Journal of Programming Languages **6** (1998), 1–35

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, USA
rinard@mit.edu

Programmers use abstract data types to control the complexity of developing serial programs. Abstract data types promote modular programming by encapsulating state and operations on that state. In parallel environments abstract data types must also encapsulate the concurrency generation and synchronization code required for their correct use and present interfaces that require no information about the global concurrency pattern. An abstract data type is said to be implicitly synchronized if it meets these requirements. Implicitly synchronized abstract data types promote modular parallel software and help programmers manage the complexity of developing parallel programs. This paper defines the concept of implicitly synchronized abstract data types and shows how the implicitly parallel language Jade supports their development and use.

Keywords: Parallel programming, abstract data types

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

Programmers have traditionally developed software for parallel machines using explicitly parallel languages (Lusk et al. 1987, Sunderam 1990). 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.

The author is now at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

In serial environments programmers use abstract data types to manage the construction of complex programs. Each abstract data type presents an interface consisting of a set of operations that manipulate a given data structure. Because abstract data types hide the implementation of the data structure and operations from the clients that use them, they encapsulate the complexity of implementing the data structure and its operations. Programmers can therefore control the complexity of implementing a serial program by decomposing the program into a set of independent abstract data types.

In parallel environments abstract data types must encapsulate more than just the representation of data and the implementation of operations that manipulate that data. 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.

In this paper we first discuss some of the obstacles that programmers using explicitly parallel languages face when they attempt to develop implicitly synchronized abstract data types. The conclusion of this discussion is that, in general, explicitly parallel languages make it impossible to completely encapsulate the synchronization code required to correctly order operations on abstract data types: in many cases both the abstract data type and its clients must contain synchronization code to order the operations correctly. Programmers using explicitly parallel languages must therefore rely on less modular synchronization techniques or only use abstract data types in restricted contexts.

We then present the implicitly parallel programming language, Jade. Jade programmers start with a program written in a standard serial, imperative language, then use Jade constructs to describe how parts of the program access data. The Jade implementation analyses this information to extract the concurrency automatically and execute the program in parallel. The main contribution of this paper is to show how the Jade language design supports the effective construction of implicitly synchronized abstract data types. In particular, Jade's implicitly parallel approach effectively solves the operation ordering problem that often precludes the construction of implicitly synchronized abstract data types in explicitly parallel languages.

The rest of the paper is organized as follows. In section 1 we set the context for the paper by describing how existing parallel languages have approached the issue of modularity. Sections 2–7 present the Jade language and show how to build different kinds of implicitly synchronized abstract data types using Jade. Section 8 caps the discussion of Jade by providing a detailed exposition of the semantics of Jade constructs. Section 9 presents the key data structure, object queues, that the Jade implementation uses to extract the concurrency and synchronize the computation. This data structure allows the Jade implementation to present a high level parallel programming model that supports the construction of implicitly synchronized abstract data types. We describe our applications experience in section 10; this experience enables us to evaluate how the modularity ideas in Jade worked in practice.

1 Background

The issue of modularity arises in many parallel programming language designs. In this section we briefly discuss different approaches that various languages have taken.

1.1 Modularity in explicitly parallel languages

Many explicitly parallel languages contain features that are designed to support the construction of implicitly synchronized abstract data types. Almost all of these designs attempt to leverage the consistency properties of the abstract data type to synchronize the computation. The basic mechanism is to allow an abstract data type to delay the execution of operations that it is temporarily unable to execute in a satisfactory way. A standard example is a queue abstract data type that delays dequeue operations when there are no elements in the queue.

We briefly survey several proposed language mechanisms that are designed to allow programmers to specify when operations should be delayed. Path Pascal programmers can specify sets of enabled operations using path expressions (Campbell and Kolstad 1980, Kolstad and Campbell 1980). Each path expression specifies a partial order on the operations of a given abstract data type. At any given time the abstract data type can only execute operations that are consistent with the specified partial order. Argus provides atomic transactions that access multiple abstract data types (Liskov 1988). Conceptually, the Argus implementation ensures the serializability of the transactions by delaying the execution of operations that would interfere with their atomicity.

Many concurrent object-oriented languages have provided mechanisms that allow programmers to synchronize operations on objects. In the Actor model of computation, for example, the execution of an operation is delayed until the previously executing operation specifies a replacement behaviour for the object (Agha 1986). If an object cannot successfully execute a given operation in its current state, it can explicitly buffer the operation. It then resends the message to itself when it becomes able to execute the operation.

Other concurrent object-oriented programming languages have provided more sophisticated support for delayed operations. A basic mechanism is that each operation executes with exclusive access to the receiver object. This implicit form of mutual exclusion synchronization originally appeared in the context of monitors (Hoare 1974), and was used in monitor-based languages such as Concurrent Pascal (Brinch-Hansen 1975, Brinch-Hansen 1977) and Mesa (Mitchell et al. 1979, Lambert Butler and Redell 1980). Concurrent object-oriented languages also adopt this synchronization mechanism, in part because it meshes well with the concept of operations on objects. Many languages relax the exclusive execution constraint to allow the concurrent execution of operations that only read the same object. Like monitor languages, many concurrent object-oriented languages also provide condition variables to temporarily delay the execution of operations.

Some concurrent object-oriented languages provide higher-level support for delayed operations. In POOL-T and ABCL/1 each object's behaviour is determined by a script that it executes (America 1987, Yonezawa et al. 1986). When the object is ready to execute another operation, it executes a select construct that specifies the set of operations that it can legally execute at that point in time. The construct blocks until one of the operations attempts to execute. The Rosette system (Tomlinson and Singh 1989) allows programmers to identify a set of object states and specify the set of operations that an object in a given state can legally execute. When an operation completes its execution it specifies the object's next state. Capsules (Gehani 1993) allow the programmer to declaratively specify conditions that the object's state must meet for it to execute each operation.

1.2 A fundamental problem with explicit parallelism

In all of the explicitly parallel languages described above, the basic assumption is that multiple threads of control will attempt to asynchronously perform operations on abstract data types. The goal of the synchronization code is to order the operations correctly. But this assumption introduces a fundamental problem into the programming model. The behaviour of an abstract data type depends on the order in which its operations execute. In general, the operations must execute in a specific order for the program to generate the correct result. This order typically depends not only on the semantics of the abstract data type, but also on the semantics of its clients. It is therefore impossible for the abstract data type to contain all of the synchronization code required for its correct use. The clients must also contain synchronization code that helps to impose a correct operation execution order. This synchronization code violates the encapsulation of the abstract data type and destroys the modularity of the program.

We illustrate this problem with an example. Consider a bank balance abstract data type that has deposit and withdraw operations. The bank's policy is to charge a penalty if the balance drops below a certain minimum amount. The ending balance in the account may depend on the order in which the deposit and withdraw operations execute – in some execution orders the account may incur penalties; in others it may not. For a program that uses this abstract data type to execute correctly, it must execute the deposit and withdraw operations in a correct order. Furthermore, the abstract data type by itself cannot determine what a correct order should be – from the perspective of the abstract data type, orders that generate minimum balance penalties are just as consistent as orders that do not. To use this abstract data type correctly in an explicitly parallel context, the clients must contain additional synchronization that orders the deposit and withdraw operations. This synchronization violates the encapsulation of the bank balance abstract data type.

1.3 Synchronization strategies for explicitly parallel languages

Because explicitly parallel languages do not support implicitly synchronized abstract data types, programmers have evolved a set of less modular synchronization strategies. An examination of these strategies provides additional insight into the synchronization problems that programmers face when they use explicitly parallel languages.

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 (i.e. generate the same result regardless of the order in which they execute) or can execute concurrently. For example, the Barnes–Hut application for solving gravitational N-body problems (Singh 1993) organizes its data into an octree. 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 (Rothberg 1993). 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.

The SAM (Scales and Lam 1994) and VDOM (Feeley and Levy 1992) systems introduce another mechanism: associating 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.

Both SAM and VDOM rely on some external mechanism to determine which version of each object the program needs to access. Requiring the client of the object to correctly generate the version identifiers can destroy modularity. Each part of the program that uses the object must know how many times other parts of the program have written the object. SAM addresses this problem by providing libraries that encapsulate the version identifier calculation for common data structures with common accessing patterns. In effect, each such data structure is an implicitly synchronized abstract data type built on top of the version number synchronization mechanism.

