, which is the key to look up, and `d`, which holds the result of the lookup. Following our programming methodology, this operation uses the `withonly` construct to create a task to perform the lookup. This task's access specification uses the `rd(h)` access declaration statement to declare that the task will read the hash table and the `wr(d)` access declaration statement to declare that the operation will write the result. Similarly, the `insert` operation in Figure 4 creates a task to perform the insertion. The semantics of the `index` abstract data type allow successive insertions to execute in any order as long as they each have exclusive access to the hash table. The programmer expresses this commutativity information with the `cm(h)` access declaration statement. This statement declares that the operation may read and write the hash table, but that its accesses commute with other operations that also declare commuting accesses.

The access specifications determine whether index operations execute serially or in parallel. Successive `insert` operations execute serially, although the actual execution order may vary from run to run. Successive `lookup` operations can execute concurrently. The implementation executes interleaved `lookup` and `insert` operations in the same order as in the original serial program.

An `index` is an implicitly synchronized abstract data type. It encapsulates the representation of the state, the implementations of the operations that manipulate the state, and, of equal importance, the constructs required to exploit correctly synchronized parallel execution both within and between operations. This full encapsulation means that client has no dependence on any aspect of the `index` implementation.

3.2. Using the `index` *Abstraction*

```
lookup(h, 1, d1);
insert(h, 2, 5);
insert(h, 3, 6);
lookup(h, 2, d2);
lookup(h, 3, d3);
```

Figure 5: Client Code

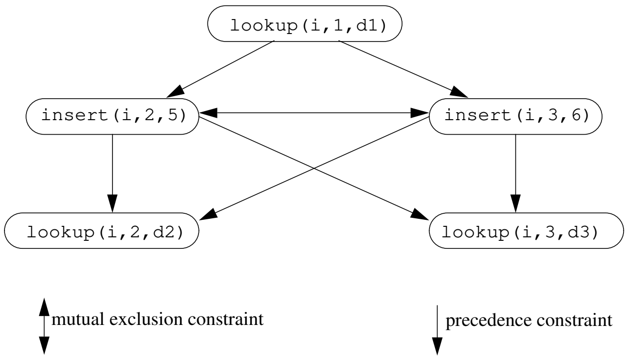


Figure 6: Task Graph

To use the `index` abstraction, the client just invokes the operations. Figure 5 contains a code fragment that uses an index. Figure 6 contains the dynamic task graph that this computation generates. The first `lookup` operation executes before the two `insert` operations. The `insert` operations commute, executing with exclusive access to the hash table. The final two `lookup` operations execute concurrently. The client contains no code that deals with the parallel execution.

4. Layered Abstract Data Types

Programmers often structure large, complex programs using nested layers of abstract data types, implementing one layer in terms of the abstraction presented by the next layer. In our methodology the logical nesting of abstract data types generates nested task creation. The task that performs a given operation will invoke operations on the next layer of abstract data types. These operations will in turn create child tasks to perform their operations.

In Jade, each task must declare how its entire computation accesses shared objects, including how its child tasks access shared objects. This requirement ensures that the Jade implementation has enough information to correctly synchronize the computation in the face of hierarchically created tasks that update mutable data. To satisfy this requirement, each operation's task must declare both how it will access data and how the operations that it invokes will access data.

4.1. Encapsulating Access Declarations

In layered contexts implicitly synchronized abstract data types preserve their encapsulation boundary by exporting operations that declare how other operations access data. In this methodology operations on abstract data types come in pairs. One operation performs the actual computation while an associated access declaration operation declares how the first operation will access shared objects. If a task invokes an operation, it includes the associated access declaration operation in its access specification section. Access declaration operations extend abstract data types for use in layered contexts without compromising encapsulation.

```
void declare_insert(h)   { df_cm(h); }
void declare_lookup(h,d) { df_rd(h); df_wr(d); }
```

Figure 7: Index Access Declaration Operations

Figure 7 shows the access declaration operations required to use the `index` abstract data type in layered contexts. The operations use the deferred form of the access specification statements. Deferred access specification statements declare that the task or its child tasks may eventually access the given object, but that the task itself will not do so immediately. It can therefore execute concurrently with earlier tasks that do access the object. When the task needs to access the object it upgrades the deferred declaration to an immediate declaration (Section 5 describes the construct used to perform the upgrade) and suspends until it can legally perform the access. In our example the client of the `index` abstraction will never directly access the index's hash table. It will instead invoke index operations, which in turn create child tasks that actually perform the access.

