An Integrated Synchronization and Consistency Protocol for the Implementation of a High-Level Parallel Programming Language

Martin C. Rinard
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
martin@cs.ucsb.edu

Abstract

This paper presents the integrated synchronization and consistency protocol used in the implementation of Jade, an implicitly parallel language for coarse-grain parallel computation. The consistency protocol tags each replica of shared data with a version number. The synchronization algorithm computes the correct version numbers of the replicas of shared data that the computation will access. Because the integrated protocol piggybacks the version number information on the synchronization messages, it eliminates the excess message traffic characteristic of standard update and invalidate protocols. This paper characterizes the performance impact of the consistency protocol by presenting experimental results for several Jade applications running on the iPSC/860 under several different Jade implementations. We believe that implementors of other parallel software systems will be able to improve the efficiency of their systems by using similar integrated protocols.

1 Introduction

The traditional shared memory programming model provides separate abstractions for communication and synchronization. The parallel threads of control communicate implicitly by reading and writing the shared memory and synchronize explicitly using synchronization constructs such as locks and barriers. Implementing the abstraction of shared mutable memory on a modern distributed memory hardware platform is a challenging task. System builders must implement a message passing protocol to coordinate multiple accesses to multiple physical memories. Efficiency concerns complicate the implementation process. The long latencies and potential message passing overhead associated with accessing a remote memory have motivated implementors to build systems that replicate data for fast local access. But any time a system replicates mutable data, it must solve the consistency problem: it must prevent tasks from accessing out of date replicas of shared data. The standard way to solve the consistency problem is to use an invalidate or update consistency protocol. These protocols send messages that eliminate or overwrite any out of date replicas whenever a task writes shared data.

The end result [2, 8] is usually a system that contains two distinct protocols: a synchronization protocol to implement the synchronization abstractions, and a consistency protocol to manage the replication. But because most programs synchronize their accesses to shared data, separating the protocols may generate more message traffic than that required to correctly execute the program. A locked update to a shared object, for example, may generate multiple interactions with remote processors: one to acquire the lock, one to fetch the shared object, and, when other processors write the object, interactions to invalidate or update the out of date replica of the object. A more efficient system would combine the messages and generate fewer interactions.

In this paper we present the integrated synchronization and consistency protocol used in the implementation of Jade, a portable, implicitly parallel programming language designed for exploiting task-level concurrency[11, 13]. Jade programmers start with a program written in a standard serial, imperative language, then use Jade constructs to declaratively specify how parts of the program access data. The Jade implementation analyzes this data usage information to automatically execute the program in parallel.

The Jade implementation exploits both its knowledge of how the computation will access data and its control of the synchronization algorithm to merge the consistency protocol into the synchronization protocol. Instead of using an update or invalidate protocol to eliminate out of date replicas, the Jade implementation uses a consistency protocol based on version numbers. All replicas are tagged with a version number that counts the number of times the program wrote the data to generate that version. Every time the program accesses shared data, the synchronization algorithm automatically calculates the number of the correct version to access. The consistency algorithm uses the version numbers to ensure that the program always accesses the correct version of each piece of shared data. The implementation eliminates excess excess message traffic by piggybacking all version number information on the messages that implement the synchronization protocol. The end result is a single efficient protocol that generates fewer total messages than the combination of an isolated synchronization protocol and an isolated consistency protocol.

Stable implementations of Jade for shared memory machines, message passing machines and heterogeneous networks of workstations have existed for over two years. Jade programs port without modification between all platforms. In this paper we present experimental results for several Jade applications running on the iPSC/860[1]. These experimental results characterize the performance impact of the Jade version number protocol relative to an invalidate or update protocol. These experiments reveal that there is a wide variance in the performance impact of the update protocol relative to the other two protocols. For some applications the update protocol degrades the performance by sending many superfluous update messages to processors that never access the updated version. For other applications the update protocol improves the performance by distributing a new version of shared data with a single broadcast message rather than multiple point to point messages. Motivated by our applications experience, we developed a hybrid communication and consistency protocol for Jade that realizes the broadcast performance improvements while avoiding any performance degradation caused by excessive communication.