1.4 Eliminating mutable data

One way to eliminate the synchronization problems associated with explicit concurrency is to eliminate mutable data. Futures (Halstead 1985, Halstead 1986) and I-Structures (Arvind and Thomas 1981, Nikhil and Pingali 1989, Nikhil 1990) 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.

Futures and I-structures support a monotonic model of computing. Instead of modelling computation as reads and writes to a mutable shared memory, languages built on these mechanisms model computation as the monotonic generation of information. This monotonicity enables the producer and consumer to synchronize implicitly via the value definition mechanism.

Both futures and I-structures allow programmers to build implicitly synchronized abstract data types for monotonic computations. 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.

1.5 Jade

Like the explicitly parallel languages described in section 1.1, Jade supports a programming model based on operations on abstract data types that implement their state with mutable data. The key difference is that Jade is an implicitly parallel language. Its sequential semantics totally orders the operations in the program. Programmers using explicitly parallel languages must write complex synchronization code to regenerate a correct execution order for multiple operations executed by multiple parallel threads of control. Jade programmers, on the other hand, write code that declaratively specifies how operations in a serial program will access the data that implements the abstract data type's internal state. The Jade implementation then relaxes the serial execution order to generate parallel execution. Independent operations (operations are independent if neither accesses a piece of data that the other writes) can execute concurrently. If operations are not independent, the Jade implementation preserves the correct execution order from the original serial program. As we will see, this approach allows Jade programmers to build abstract data types that completely encapsulate all of the synchronization and concurrency generation code required for their correct use.

Although we focus on how Jade supports implicitly synchronized abstract data types, we note in passing that Jade's implicitly parallel programming model offers several other advantages over explicitly parallel models. Explicit parallelism introduces the possibility of undesirable program behaviour such as deadlock or livelock. It also introduces the possibility of nondeterministic execution, which is often undesirable, for example, because it complicates the debugging process. Because Jade preserves the serial semantics, it is impossible to write a Jade program that deadlocks or livelocks. Jade programs are also guaranteed to execute deterministically. (Section 3.3 introduces an extension, commuting access declaration statements, that programmers can use to create nondeterminisitic execution.)

Although the current Jade implementation uses no sophisticated static analysis, the language is designed to enable such analysis. Because Jade programs have the same semantics as the serial languages that compilers are designed to work with, compilers can apply the same analysis techniques and transformations to Jade programs that they apply to serial programs. Explicit parallelism, on the other hand, can dramatically reduce the compiler's ability to analyse the program and apply transformations (Chow and Harrison 1992).

2 The Jade programming language

Jade programmers start with a serial program and use Jade constructs to specify how parts of the program access data. The Jade implementation dynamically analyses this information to automatically extract the 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, including 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, respectively, the data and the computation. 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.

int **shared** **value*;

Fig. 1. Shared object declaration.

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. Programs use pointers to refer to shared objects. Programmers use the **shared** keyword to identify pointers to shared objects; Fig. 1 shows how to use this keyword to declare a pointer to an *int* 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 conceptually 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. In a language with explicit support for abstract data types the compiler would enforce the encapsulation of shared objects. Because C has no such support, the encapsulation is a programming convention in the current implementation of Jade.

2.2 Tasks

Jade programmers explicitly decompose the serial computation into tasks. Each task corresponds to the execution of a block of code; the programmer uses a Jade construct (the **withonly** construct described in section 3) to explicitly identify the blocks of code whose execution generates a task.

Because Jade is an implicitly parallel language with serial semantics, Jade programmers use tasks only to specify the granularity of the parallel computation. The implementation, and not the programmer, decides which tasks execute concurrently. A legal Jade implementation may, for example, simply execute the program sequentially. The Jade tasking construct therefore identifies potential concurrency and does not force the implementation to provide the abstraction of parallel execution. The ICC++ **conc** construct exhibits a similar property. While the ICC++ implementation may execute the statements in a **conc** block concurrently (subject to local data dependence constraints), it is also free to execute the statements serially (Chien et al. 1996). A major difference is that the ICC++ implementation is only guaranteed to preserve the local data dependence constraints. The Jade implementation preserves all of the data dependence constraints.

In most parallel programming languages the implementation is not free to execute tasks sequentially – it must preserve the abstraction of parallel execution. In these languages tasks interact using standard synchronization and communication constructs such as monitors (Hoare 1974), futures (Halstead 1985) and message-passing operations (Pierce 1988). Programmers can use these constructs to write tasks that can not complete without interacting with other tasks. To execute the program correctly, the implementation must preserve the abstraction that tasks execute in parallel. Failure to do so may lead to deadlock. This issue arises when an implementation executes tasks sequentially to eliminate task creation and management overhead. To avoid deadlock, the implementation must be prepared to undo the decision to sequentialize a task; lazy task creation provides this functionality (Mohr et al. 1990).

Jade 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

In Jade, 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 analyses access specifications to determine which tasks can execute concurrently. This analysis takes place at the granularity of individual shared objects. For access specifications that only declare reads and writes, 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 analyses 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 2 presents the general syntactic form of the construct.

The *task body* section contains the serial code executed when the task runs. The *parameters* section contains a list of variables from the enclosing environment. These variables may be either base values from the underlying language (*int*, *double*, *char*, etc.) or pointers to shared objects. When the task is created, the implementation copies the values of these variables into a new environment; the newly created task

```
withonly { access specification } do (parameters for task body) {
    task body
}
```

Fig. 2. The **withonly** construct.

will execute in this environment. This environment construction mechanism is similar to the standard call-by-value procedure call mechanism. If a parameter points to a shared object, inside the task it still points to the same object as it did in the context enclosing the task. There is no implicit deep copy. It would be straightforward for the compiler to automatically generate the *parameters* section by performing a simple scan of the task body to identify all of the accessed variables from the enclosing context. We elected, however, to emphasize the similarity with the standard parameter passing mechanism by requiring the programmer to explicitly list the task parameters.

When a task is created, the Jade implementation executes the *access 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. The task uses pointers to shared objects to declare which objects it will access. For example, the **rd**(o) (read) statement declares that the task may read the shared object that *o* points to, and the **wr**(o) (write) statement declares that the task may write the shared object that *o* points to. The declaration order does not matter – if a task declares that it will read or write an object, it can perform the access an arbitrary number of times at any point during its execution. The *access 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 violate its access specification, the implementation will detect the violation and generate a run-time error identifying the undeclared access. The Jade implementation does not check for lower level errors such as incorrect casts or array bounds violations. We view any such checking as the responsibility of the language that Jade extends.

3.1 A methodology for simple abstract data types

The **withonly** construct supports a methodology for building simple abstract data types. Starting with a serial implementation of the abstract data type, the programmer uses the **withonly** construct to make each operation into a separate task. The *access specification* section of each **withonly** construct declares how the operation will access the data structure representing the state of the abstract data type.

Because the **withonly** construct is encapsulated inside each operation, the new parallel abstract data type presents the same interface as the original serial abstract data type. When clients invoke operations, the Jade implementation dynamically analyses the data usage declarations to concurrently execute operations with no dynamic data dependences. Because the abstract data type completely encapsulates the code required to generate correct parallel execution, the abstract data type is implicitly synchronized.

```
typedef struct {
  int key, data;
} entry;

typedef entry shared *index;
```

Fig. 3. Index data structure declarations.

3.2 An example: the index abstract data type

We illustrate our programming methodology by presenting the implementation of a simple index abstract data type. This data type implements a mapping from keys to data items; this data type is also known as a dictionary. The implementation of the index uses a hash table implemented with an array. To find the element corresponding to a given key, the abstract data type hashes the key to obtain an index into the array. If there are no collisions the element will be stored in the array element stored at that index. It uses linear probing to resolve collisions in the array (Sedgewick 1988). Figure 3 defines the hash table data structure. The programmer uses the **shared** keyword to declare that the hash table is a shared object.

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 4 contains the definition of the *lookup* operation. This operation takes three parameters: *i*, 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. The operation first hashes the key, then uses the result to index the array that implements the hash table. A check of the key stored in the array element determines if that entry contains the correct item. Because of the linear probing algorithm used to resolve collisions, the item may not be in the first array element that the operation checks. It therefore linearly searches the array starting at the first array element looking for an entry with the correct key. 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 $\mathbf{rd}(i)$ access declaration statement to declare that the task will read the hash table and the $\mathbf{wr}(d)$ access declaration statement to declare that the operation will write the result.