4.2. A Layered Abstract Data Type

We now demonstrate how to construct a layered abstract data type built on the `index` abstraction. This new abstract data type is part of an employee database system and, given an employee number, stores that employee's phone number and salary. Figure 8 contains the declaration of the shared object that implements this `record` abstraction. This shared object contains two `index` abstract data types. Figure 9 presents the implementation of the `store` operation, which stores a phone number and salary into the record. This operation uses the `declare_insert` access declaration operation to declare the accesses that the nested `insert` operations will perform. Because the two `insert` operations access different hash tables, they can execute concurrently.

5. Advanced Constructs

Jade also provides several more advanced constructs that allow the programmer to achieve more sophisticated parallel behavior. We next provide a brief overview of these constructs.

Figure 10 contains the general syntactic form of the `with` construct. As for the `withonly` construct, the `specification` section is an executable piece of code containing access declaration statements. Instead of creating a new access specification for a new task, however, the construct updates the access specification of the current task to more precisely reflect how the rest of the task will access shared objects. If the task originally declared a deferred access, for example, it may use a `with` construct to declare that it now needs to actually perform the access. The task will then suspend until it can legally perform the access. The

```
typedef struct {
  index salary,
        phone;
} shared *record;
```

Figure 8: Record Data Structure Definition

```
void store(record r,
    int e, int s, int p) {
  withonly {
    rd(r);
    declare_insert(r->salary);
    declare_insert(r->phone);
  } do (r,e,s,p) {
    insert(r->salary, e, s);
    insert(r->phone, e, p);
  }
}
```

Figure 9: Store Operation

task may also use a `with` construct to declare that it will no longer access a given object, potentially eliminating conflicts between the task executing the `with` and later tasks. In this case the later tasks may execute as soon as the `with` completes rather than waiting until the task itself finishes.

```
with { specification } cont;
```

Figure 10: The `with` Construct

Finally, the tasks in parallel programs that manipulate recursive data structures such as lists and trees often incrementally refine the set of list or tree nodes that they can potentially access as they traverse the list or tree. Jade supports these programs by allowing programmers to expose the hierarchical nature of the data structure. The basic idea is that there are parent objects and child objects. If a task declares that it will access a parent object, this implicitly gives the task the right to declare that it will also access a child object of the parent object, or to create child tasks that access child objects. If a task traverses a tree, for example, it initially declares that it will access the root node. At each step of the traversal it declares that it will next access one of the child nodes of the current node and no longer access the parent. Hierarchical data structures allow the programmer to express such concurrency patterns in an efficient and succinct manner.

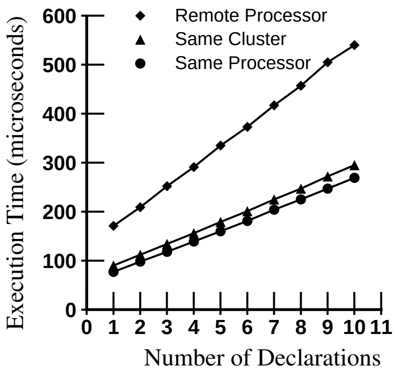
6. Implementation

The Jade implementation uses an object queue mechanism to detect concurrency and synchronize the computation. There is a queue associated with each object that controls when tasks can access that object. Each task has an entry in the queue of every object that it declares that it will access. Each entry declares how the task will access the object. Entries appear in the queue in the same order as the corresponding tasks would execute in a sequential execution of the program. Each entry stays in the queue until the task finishes or uses a `with` construct to eliminate the corresponding access declaration.