This paper makes the following contributions:

- It presents an integrated synchronization and communication protocol that generates fewer messages than the standard combination of an isolated synchronization and isolated communication protocol.
- It presents experimental results that quantify the performance difference between standard update and invalidate protocols and the integrated protocol presented in the paper. We obtained these performance results by instrumenting a real implementation of Jade running real applications on real hardware: there are no simulation results and no artificial benchmark programs.
- It shows how to integrate messages from other protocols into the base protocol when these messages improve the performance. In particular, it describes a hybrid communication protocol that combines the advantages of the base protocol and an update protocol, and presents experimental results that quantify the performance improvement that results from the use of the hybrid protocol.

The rest of the paper is structured as follows. In Section 2 we provide an overview of the Jade language; Section 3 describes some of the more advanced features of Jade. In Section 4 we describe the Jade synchronization and consistency protocols in detail. In Section 5 we describe the different implementations used in the experimental evaluation. Section 6 describes the applications, while Section 7 presents the experimental results. Section 8 presents a hybrid protocol that improves the performance of some applications by selectively broadcasting widely accessed data. We review related work in Section 9, then conclude in Section 10.

2 Language Overview

Because of Jade's sequential semantics, it is possible to implement Jade as an extension to a sequential programming language. This approach allows programmers to learn how to use Jade quickly, and preserves much of the language-specific investment in programmer training and software tools. We have currently implemented Jade as an extension to C.

2.1 Object Model

Jade supports the abstraction of a single shared memory that all tasks can access; each piece of data allocated (statically or dynamically) in this memory is called a *shared object*. Pointers to shared objects are identified in a Jade program using the `shared` type qualifier. For example:

```
double shared A[10];
double shared *B;
```

The first declaration defines a statically allocated shared vector of `doubles`, while the second declaration defines a reference (pointer) to a dynamically allocated shared vector of `doubles`. Programmers use the `create_object` construct to dynamically allocate shared objects; this construct takes as parameters the type and number of allocated items in the shared object. Given the declaration of `B` presented above, the programmer could dynamically allocate a shared vector of 100 `doubles` as follows:

```
B = create_object(double, 100);
```

2.2 Basic Construct

Jade programmers use the `withonly` construct to identify a task and to specify how that task will access data. Here is the general syntactic form of the construct:

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

The `task body` section contains the serial code executed when the task runs. The `parameters` section declares a list of variables from the enclosing environment that the task body may access. The `access declaration` section summarizes how the task body will access the shared objects.

The access declaration is an arbitrary piece of code containing *access specification statements*. Each such statement declares how the task will access a given shared object. For example, the `rd` (read) statement declares that the task may read the given object, while the `wr` (write) statement declares that the task may write the given object. Each access specification statement takes one parameter: a reference to the object that the task may access. The task's access declaration section may contain dynamically resolved variable references and control flow constructs such as conditionals, loops and function calls. The programmer may therefore use information available only at run time when generating a task's access specification. Consequently, Jade programs can exploit dynamic concurrency that can only be determined at run time.

Since the parallelization and data movement is based on access specifications, it is important that the access specifications be accurate. The Jade implementation therefore dynamically checks each task's accesses to shared objects. If a task attempts to perform an undeclared access, the implementation will generate a run-time error. The Jade type system has a mechanism that allows each task to perform the dynamic access check once for each object. Because the overhead is then amortized over many object accesses, in practice the total checking overhead is negligible.

Each shared object represents a unit of synchronization. If one task writes part of a shared object and another task accesses that object, the implementation executes the tasks serially. Tasks that access different shared objects or read the same objects can execute concurrently. The programmer must therefore allocate objects at a fine enough granularity to expose the desired amount of concurrency.