Similary, the *insert* operation in Fig. 5 creates a task to perform the insertion. The insertion first hashes the key to find the array element in which to store the item. If another key/item pair has already hashed to that array element, the operation linearly searches the array looking for a free entry. When it finds a free entry it inserts the item. Because the operation will both read and write the hash table, the **withonly**'s access specification section contains both a $\mathbf{rd}(i)$ and a $\mathbf{wr}(i)$ access declaration statement. (The $\mathbf{wr}(i)$ access declaration statement by itself only allows the task to write but not read the hash table.)

The *index* is an implicitly synchronized abstract data type. All of the code required to generate correct parallel execution is encapsulated in its implementation. The tasking constructs and access declaration statements are completely encapsulated inside the operations in the sense that the client can not observe the presence or absence of these statements. It is possible to substitute parallel and serial implementations of the abstract data type without changing the client.

The access specifications determine whether successive index operations execute serially or in parallel. Successive *insert* operations execute serially. Successive *lookup* operations can execute concurrently. Moreover, the implementation preserves the sequential execution order for tasks with dynamic data de-

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

Fig. 4. Index lookup operation

```
void insert(index i, int k, int d) {
  withonly { rd(i); wr(i); } do (i,k,d) {
    int j, first;
    j = hash(k);
    first = j;
    while (TRUE) {
      if (i[j].key == 0) {
        i[j].key = k;
        i[j].data = d;
        break;
      }
      j++;
      if (j == SIZE_HASH) j = 0;
      if (j == first)
      /* table full */
      exit(-1);
    }
  }
}
```

Fig. 5. Index insert operation.

$lookup(i, 1, d1)$;
$insert(i, 2, 5)$;
$insert(i, 3, 6)$;
$lookup(i, 2, d2)$;
$lookup(i, 3, d3)$;

Fig. 6. *index* client code

pendences. For example, the implementation executes interleaved *lookup* and *insert* operations in the same order as in the original serial program. Because this ordering preserves the data dependences of the original serial program, the Jade program is guaranteed to generate the same result. This execution strategy therefore preserves such desirable characteristics as deterministic execution.

Superficially, the access declaration statements resemble read and write lock constructs in explicitly parallel languages. It is important to realize that there is a fundamental difference between Jade and explicitly parallel languages with read and write locks. Read and write locks merely ensure that if a task writes an object, it will execute in some sequential order with respect to other tasks that access the object. But the order is arbitrary. The programmer must insert additional synchronization to ensure that the program generates a correct execution order. All too often the result is a set of synchronization constructs scattered throughout the program. Because these constructs interact with each other in complicated and often unanticipated ways, they impair the structure and modularity of the program. Because Jade preserves the original sequential execution order between tasks that write an object and tasks that access the object, there is no need for additional concurrency management code.

3.3 Commuting operations

The semantics of the *index* abstract data type allow successive insertions to execute in any order as long as they have exclusive access to the hash table. The programmer may express this commutativity information by replacing the **rd**(i) and **wr**(i) access declaration statements in Fig. 5 with the **cm**(i) 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.

3.4 Using the index abstraction

To use the *index* abstraction, the client just invokes the operations. Figure 6 contains a code fragment that uses an index. Figure 7 contains the dynamic task graph that this computation generates under the assumption that the programmer uses the **cm**(i) access declaration statement in the *insert* operation. 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.

3.5 Using commuting operations

Because the Jade implementation does not verify that operations declared to commute do in fact commute, commuting access declaration statements introduce the possibility of nondeterministic execution. We therefore view commuting operations as an extension of the basic Jade paradigm rather than an integral part of it. We expect programmers to only use commuting access declaration statements when the

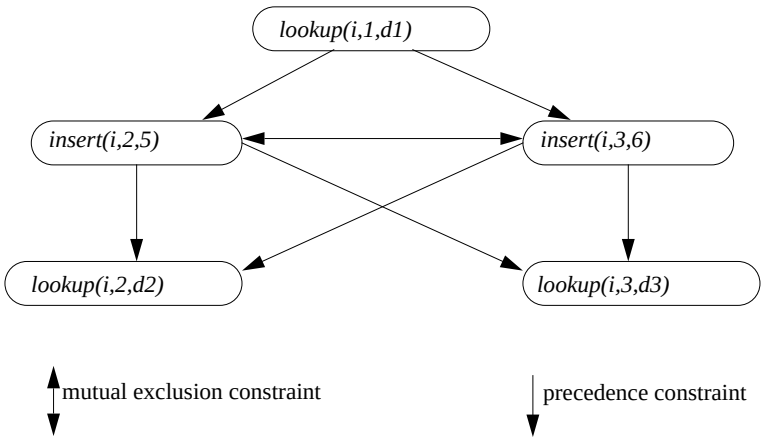


Fig. 7. Task graph

operations return equivalent values to their callers and leave the abstract data type in equivalent states regardless of the order in which they execute. (We assume the following notion of equivalence: two return values are equivalent if they are equal. States are equivalent if it is impossible to observe that they are different – in other words, all sequences of operations that execute on abstract data types in equivalent states return equivalent values.) These two properties together ensure that programs with commuting operations preserve the semantics of the original program.

The granularity at which the operations access the state of the abstract data type can have a significant impact on whether they commute or not. Commuting operations typically access the state of the object multiple times; in most cases all of the accesses must be performed atomically for operations to commute. The insert operation in Fig. 5, for example, first finds an empty array element, then stores the new entry into that element. The index abstract data type must implement and export its insertion functionality at at least that combined granularity if the operations that implement the insertion functionality are to commute.

Granularity issues may also arise at the level of multiple abstract data types. It is not difficult to imagine a sequence of operations on several abstract data types that commutes with other sequences of operations. The individual operations themselves, however, may not commute with the individual operations in other sequences. Consider, for example, a bank account abstract data type that exports two operations: deposit-with-balance, which deposits an amount into the account and returns the new balance, and withdraw-with-balance, which withdraws an amount and returns the new balance. (We would like to thank the first reviewer for this example.) These operations do not commute: even though they leave the account in the same state regardless of the order in which they execute, the return values may be different in different execution orders. It is possible to use withdraw-with-balance and deposit-with-balance to implement a transfer-with-balance computation. This computation transfers an amount from one account to another and returns the combined amount in the two accounts. Multiple transfer-with-

balance computations commute, but the operations used to implement the computation do not commute. The commutativity arises only because of interactions between the two account abstract data types that participate in the computation.

The methodology presented in this paper for implicitly synchronized abstract data types does not support this kind of commutativity – in part because the commutativity arises only at the level of multiple abstract data types. Jade, however, does support this kind of commutativity. A Jade task can declare commuting accesses to multiple shared objects. As long as the task performs the accesses sequentially, the computation will execute correctly.

It is also possible to imagine a set of operations on several abstract data types that commutes with other sets of operations. But within each set operations may execute concurrently with or commute with other operations in the same set, as long as each set as a whole executes atomically with respect to the other sets. It might be possible, for example, to execute the withdraw-with-balance and deposit-with-balance operations that together make up a transfer-with-balance computation concurrently. Jade does not support these kinds of concurrency patterns – there is no easy way to express the relaxation of the sequential execution order in a way that not only exposes the concurrency and commutativity but also preserves the atomicity of each set of operations relative to the other sets of operations.

This discussion illustrates a fundamental limitation of the concept of implicitly synchronized abstract data types. There are concurrency patterns whose minimal synchronization constraints arise because of interactions between abstract data types. In such situations it may be impossible to both maintain the modularity of the abstract data types and fully exploit the concurrency and commutativity. The methodology described in the paper imposes a conservative solution: it uses the original sequential execution order for tasks with dynamic data dependence constraints. This solution preserves the modularity boundaries, but may not fully exploit the available commutativity.

3.6 Abstract data types with multiple shared objects

The index abstract data type described above implements its state using a single shared object. It is of course possible to conceive of more complex abstract data types that implement their state with multiple shared objects. This situation may arise, for example, if the abstract data type encapsulates some private data that is not visible outside the implementation. Jade fully supports this implementation strategy: the access specification sections inside the abstract data type simply use multiple access declaration statements to specify how the operations will access the individual shared objects that together implement the abstract data type's state. Jade also supports more structured versions of this approach. Section 4 below presents a programming methodology for layered abstract data types; section 7 describes how Jade supports hierarchical abstract data types. In both of these cases the abstract data types implement their internal state with multiple encapsulated shared objects.