A task's entry is *enabled* when it can legally perform the declared access. A read entry is enabled when there are only read entries before it in the queue. A write entry is enabled when it is the first entry in the queue. A commuting entry becomes enabled when there are only other commuting entries before it in the queue and it has obtained exclusive access to the shared object. Because deferred declarations do not give tasks the right to actually access the object (the task must use a `with` construct to change the deferred declaration to an immediate declaration before performing the access), the corresponding entries are always enabled. Each task has a count of the number of its queue entries that are not enabled. When this count drops to zero the task is enabled and can legally execute. The DASH implementation uses a distributed task queue algorithm to assign enabled tasks to processors, applying a dynamic load balancing algorithm for good processor utilization. The current iPSC/860 implementation uses a centralized algorithm, with a single processor tracking processor usage and dynamically balancing the load.

In the current implementation almost all activities take place at run time. A simple preprocessor converts the `with` and `withonly` constructs into procedure calls into the Jade library. The execution of the access specifications, concurrency detection and processor scheduling all take place dynamically. The access checking required to ensure that tasks respect their access specifications, however, has significant preprocessor support. Conceptually, all access checks take place dynamically. But there is a mechanism in the Jade type system that programmers can use to ensure that each task performs at most one dynamic check per object. Because the checking overhead is amortized over many object accesses, in practice the total overhead is negligible.

Figures 11 and 12 graph the dynamic overhead per task as a function of the number of access specification statements for the current version of Jade. Com-



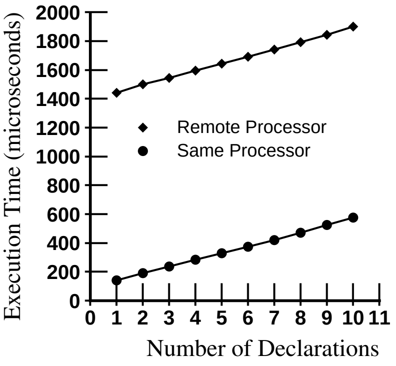


Figure 11: Task Overhead on DASH Figure 12: Task Overhead on iPSC/860

munication costs are a significant part of the overhead for remote task execution.

Jade is designed for applications that exploit task level concurrency. These programs contain tasks large enough to profitably amortize the dynamic task creation overhead. Because Jade is structured as a declarative extension to a serial programming language, it is possible to apply compiler techniques developed for analyzing serial programs directly to Jade programs. Static analysis would allow the implementation to implement statically analyzable concurrency patterns very efficiently. Such an implementation would support the exploitation of finer grain concurrency.

7. Related Work

Jade supports implicitly synchronized abstract data types that encapsulate mutable state. Other parallel languages support implicitly synchronized abstract data types that can be used in more restricted contexts. In this section we discuss several other languages and describe how programmers synchronize programs written in explicitly parallel languages.

7.1. Futures and I-Structures

Futures[3] and I-Structures[5] support implicitly synchronized data structures that are written only once. A future is a place holder that holds the result of a function that may execute concurrently with its caller. Parts of the computation that read the future implicitly suspend until the function generates the result. Like futures, I-structures implicitly support the producer/consumer synchronization patterns characteristic of many parallel programs. An I-structure initially starts out undefined, with parts of the program that read the I-structure implicitly suspending until the computation defines it.

Both futures and I-structures allow programmers to build implicitly synchronized abstract data types for computations with no mutable data. All synchronization is implicit; the producer need have no knowledge of how the consumer will use the value and the consumer needs to know nothing about how the producer will generate the value. Programmers can represent the state of abstract data types with data structures containing futures and I-structures. The operations either incrementally define the encapsulated data structure or read some part of it. These operations can be invoked from any part of the program with no additional synchronization. The only restriction is that the data structure cannot contain mutable data.