In a message-passing environment, each shared object is also a unit of communication. If a task will access an object, the implementation will either move (on a write access) or copy (on a read access) the entire object to the machine on which the task will execute. If the task only accesses a small part of the object, the communication overhead may negate any performance gains from concurrency. The programmer must therefore allocate objects at a fine enough granularity to minimize superfluous communication.

3 Advanced Language Features

We have, so far, described the basic programming model in Jade. Here we describe briefly some of the advanced constructs of the Jade language that allow a Jade programmer to achieve more sophisticated parallel behavior.

3.1 Hierarchical Concurrency

Programmers may create hierarchical forms of concurrency in a Jade program by dynamically nesting `withonly` constructs. The task body of a `withonly` construct may execute any number of `withonly` constructs, in a fully recursive manner. The access specification of a task that hierarchically creates child tasks must declare both its own accesses and the accesses performed by all of its child tasks.

3.2 More Precise Access Specifications

As described so far, Jade constructs support only a limited model of parallel computation in which all synchronization takes place at task boundaries. Two tasks may either execute concurrently (if none of their accesses conflict) or sequentially (if any of their accesses conflict). The `withonly` construct by itself allows no partial overlap in the executions of tasks with data dependence conflicts. This limitation makes it impossible to exploit pipelining concurrency available between tasks that access the same data.

Jade relaxes this restricted model by allowing the programmer to provide more precise information about when a task actually accesses data. Jade provides this functionality with an additional construct, `with`, and the additional access specification statements `df_rd`, `df_wr`, `no_rd`, and `no_wr`. The general syntactic form of the `with` construct is:

```
with { access declaration} cont;
```

As in the `withonly` construct, the `access declaration` section is an arbitrary piece of code containing access declaration statements. These statements update the task's access specification, allowing the specification to reflect more precisely how the remainder (or continuation, as the `cont` keyword suggests) of the task will access shared objects.

The `df_rd` and `df_wr` statements declare a *deferred* access to the shared object. That is, they specify that the task may eventually read or write the object, but that it will not do so immediately. Because the task cannot immediately access the object, it can execute concurrently with earlier tasks that do access the object. When the task reaches the point where it will access the object, it must execute a `with` construct that uses the `rd` or `wr` access declaration statements to convert the deferred declaration to an *immediate* declaration. This immediate declaration then gives the task the right to access the object. In order to preserve the serial semantics, the Jade implementation may suspend the task at the `with` construct until preceding tasks with conflicting accesses to the object have completed. It is also possible to use the `df_rd` and `df_wr` statements inside a `with` construct to convert immediate declarations to deferred declarations. This may allow the task and any future child tasks that declare immediate accesses to the objects in question to execute concurrently.

The `no_rd` (no future read) and `no_wr` (no future write) statements allow the programmer to declare that a task has finished its access to a given object and will no longer read or write the object. This declaration dynamically reduces a task's access specification, which may potentially eliminate a conflict between the task executing the `with` and later tasks. In this case the later tasks may execute as soon as the `with` executes rather than waiting until the first task completes.

4 The Synchronization and Consistency Protocols

We now turn our attention to the mechanisms that the Jade implementation uses to synchronize the computation, generate the communication and ensure that each task accesses the correct version of each object. We first describe the synchronization protocol, then describe how the implementation piggybacks the information required by the consistency protocol onto the synchronization messages.

4.1 The Synchronization Protocol

The implementation extracts the concurrency and synchronizes the computation using *object queues.* There is a queue associated with each object that synchronizes tasks' accesses to that object. When a task creates an object, the object's queue starts out with one entry that declares a deferred read and write access to the object. When a child task declares that it will access an object, the implementation inserts an entry into the object's queue, putting the new entry just before the entry of the task's parent. The entries therefore appear in the queue in the same order as the corresponding tasks would execute in a sequential execution of the program. If a task declares an immediate write access, its entry becomes enabled when it reaches the front of the queue. Immediate read entries become enabled when there are no write entries before them in the queue. Deferred entries are always enabled. When all of a task's entries become enabled it can legally execute.