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 interface exported by the next layer. In our methodology the logical nesting of abstract data types generates nested task creation. A task that performs a given operation invokes operations on the next layer of abstract data types. These operations in turn create child tasks that perform the actual operation.

```
void declare_insert(index i, int k, int d) {
  df_cm(i);
}
void declare_lookup(index i, int k, int shared *d) {
  df_rd(i); df_wr(d);
}
```

Fig. 8. Index access declaration 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 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.

Figure 8 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 declaration statements (**df_rd**(o), **df_wr**(o), and **df_cm**(o)) rather than the immediate form that appears in earlier examples. The deferred form declares that the task may eventually access the object, but that it will not do so immediately. Before the task can access the object, it must change its deferred access declaration to an immediate access declaration (section 6 below describes the **with** construct that the task uses to change its access specification). This information about when a task will perform its accesses may allow the implementation to further relax the sequential execution order. For example, if a task declares an immediate access, it can not execute until all previous tasks that access the object have completed. If the task declares a deferred access, it may execute concurrently with other previous tasks that also access the object. Only when the task changes its deferred access declaration to an immediate access declaration does it have to wait for previous tasks that access the object to complete.

In addition to giving a task the right to eventually access an object, deferred access declarations also enable a task to create child tasks that declare that they will access the object. In the 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 to actually perform the accesses.

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

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

Fig. 9. Record data structure definition.

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

Fig. 10. Store operation

that employee's phone number and salary. Figure 9 contains the declaration of the shared object that implements this *record* abstraction. This shared object contains two *index* abstract data types.

Figure 10 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.

4.3 Access declaration operations

Access declaration operations are the key addition required to extend the programming methodology to layered abstract data types. They enable each operation to declare how all of its invoked operations will access data without violating the encapsulation of any of the abstract data types. But the addition of these operations complicates the methodology – they increase the size of each abstract data type's interface and appear to create an additional programming burden. We believe, however, that the complication is minimal. First, the access declaration operations are used in a highly regular, stylized way. Access declaration operations and normal operations come in pairs. This association dramatically reduces the cognitive complexity of the interface. Second, it is possible in our methodology to automatically generate the access declaration operations. Each operation consists of a **withonly** construct whose task body performs the computation associated with the operation. The corresponding access declaration operation consists of the code from the access declaration section of the **withonly** construct with all immediate access declaration statements converted to the corresponding deferred access declaration statements. If the abstract data type is layered, the access specification sections may invoke access declaration operations for operations on nested abstract data types. These invocations are simply transferred unchanged into the body of the automatically generated access declaration operation. Although we have not built a tool to automatically generate access declaration operations, we believe the fact that it would be straightforward to do so attests to their minimal engineering impact.

Access declaration operations also raise a potential efficiency issue. Because the implementation executes them dynamically, they may increase the overhead. The problem obviously becomes worse for deeply layered abstract data types. For these abstract data types the access declaration operations may introduce additional traversals of the layered objects as the access declaration operations traverse the data structures to generate correct access specifications. In general the overhead becomes an issue if the task size is not large enough to amortize it away profitably.

Finally, situations may arise in which a significant part of a task's computation is devoted to determining precisely which objects it will access. Most of these cases are best handled using hierarchical objects as described below in section 7. In some cases, however, the inability to quickly generate precise access specifications may impose unacceptable overhead. The task may duplicate computation performed in its access specification section as it computes which objects to access.

4.4 Atomicity and layered abstract data types

Each operation on a layered abstract data type generates multiple operations on the nested objects that implement its state. If all of the abstract data types only use read and write access declaration statements, it is clear that the clients cannot observe this decomposition. The serial semantics ensures that the execution order preserves the data dependences, which guarantees that the program generates the same result as if it executed serially.

With commuting access declaration statements the situation is a bit more complicated. In our example the two *store* operations will generate two insertions into the salary index and two insertions into the phone index. Furthermore, the declaration that *insert* operations commute enables the Jade implementation to perform the insertions into the salary database in a different order from the insertions into the phone database. The *store* operation may therefore not execute atomically in the sense that *insert* operations from one *store* operation may execute in different orders with respect to *insert* operations from another *store* operation.

The key point is that if the operations commute according to the definition in section 3.3, it is impossible for the clients to observe this lack of atomicity. Any operation that performs *lookup* operations on the two *index* abstract data types will be ordered in one of three ways with respect to the two *store* operations: either before both of the operations, after both of the operations, or between the two operations. If the *lookup* is ordered before the two *store* operations, it will execute before all of the *insert* operations that the *store* operations generate. If it is ordered after the two *store* operations, it will execute after all of the *insert* operations have completed. If it is ordered between them, both *insert* operations from the first *store* operation will have completed and neither *insert* operation from the second *store* operation will have executed. In no case will the *lookup* be able to observe a state in which one of the *insert* operations from a given *store* operation has executed and the other *insert* operation has not. The implementation therefore preserves the atomic semantics of the *store* operation, even though the implementation may interleave the execution of *insert* operations from different *store* operations. As long as the operations commute, this argument generalizes to arbitrary layered abstract data types that contain commuting access declaration statements.

5 Creating objects

Jade programmers create new objects using the **create_object** construct. Figure 11 shows how to use this construct to implement a *create_index* operation that creates a new index. This operation uses the

```
index create_index(int n) {
  return(create_object(entry[n]));
}
```

Fig. 11. Creating a new index.

with { *access specification* } **cont**;

Fig. 12. The **with** construct.

create_object construct to create and return a new array of *entry* data structures. The **create_object** construct itself takes one parameter: the type of the object to create. The task that created the object automatically obtains a deferred read, write and commuting access declaration on the created object. It can therefore create child tasks that declare that they will access the new object or use a **with** construct to declare that the task itself will access the new object.

6 The **with** construct

Each abstract data type operation uses the **withonly** construct to specify how it will access shared objects and to create a task to perform its computation. Jade also provides another construct (the **with** construct) and several additional access declaration statements. Programmers can use these constructs to update tasks' access specifications. This mechanism allows programmers to exploit pipelining concurrency available between operations on the same abstract data type.

Figure 12 presents the general syntactic form of the **with** construct. (The **cont** keyword stands for continue and is intended to emphasize the imperative interpretation of the construct.) As in the **withonly** construct, the *access specification* section is an arbitrary piece of code containing access declaration statements. These statements refine the task's access specification to more precisely reflect how the remainder of the task will access shared objects.

The **no_rd**(o) (no future read to o), **no_wr**(o) (no future write to o) and **no_cm**(o) (no future commuting access to o) allow programmers to declare that a task will no longer access a shared object in the specified way. This reduction may eliminate conflicts between the task executing the **with** and later tasks. The later tasks may therefore execute as soon as the **with** executes rather than waiting until the first task completes.

Programmers can also use the **rd**(o), **wr**(o) and **cm**(o) access declaration statements in a **with** construct to change a deferred access declaration to the corresponding immediate access declaration. In this case the task waits until it can legally perform the access, then proceeds. It is also possible to use the **df_rd**(o), **df_wr**(o) and **df_cm**(o) access declaration statements in a **with** construct to change an immediate access declaration back into a deferred declaration.

In the absence of hierarchical objects (described in section 7), a **with** construct can only change a deferred access declaration to an immediate access declaration, an immediate declaration to a deferred declaration or eliminate a declaration. It cannot add a declaration on another object to the task's access specification. In the absence of hierarchical objects, the only way a task can acquire its initial access declaration on an object is to either create the object (see section 5) or to have its initial **withonly** statement declare that the task will access the object. The **with** construct is therefore used for two purposes: to allow tasks to access objects that they create and to allow tasks with dynamic data dependences to overlap the

```
typedef struct t shared *tree;
typedef struct t {
  int key, data;
  tree left, right;
} node;
typedef tree index;
```

Fig. 13. Tree data structure declaration

parts of their execution that are independent.

7 Hierarchical objects

Many programs manipulate hierarchical data structures such as lists and trees. Jade supports these programs by allowing programmers to expose the hierarchical structure between shared objects. Hierarchical objects allow programmers to more conveniently and efficiently express how programs access hierarchical data structures.

The basic idea is that there are parent objects and child objects. The parent–child relationship is established when the child object is created. If a task declares that it will access a parent object, this declaration 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. Because the concept of hierarchical objects nests, the programmer can create tasks that incrementally refine their access specifications as they traverse hierarchical data structures.