7.2. Concurrent Object Oriented Languages

Concurrent object oriented languages[2,1] combine the monitor concept with concepts from serial object oriented languages. The resulting languages are built on the abstraction of atomic object operations: each operation executes with exclusive access to the object. Some languages extend this model to allow operations that only read the object to execute concurrently. Many languages also provide condition variables. If an operation cannot complete without violating the object's internal consistency requirements, it suspends on a condition variable. When another operation makes it possible for the suspended operation to consistently complete, it signals the condition variable and the suspended operation continues.

These mechanisms support the construction of implicitly synchronized abstract data types when all operations from different parallel threads commute or when the objects' internal consistency enforcement mechanisms implement all of the precedence constraints required to correctly execute the program. But in many cases the program requires additional precedence constraints. In these cases the programmer must augment the program with the synchronization operations required to enforce these constraints. Because these operations must appear in the client of the object, they destroy the object's modularity.

7.3. Explicitly Parallel Languages

Programmers who use explicitly parallel languages can only build implicitly synchronized abstract data types for a restricted set of contexts. Other contexts impose precedence constraints that require the parallel tasks to perform the operations in a specific order. Because this order depends on the semantics of the client, it is impossible to encapsulate the required synchronization inside the abstract data type. Programmers have therefore evolved a set of less modular synchronization strategies.

Many parallel programs can be structured as a sequence of phases in which precedence constraints occur only between phases. Within phases all operations on a given data structure either commute or can execute concurrently. For example, the Barnes-Hut application for solving gravitational N-body problems organizes its data into an octree.[10] The computation can be decomposed into a tree building phase (in which all tree operations commute) and a tree use phase (in which all tree operations execute concurrently). For such computations programmers can easily build abstract data types that are implicitly synchronized within each phase.

The only problem left is generating the synchronization required to separate phases. The most common way to do this is to insert barriers between phases. When a thread completes its computation for one phase, it waits until all other threads have completed that phase. The threads then proceed to the next phase.

Barrier synchronization can reduce the cognitive complexity of generating the synchronization code by eliminating the need to consider interactions between phases. But it also wastes concurrency potentially available between phases and can cause poor performance if the computational load of each phase is not evenly balanced across the processors. The rigidity of barriers also limits the range of applications that they can effectively synchronize.

Event counts are another synchronization mechanism. The programmer may know that one operation on a data structure must be performed a certain number of times before another operation can legally execute. The programmer can synchronize the computation by maintaining a count of the number of times the first operation has been performed. If the second operation is invoked too early, the invoking task suspends until the required number of other operations execute. Many parallel sparse Cholesky factorization algorithms use this synchronization mechanism.[8] The problem with this mechanism is that some part of the program must know enough about the global structure of the computation to generate the number of times the different parts of the computation will update each data structure. SAM[9] introduces a similar mechanism: allowing the program to associate a version number with each mutable object. The version number counts the number of times the object has been modified to generate the current version. When a part of the program reads the object it specifies which version it needs to access, waiting until that version is generated. New versions are generated into new storage, leaving the old version intact for tasks to access.

8. Conclusion

We have described a methodology for parallel programming based on implicitly synchronized abstract data types. These abstract data types promote the development of modular parallel programs by encapsulating all of the code required to generate correct parallel execution. This paper also shows how to implement implicitly synchronized abstract data types in Jade, demonstrating how Jade's implicitly parallel approach allows programmers to adapt a proven methodology from serial computing for use in parallel contexts.

Acknowledgements
The author would like to thank Dave Probert and Klaus Schauser for their invaluable comments and recommendations.

1. H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.
2. R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.
3. R. Halstead, Jr. An assessment of Multilisp: lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, December 1986.
4. E. Lusk, R. Overbeek, J. Boyle, R. Butler, T. Disz, B. Glickfield, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.
5. R. Nikhil. Id version 90.0 reference manual. Technical Report 284-1, Computation Structures Group, MIT Laboratory for Computer Science, September 1990.
6. M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.
7. M. Rinard, D. Scales, and M. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.
8. E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford, CA, January 1993.
9. D. Scales and M. S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.
10. J. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*. PhD thesis, Stanford University, February 1993.
11. V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.