We next describe the protocol used to implement the object queue synchronization algorithm in a message passing environment. At any given time each object queue resides completely on one processor; this processor is called the owner of the object queue. When a task first declares that it will access an object, the implementation generates messages that locate the object queue and transfer the object queue as a unit to the processor that will insert the task's entry into the object queue. When the queue arrives, the processor performs the insertion.

The implementation locates the object queue using a forwarding pointer algorithm. For each object queue the processor maintains a forwarding pointer to the most recent processor known to have requested the object queue. The queue location algorithm follows forwarding pointers until it finds a processor that either has the queue or has requested but not yet received the queue. That processor forwards the queue to the original requesting processor after it has received the queue and performed any pending local operations. The queue only migrates when a new entry is inserted. As described below, other operations take place remotely when the message specifying the operation arrives at the owner of the queue. This design is optimized for situations in which one processor inserts almost all of the entries into the task queue. In this case the queue would migrate to the processor performing most of the insertions and most of the insertions would not incur the overhead of a remote message send. In practice almost all insertions are performed on object queues that are available locally. See [11] for more details.

At any given time each task resides completely on one processor; this processor is called the owner of the task. When an entry becomes enabled the implementation sends an enable message to the task's owner informing it that the entry is enabled. The object queue also sends a similar message to the owner when the task has a deferred entry in the object queue. The implementation must send this message because the owner does not inform the object queue when the task uses a `with` construct to convert the deferred declaration to an immediate declaration.

When a task uses a `with` construct to eliminate an access declaration, the implementation sends a deletion message to the processor that owns the queue informing it of the deletion. When the processor receives this message, it removes the corresponding entry from the queue, potentially enabling other queue entries.

A similar process takes place when a task finishes its computation. The implementation sends a message containing the completed task's access specification to the processor on which the task was created. When the message arrives, the implementation processes each of the task's access declarations. For each access declaration the implementation sends a deletion message to the processor that owns the corresponding object queue informing it that the task has finished. When the processor receives this message it removes the corresponding entry from the queue, potentially enabling other queue entries.

In the protocol described above, the implementation does not actually send a message to perform an object queue or task operation if the queue or task is available locally—it just performs the operation directly on the local queue or task data structure. This optimization eliminates the message passing overhead associated with sending a message from one processor back to itself. In practice almost all object queue deletions and insertions are performed locally and generate no communication.

4.2 Correctness of the Synchronization Algorithm

In this section we discuss why the object queue mechanism correctly synchronizes the computation. There are two aspects to the correctness of a synchronization mechanism for Jade programs: 1) at every point of the computation there is at least one task that can execute (i.e. the synchronization mechanism is deadlock free) and 2) the synchronization algorithm preserves the relative order of reads and writes to the same object.

Tasks' entries appear in the object queues in the same order that the tasks would execute in a serial program (the serial entry order property). This is true of the original state of an object queue, which contains one entry from the task that created the object. Subsequent object queue operations preserve the serial entry order property. If a task completes or eliminates an access declaration, the implementation removes its entries from the object queue. The new sequence of entries is then a subsequence of the original sequence of entries, and the property holds.

The only other queue operation is the insertion of a new entry. The new entry appears before the entry of its parent task. In the original sequence all entries which appeared before the parent task entry belonged to tasks which executed before the parent task. The child task's entry appears after all of these entries, which is correct because in the sequential execution the child task executes after all of these tasks. Similarly, the child task's entry appears before all of the entries after its parent task's entry, and the child task should execute before all these tasks. Finally, the child task's entry appears before its parent task's entry. This is again correct because the child task should execute before the remainder of its parent task.

The serial entry order property implies the deadlock freeness of the object queue mechanism. The sequential execution order is a total order on the tasks, so at every point in the computation there is at least one task such that every one of that task's entries are at the front of their object queues. This task can execute.