7.1 A binary lookup tree index

We illustrate the use of hierarchical objects by presenting another implementation of the *index* abstract data type. This implementation represents the index with a binary search tree. Figure 13 contains the declaration of the tree node shared object. Each node points to a left node and a right node; each node is a child object of the node that points to it.

Programmers create hierarchical objects using the **create_child_object**(p,t) construct. The first parameter (p) is a pointer to the parent object of the new child object. The second parameter (t) is the type of the new object. The construct returns a pointer to the new child object. Figure 14 illustrates the use of the **create_child_object** construct.

Figure 15 contains the definition of the *insert* operation. This operation is implemented with a wrapper routine that creates a task to recursively perform the actual insertion. The task walks down the tree looking for an empty slot. When it finds an empty slot it calls the *create_node* operation to create a new tree node, then inserts the new node into the empty slot. The *create_node* operation uses the **create_child_object** construct to make the new node a child object of its parent node in the tree.

Structuring the tree as hierarchical objects allows the implementation to cleanly describe this data access pattern. At every step in the traversal the task declares that it will commutatively access the current node. When the task decides which subtree to search, it uses a **with** construct to describe how the step changes its access specification. The **with** construct uses the **wr**(c) statement to declare which child node it will access and the **no_wr**(t) access declaration statement to declare that it will no longer access the current node. Each **with** construct therefore refines the task's access specification to match the way it

```
tree create_node(tree t, int k, int d) {
  tree c = create_child_object(t, node);
  withonly { wr(c); } do (c,k,d) {
    c → key = k;
    c → data = d;
  }
  return(c);
}
```

Fig. 14. Tree node creation routine.

```
void declare_insert(tree t, int k, int d) {
  df_wr(t);
}

void insert(tree t, int k, int d) {
  withonly { wr(t); } do (t,k,d) {
    aux_insert (t,k,d);
  }
}

void aux_insert(tree t, int k, int d) {
  tree c;
  if (k < t → key) {
    if (t → left == NULL)t → left = create_node(t,k,d);
    else {
      c = t → left;
      with { wr(c); no_wr(t); } cont;
      aux_insert(c,k,d);
    }
  } else if (k > t → key) {
    if (t → right == NULL)t → right = create_node(t,k,d);
    else {
      c = t → right;
      with { wr(c); no_wr(t); } cont;
      aux_insert(c,k,d);
    }
  } else t → data = d;
}
```

Fig. 15. Tree insert routines.

```
void declare_lookup(tree t, int k, int shared *d) {
  df_rd(t); df_wr(d);
}

void lookup(tree t, int k, int shared *d) {
  withonly { rd(t); wr(d); } do (t,k,d) {
    aux_lookup(t,k,d);
  }
}

void aux_lookup(tree t, int k, int shared *d) {
  tree c;
  if (t -> key == k) {
    *d = t -> data;
    return;
  }
  if (k < t -> key)c = t -> left;
  else c = t -> right;
  if (c == NULL) {
    *d = 0;
    return;
  }
  with { rd(c); no_rd(t); } cont;
  aux_lookup(c,k,d);
}
```

Fig. 16. Tree lookup operation.

traverses the tree. The tree traversal works in part because of an informal contract between *aux_lookup* and its caller *lookup*. If *aux_lookup* is called with an immediate read declaration on the root of the subtree, it will successfully perform the lookup on the subtree. As part of its computation the *aux_lookup* is free to declare that its enclosing task will no longer access parts of the tree. The fact that the tree is encapsulated inside a single abstract data type ensures that the conditions on this contract will always be met and that the computation will execute correctly.

Figure 16 contains the new *lookup* operation. Like the *insert* operation, this operation consists of a wrapper routine and a traversal routine. The traversal routine walks down the tree looking for a node with a matching key, incrementally refining its access specification at each level.

Like the hash table implementation of the *index* abstraction, the binary search tree implementation is implicitly synchronized: it completely encapsulates the code required for concurrent execution. Both implementations exploit concurrency between lookup operations and correctly synchronize lookups and insertions. But because the binary search tree implementation uses a more concurrent data structure, it also exploits pipelining concurrency available between both operations. Successive inserts and lookups concurrently access the tree in a pipelined fashion, with the second following the first down the tree as

they refine their access specifications. As soon as one goes left and the other goes right they execute concurrently. Successive lookups, of course, execute concurrently from the start.

7.2 Interchangability

Both the hash table and the tree implementations of the *index* abstract data type completely encapsulate 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 allows the programmer to use the implementations interchangably, picking the alternative best suited for the situation at hand. The client code, of course, remains the same for all implementations.

7.3 Limitations

Hierarchical objects are designed for data structures whose natural access pattern is hierarchical. Such data structures include standard search trees and space subdivision trees such as octrees. But hierarchical data structures are in general unsuitable for expressing parallel computations that traverse graph data structures. The problem is that each graph node may not have a unique parent node. The presence of multiple paths to the same node means that the access pattern cannot be modelled using a strict hierarchy. A similar problem occurs with circular data structures or data structures such as doubly linked lists that may be traversed in multiple directions. It is impossible to model all of the access patterns of such data structures using hierarchical objects.

The reason Jade restricts hierarchical objects to hierarchies is that at each point in a hierarchy there is a single object that controls access to the entire hierarchy below it. Before a task can access any of the objects in the hierarchy, it must go through the object at the top of the hierarchy. This object provides a concrete locus of synchronization for the objects in the hierarchy. Any potential dynamic data dependence between tasks that may access the same object in the hierarchy will show up in the access declarations for all objects above that object in the hierarchy. This property enables the Jade implementation to use the task queue mechanism described in section 9 to synchronize tasks' accesses to objects.

8 Semantics of access specifications

The current Jade implementation uses a conservative approach to exploiting concurrency. It does not execute a task until it knows that the task can legally perform all of its declared accesses. This section defines how the access specifications of tasks interact to generate parallel and serial execution.

Each access specification consists of a set of access declarations. Each access declaration is generated by an access declaration statement and gives the task the right to access a given object. Access declarations also impose serialization constraints on the parallel execution. The nature of the constraint depends on the semantics of the declared accesses. The current Jade implementation supports mutual exclusion constraints and constraints that force tasks to execute in the same order as in the original serial program.

The set of serialization constraints is a lattice. Figure 17 shows the serialization constraint lattice for the current version of Jade. In principle this lattice could expand to include arbitrary constraints. The serialization constraint between two tasks is the least upper bound of the set of serialization constraints induced by the cross product of the two access specifications. The two access declarations must refer to the same shared object to impose a constraint.

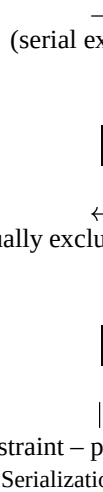


Fig. 17. Serialization constraint lattice

Table 1. Induced serialization constraints

	rd(o)	**wr**(o)	**cm**(o)	**df_rd**(o)	**df_wr**(o)	**df_cm**(o)
rd(o)						
wr(o)						
cm(o)						
df_rd(o)						
df_wr(o)						
df_cm(o)						

Table 1 presents the induced serialization constraints for the current version of Jade. In this table the access declaration from the first task (in the sequential execution order) is in the leftmost column of the table; the access declaration from the second task is on the top row of the table.

There is one somewhat subtle point about the serialization constraints. There may be serialization constraints between a child task and its parent task. Because the child task executes before the remainder of the parent task, in Table 1 the child task's access declaration would appear in the leftmost column while the parent task's access declaration would appear in the top row. If there is an induced serialization constraint between the child and parent tasks, the parent task must suspend until the child task finishes or executes a **with** construct that eliminates the constraint.

For Jade's conservative approach to exploiting concurrency to succeed, the implementation must know ahead of time a conservative approximation of the accesses that each task and its child tasks will perform. The Jade implementation therefore requires that each task's access specification correctly summarize how it (and its child tasks) will access shared objects. From the programmer's perspective, this requirement takes the form of several rules which govern the use of access specifications.

- To access a shared object, a task's access specification must contain an immediate access declaration

Table 2. Enabled accesses

Access declaration	Enabled accesses
rd(o)	read from o
wr(o)	write to o
cm(o)	read from o and write to o

that enables the access. Table 2 summarizes which access declarations enable which accesses.

- If a task's initial access specification contains an access declaration on a given object, its parent task's access specification (at the time the task is created) must also contain one of the following access declarations:
  - A corresponding access declaration on the same object. The declaration must be either a deferred or an immediate access declaration.
  - A corresponding access declaration on the object's parent object. The declaration must be an immediate access declaration.

  Table 3 summarizes the rules.
- A **with** construct can change a deferred access specification to a corresponding immediate access specification, an immediate to a corresponding deferred access specification, or eliminate an access specification. Table 4 summarizes the rules.
- When a task creates an object that has no parent object, its access specification is automatically augmented with deferred read, write and commuting access declarations for that object.
- When a task creates a child object, its access specification does not change.

There are additional restrictions associated with commuting access declarations. To prevent deadlock, the Jade implementation does not allow a task to execute a **with** construct that declares an immediate access to any object when the task's access specification already contains an immediate commuting declaration. The implementation also prevents a task from creating any child tasks while its access specification contains an immediate commuting declaration.

9 Implementation

Given the declarative nature of the Jade programming paradigm, the Jade implementation assumes the responsibility for extracting the concurrency and synchronizing the computation. In effect, there is a general-purpose synchronization algorithm encapsulated inside the Jade implementation. The programmer obliviously reuses this synchronization algorithm every time he or she writes a Jade program. The specific mechanism that the Jade implementation uses to extract the concurrency and synchronize the computation is object queues.

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

Table 3. Access specification rules for the **withonly** construct

If child task declares	Parent task must declare one of
$st(o)$	$st(o)$, **df**_$st(o)$ or $st(po)$
df_$st(o)$	$st(o)$, **df**_$st(o)$ or $st(po)$

In this table po is the parent object of o and $st \in \{\mathbf{rd}, \mathbf{wr}, \mathbf{cm}\}$.

Table 4. Access specification rules for the **with** construct

If a **with** declares	Task must declare one of
$st(o)$	$st(o)$, **df**_$st(o)$ or $st(po)$
df_$st(o)$	$st(o)$, **df**_$st(o)$ or $st(po)$
no_$st(o)$	$st(o)$ or **df**_$st(o)$

In this table po is the parent object of o and $st \in \{\mathbf{rd}, \mathbf{wr}, \mathbf{cm}\}$.

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. 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 deferred 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 can legally execute. The implementation then schedules the task for execution and eventually the task runs.

9.1 A task lifetime

We describe the object queue algorithm in more detail by presenting the object queue manipulations that take place during the lifetime of a task. Here are the events that take place:

- **Task creation:** When a task is created, it inserts an entry into the queue of each object that it declares that it will access. If the task's parent task has an entry in the object queue, the new task inserts its entry just before the parent task's entry. If the parent task has no entry in the object queue, the object must be a child object and the parent must have an entry in the parent object's queue. In this case the implementation inserts the task's entry at the end of the object queue. After the implementation inserts the object queue entries, it sets the task's count to the number of entries that are not enabled.

  The new task's entries may also change its parent entries from enabled to not enabled. In this case the implementation must also increase the parent task's count and suspend the parent task until the child completes or eliminates its corresponding entries.

- **Changing a deferred to an immediate entry:** When a task executes a **with** construct, it may change a deferred entry to an immediate entry. If the new immediate entry is not enabled, the implementation increments the task's count and suspends the task.

- **Changing an immediate to a deferred entry:** A task may also change an immediate entry to a deferred entry. This change has no effect on the task's count.

- **Eliminating an entry:** A task may use a **with** construct to eliminate an object queue entry. This change may make other tasks' entries become enabled; if so, the implementation decrements the task counts appropriately. If one of the tasks' counts becomes zero, the implementation enables the task to execute.

- **Creating a new entry:** If a **with** construct declares an access to an object that is not in the task's access specification, the object must be a child object and the task must declare that it will access its parent object. In this case the implementation inserts the new declaration at the end of the object queue. If the new entry is not enabled, the implementation increments the task's count and suspends the task.

- **Task completion:** When a task completes, the implementation eliminates all of its entries. This elimination may cause other tasks' entries to become enabled; the implementation decrements their counts as appropriate. If one of the tasks' counts becomes zero, the implementation enables the task to execute.

9.2 Extensions for commuting declarations

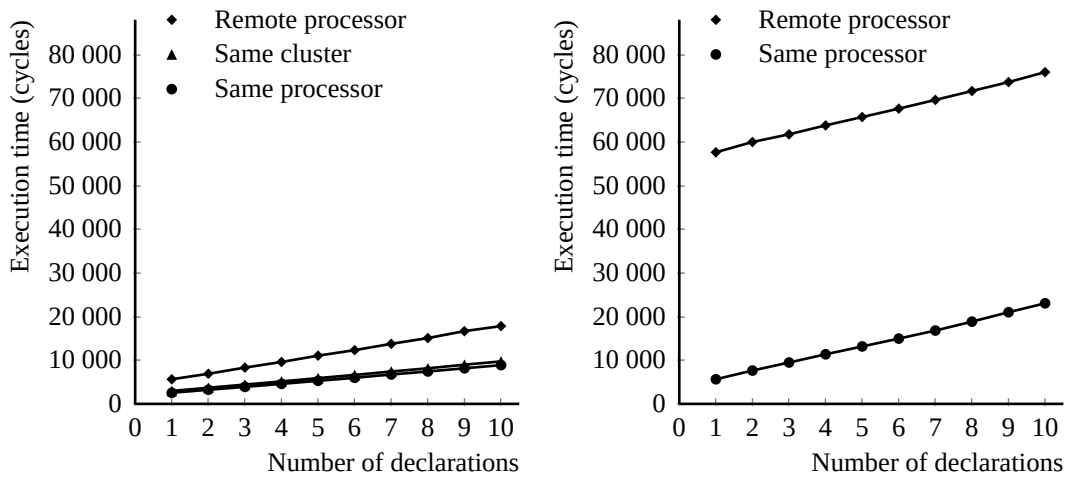
Commuting declarations introduce an extra level of complexity into the synchronization algorithm. There are two kinds of synchronization constraints that apply to commuting declarations: serialization constraints (the implementation must preserve the serial order for commuting accesses relative to other accesses) and exclusion constraints (the commuting accesses must execute in some serial order).

The implementation enforces serialization constraints using the object queue mechanism. Each commuting declaration inserts an entry in the object queue. Immediate commuting entries become enabled when there are only commuting entries before them in the queue.

The implementation enforces exclusion constraints with an exclusion queue. Associated with each object is an exclusion queue. If a task declares an immediate commuting access it will insert an exclusion entry into the corresponding exclusion queue before acquiring the right to access the object. The entry is enabled when its entry becomes the first entry in the exclusion queue. The task itself becomes enabled when all of its object queue and exclusion queue entries are enabled. The task removes its exclusion entry when it completes or uses a **with** construct to eliminate the immediate commuting declaration.

The implementation avoids deadlock by properly sequencing the queue insertions. When a task is created it inserts all of its object queue entries into the object queues according the algorithm in section 9.1. It then waits for all of its object queue entries to become enabled. It then sorts its immediate commuting entries according to an arbitrary order on objects and progressively inserts the exclusion entries into the corresponding exclusion queues. It inserts each exclusion entry only after all previous exclusion entries are enabled.

In effect, the exclusion queues implement a mutual exclusion lock on each object, and the tasks acquire the locks in the sort order. Every task that holds an object lock is either enabled or will become enabled as soon as it acquires the rest of its locks. Because the tasks acquire locks in the sort order, one task must eventually acquire all of its locks and become enabled.



(a) DASH

(b) iPSC/860

Fig. 18. Task overhead in processor cycles.

9.3 Performance results

Figure 18 graphs the dynamic overhead per task as a function of the number of access declaration statements for the current version of Jade. Communication costs are a significant part of the overhead for remote task execution. On the iPSC/860 the software message overhead also contributes substantially to the overall remote task execution overhead. The task creation overheads are determined by how long it takes to perform basic operations such as allocating a task, inserting and deleting object queue entries and communicating a task's data to a remote processor. As the number of processors increases, the only component of these times that increases is the communication time. On our two platforms this time increases because messages must traverse more communication links to get to remote processors on a large machine than on a small machine. But the link traversal time is a very small part of the overall communication time (the communication time is dominated by the time required to launch the communication into the network), so to a first approximation the task overhead does not increase as the number of processors increases. Figure 18 characterizes the overhead in the absence of contention. If concurrently created tasks declare that they will access disjoint sets of objects, there is no contention for object queues and the overhead should be similar to that presented in Fig. 18. If concurrently created tasks declare that they will access the same object, the object queue insertions happen sequentially. The object queue mechanism may therefore cause serialization that would not be present with a different synchronization mechanism. Consider a set of tasks that all read the same object. The object queue will serialize the insertion of their declarations into the object queue. If the tasks have no inherent data dependence constraints, the task queue insertions may artificially limit the parallelism in the computation.

10 Applications

We have implemented a set of benchmark applications in Jade. We attempted to find applications that accurately reflect the scientific and engineering programs that programmers actually run in practice. The final application set consists of four complete scientific applications and one computational kernel. The complete applications are

- Water, which evaluates forces and potentials in a system of water molecules in the liquid state
- String (Harris et al. 1990), which computes a velocity model of the geology between two oilwells
- Search (Browning et al. 1994, Browning et al. 1994), which uses a Monte Carlo technique to simulate the interaction of several electron beams at different energy levels with a variety of solids
- Ocean, which simulates the role of eddy and boundary currents in influencing large-scale ocean movements.

The computational kernel is

- Panel Cholesky, which factors a sparse positive-definite matrix.

The SPLASH benchmark set (Singh et al. 1992) contains variants of the Water, Ocean and Panel Cholesky applications.

10.1 Programming effort

Table 5 presents some of the static characteristics of these programs. We have obtained the original serial version of every application except Search – this was developed in Jade and no other version exists. A comparison of the number of lines of code in the original serial version with the number of lines of code in the parallel version indicates that using Jade usually involves a modest increase in the number of lines of code in the application. The one exception is Ocean. The parallel tasks in this application concurrently write disjoint pieces of a central data structure. To express the concurrency in Jade, the programmer had to explicitly decompose this data structure into multiple shared objects. The resulting changes in the data structure indexing algorithm almost tripled the size of the program. The number of Jade constructs required to parallelize the application (and especially the number of **withonly** constructs) is usually quite small.

The main issue when building Jade applications is determining the correct shared object structure. Once an appropriate shared object structure has been chosen, inserting the task constructs and generating access specifications imposes very little additional programming overhead. The actual programming effort required to develop a correct shared object structure varied from application to application. For all of the applications except Ocean the modifications were confined to small, peripheral sections of the code. The key to the success of these applications was the programmer's ability to preserve the original structure accessing algorithm for the core of the computation. For Ocean, on the other hand, the programmer had to decompose a key array used heavily in the core of the computation. The programmer therefore had to change the basic indexing algorithm for almost all of the program's data and the program almost tripled in size.

Jade programmers typically develop a program in two stages. In the first stage they start with a serial program that performs the desired computation, then apply the appropriate data structure modifications.

Table 5. Static application characteristics

Application	Lines of code serial version	Lines of code Jade version	**withonly** sites	**with** sites	Object creation sites
Water	1219	1471	2	20	7
String	2587	2941	3	37	19
Search	–	716	1	9	3
Panel Cholesky	2047	2484	2	15	18
Ocean	1274	3262	27	28	20

Only when they have debugged the program using standard debugging techniques for serial programs do they insert the Jade constructs required to parallelize the program. The fact that Jade preserves the serial semantics makes this approach productive. Because none of the Jade constructs change the semantics, programmers can parallelize the program with no fear of introducing undetected programming errors that make the program generate an incorrect result. In our experience the fact that Jade enables programmers to use this development strategy makes it significantly easier to develop parallel programs. The use of Jade also generated clean final source programs with good modularity. All of the Jade constructs are appropriately encapsulated, and the basic structure of the serial version is preserved in the parallel version.

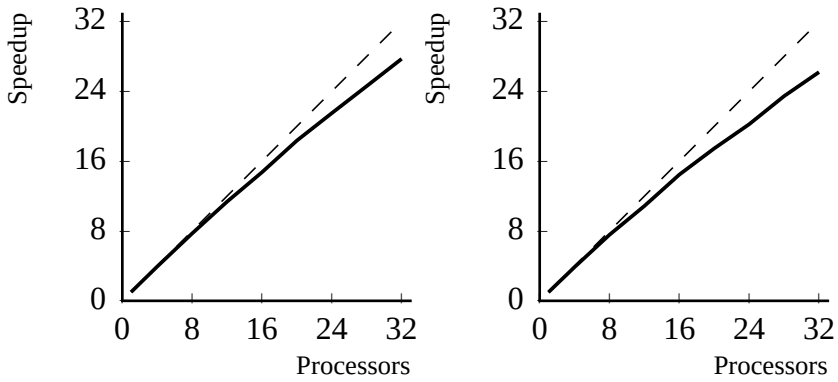
Water, String and Search exhibit similar concurrency patterns and we were able to reuse the Jade access declaration code developed for one application in the others. But even though the concurrency patterns were similar, the data structures that the applications used were different. The reuse therefore took place by copying Jade code from one application into another rather than by reusing an entire abstract data type. We believe, however, that this kind of reuse is not an important issue for Jade programs. Given the ease of generating the task constructs and access specifications given an appropriate shared object structure, the ability to reuse code in this way is a relatively unimportant aspect of Jade.

The most important kind of reuse is the implicit reuse of the general-purpose synchronization and concurrency management algorithms in the Jade implementation. These algorithms support Jade's implicitly parallel approach and enable the implementation to provide the programming advantages that Jade offers.

10.2 Performance

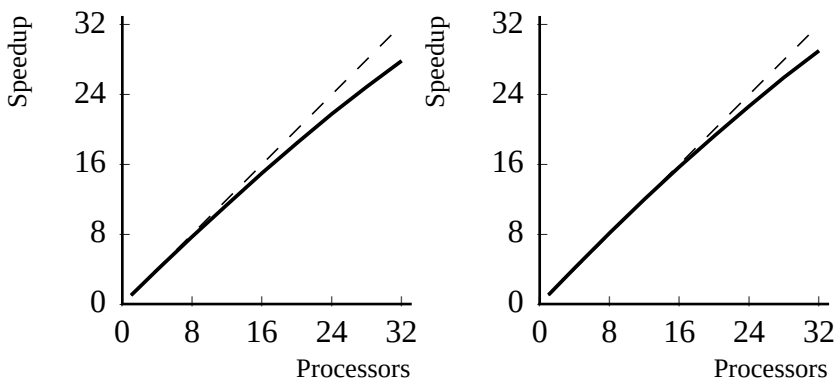
Figures 19–23 plot the speedup curves for the applications running on both DASH and the iPSC/860. We generated these curves by first developing a version of the compiler that strips all of the Jade constructs out of a Jade program, yielding a serial program that executes with no Jade overhead. We then measured the execution time of this serial program running on one processor, and the execution times of the Jade program running on multiple processors. The speedup curve for a given application plots the execution time of the serial program with no Jade overhead divided by the execution time of the Jade program as a function of the number of processors executing the Jade program. The speedup curves therefore present the performance increase over an efficient serial program that a programmer could expect to achieve using Jade, and give a good indication of how well the programs utilize the parallel machine.

The Water, String and Search applications all speed up almost linearly to 32 processors on both platforms. All of the applications have a large task size and the amortized Jade overhead is neglible. For these applications the use of Jade enables efficient execution in addition to all of the software development advantages. Because the applications scale so well, there is no potential performance improvement to be



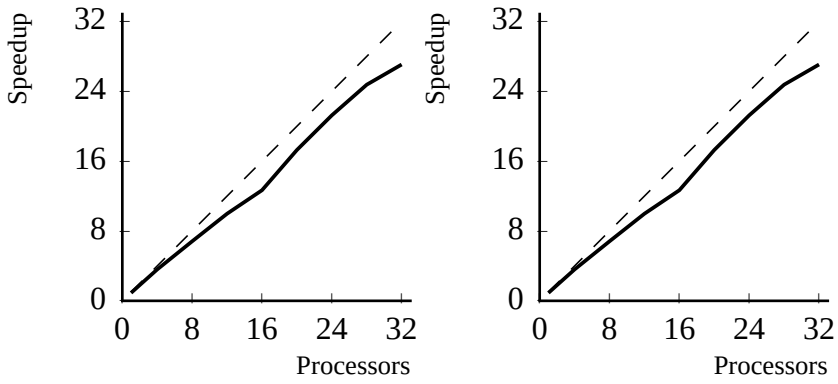
(a) DASH (b) iPSC/860

Fig. 19. Water.



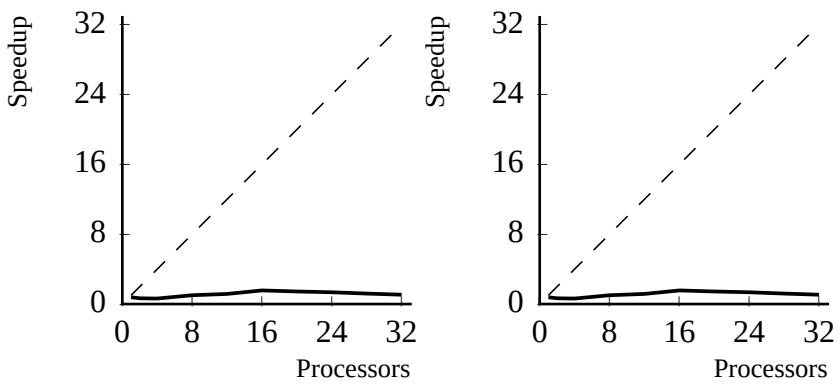
(a) DASH (b) iPSC/860

Fig. 20. String.



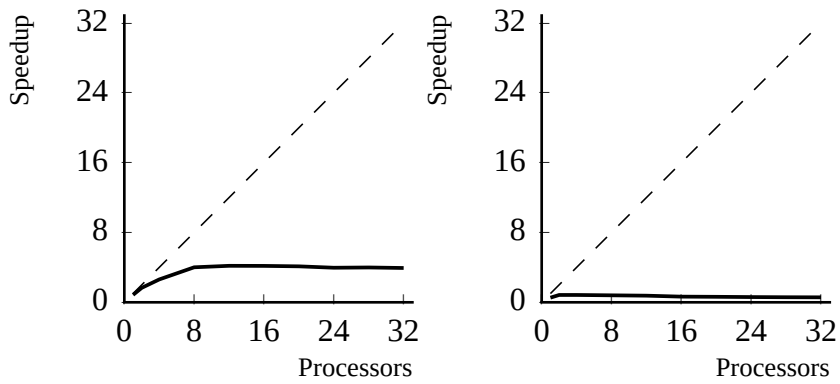
(a) DASH (b) iPSC/860

Fig. 21. Search.



(a) DASH (b) iPSC/860

Fig. 22. Ocean.



(a) DASH (b) iPSC/860

Fig. 23. Panel Cholesky.

gained by using an explicitly parallel programming system. For these applications Jade was a complete success: it allowed the programmer to develop clean, portable parallel applications that perform very well on both shared memory and message passsing machines.

The Ocean and Panel Cholesky applications do not scale as well. On DASH they exhibit relatively poor speedup and on the iPSC/860 they barely speed up at all. For these applications the poor performance is caused by serialized task management overhead. Neither application has a task size large enough to profitably amortize the task management overhead. The iPSC/860 does not support the fine-grain communication required to efficiently support task management. The iPSC/860 performance is therefore substantially worse than the performance on DASH, which provides much better support for fine-grain communication. In both of these applications the primary performance problem is the communication and computation overhead associated with managing tasks, not communication overhead caused by an inappropriate data decomposition.

It is possible for a programmer using an explicitly parallel language to get much better performance for Ocean running on DASH (Chandra et al. 1993). The programmer can develop an application-specific synchronization algorithm that has less overhead than the algorithm embedded inside the Jade implementation. It would be possible, however, to develop a Jade implementation that used static analysis to generate optimized parallel code that eliminated most if not all of the dynamic task management overhead.

For Panel Cholesky it is possible to increase the performance on DASH using sophisticated scheduling and synchronization algorithms, although an inherent lack of concurrency in the application limits the performance (Rothberg 1993). Given the dynamic nature of this computation, a maximally efficient implementation seems to require the knowledge, insight and programming effort that only a skilled programmer using an explicitly parallel language can provide.

In general, these results show that Jade delivers good performance and is an excellent overall choice for applications with a task size large enough to successfully amortize the task management overhead.

For programmers to achieve optimal performance on applications with a smaller grain size, they may need to invest the programming effort required to develop a highly tuned explicitly parallel version of the application.

11 Conclusion

In this paper 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. 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 the anonymous reviewers for their invaluable comments and recommendations.

References

Agha, G. (1986) *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA.

America, P. (1987) POOL-T: A parallel object-oriented language. In *Object Oriented Concurrent Programming* (eds A. Yonezawa and M. Tokoro), pp. 199–220. MIT Press, Cambridge, MA.

Arvind and Thomas, R. (1981) I-structures: An efficient data type for functional languages. Technical Report MIT/LCS/TM-210, MIT.

Brinch-Hansen, P. (1975) The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering* **1**(2),199–207.

Brinch-Hansen, P. (1977) *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, NJ.

Browning, R., Li, T., Chui, B. et al.(1994) Empirical forms for the electron/atom elastic scattering cross sections from 0.1–30 keV. *Journal of Applied Physics*, **76**(4), 2016–22.

Browning, R., Li, T., Chui, B. et al.(1995) Low-energy electron/atom elastic scattering cross sections for 0.1–30 keV. *Scanning* **17**(4), 250–3.

Campbell, R. and Kolstad, R. (1980) An overview of Path Pascal's design. *ACM SIGPLAN Notices*, **15**(9), 13–14.

Chandra, R., Gupta, A. and Hennessy, J. (1993) 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.

Chien, A., Reddy, U., Plevyak, J. and Dolby, J. (1996) ICC++ – A C++ dialect for high performance parallel computing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, March.

Chow, J. and Harrison, W. III (1992) Compile-time analysis of parallel programs that share memory. In *Proceedings of the Nineteenth Annual ACM Symposium on the Principles of Programming Languages*, January, pp. 130–41.

Feeley, M. and Levy, H. (1992) Distributed shared memory with versioned objects. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 247–262.

Gehani, N.H. (1993) Capsules: A shared memory access mechanism for concurrent C/C++. *IEEE Transactions on Parallel and Distributed Systems* **4**(7), 795–811.

Halstead, R. Jr (1985) Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* **7**(4), 501–38.

Halstead, R., Jr (1986) An assessment of Multilisp: lessons from experience. *International Journal of Parallel Programming* **15**(6), 459–501.

Harris, J., Lazaratos, S. and Michelena, R. (1990) Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pp. 82–5.

Hoare, C.A.R. (1974) Monitors: an operating system concept. *Communications of the ACM*, **17**(10), 549–57.

Kolstad, R. and Campbell, R. (1980) Path Pascal user manual. *ACM SIGPLAN Notices*, **15**(9), 15–24 .

Lambert Butler, W. and Redell, D.D. (1980) Experience with processes and monitors in Mesa. *Communications of the ACM* **23**(2), 105–17.

Liskov, B. (1988) Distributed programming in Argus. *Communications of the ACM* **31**(3), 300–12.

Lusk, E., Overbeek, R., Boyle, J. et al.(1987) *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, New York.

Mitchell, J.G., Maybury, W. and Sweet, R. (1979) Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April.

Mohr, E., Kranz, D. and Halstead, R. (1990) Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June, pp. 185–97.

Nikhil, R. (1990) Id version 90.0 reference manual. Technical Report 284-1, Computation Structures Group, MIT Laboratory for Computer Science, September.

Nikhil, R. and Pingali, K. (1989) I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems* **11**(4), 598–632.

Pierce, P. (1988) The nx/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, January (ed. Geoffrey Fox), pp. 384–90.

Rothberg, E. (1993) Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization. PhD thesis, Stanford, CA, January.

Scales, D. and Lam, M.S. (1994) The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November.

Sedgewick, R. (1988) *Algorithms*. Addison-Wesley, Reading, MA.

Singh, J. (1993) Parallel hierarchical *N*-body methods and their implications for multiprocessors. PhD thesis, Stanford University, February.

Singh, J., Weber, W. and Gupta, A. (1992) SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, **20**(1), 5–44.

Sunderam, V. (1990) PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, **2**(4), 315–39.

Tomlinson, C. and Singh, V. (1989) Inheritance and synchronization with enabled-sets. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October, pp. 103–12.

Yonezawa, A., Briot, J.-P. and Shibayama, E. (1986) Object oriented concurrent programming in ABCL/1. In *Proceedings of the OOPSLA-86 Conference*, Portland, OR, September, pp. 258–68.