We next establish that all parallel executions preserve the relative order of reads and writes to the same object. More precisely, we establish that if a write occurs before a read to a given object in the sequential execution, the write will occur before the read in all parallel executions and that if a write occurs after a read in the sequential execution, it will occur after the read in all parallel executions. We present an argument for the write before read case; a similar argument holds for the read before write case. See [12] for a more formal treatment of this argument.

Consider a write that happens before a read in a sequential execution. There are several cases: 1) the same task performs both the read and the write, 2) the task that performs the write is an ancestor of the task that performs the read, 3) the task that performs the read is an ancestor of the task that performs the write, and 4) two different tasks perform the read and the write, and neither is an ancestor of the other. We show that in all parallel executions the write occurs before the read.

In case 1 the operations of the task always execute in the same order as in the serial execution, so the write occurs before the read. In case 2 the parent task performs the write before it creates the reading task. In case 3 the first task on the ancestor chain of tasks from the parent task to the writing task is created before the read is performed. This child task and every task in the ancestor chain inserts a write entry into the object queue. These entries occur before the parent task's entry. When the parent task performs the read, it must have an enabled immediate read entry in the object queue. But unless the write has already been performed, there is and will continue to be at least one write entry from the ancestor chain in the queue until the write is performed. Because these write entries appear in the queue before the parent's entry, they prevent the parent's immediate read entry from becoming enabled. The parent task cannot perform the read until the write has been performed and the write entries removed from the queue.

In case 4 the writing task and the reading task share at least one common ancestor task. Find the least common ancestor task (the ancestor task with no child task that is also a common ancestor task). This ancestor task has two child tasks, one of which (the first task) either performs the write or is an ancestor of the writing task. The second task either performs the read or is an ancestor of the reading task. The first task inserts a write entry into the queue and is created before the second task, which inserts a read entry into the queue. Because the two tasks have the same parent the write entry appears in the queue before the read entry.

All of the ancestors of the writing task between the writing task and the first task must insert write entries into the queue. By the time the reading task is created and inserts its entry into the queue, either the write will have occurred, or there is and will continue to be a write entry before it in the queue until the write occurs. This write entry will prevent the read task's entry from becoming enabled, which in turn will prevent the read task from performing the read until the write task performs the write and eliminates the write entry.

4.3 The Consistency Protocol

We next present the consistency protocol. The Jade implementation tags each replica of an object with a version number. The version number counts the number of times the program wrote the object before it generated that version of the object. Whenever a task completes a write (either by executing a `with` construct that eliminates a write access declaration or by finishing its execution), the implementation increments the version number of the updated local replica of the object.

The consistency protocol uses version numbers to ensure that each task accesses the correct version of each object. The enable messages sent to inform tasks that their object queue entries have become enabled contain the version number and owner of the version of the object that the task should access. Before the implementation executes a task, it checks the objects that the task will access. If there is no locally available replica of an object or the locally available replica has an out of date version number, the implementation sends an object request message to the current owner. When the owner receives this message it sends a replica of its current version to the accessing processor. The task can execute when all of the requested objects arrive.

To calculate the correct version number and owner of the version of the object that each should access, the implementation stores the current version number and current owner of each object in the object queue. Every time the implementation deletes a write entry from the front of the queue, it increments the queue's current version number and records the number of the processor that generated that version as the current owner. Because all of the enable messages sent to inform tasks that their queue entries have become enabled originate at the object queue, the implementation can extract the current version number and owner from the object queue and insert it into the enable messages with almost no overhead.

This scheme allows the implementation to piggyback all of the consistency information on the synchronization messages. There are no extra invalidate or update messages required to keep the memory consistent, although the implementation can send such messages if they improve the performance of the application. By integrating consistency information into the synchronization protocol, the Jade implementation eliminates redundant communication associated with the separation of synchronization and consistency protocols.

4.4 Deallocating Obsolete Primary Replicas

To deliver good memory utilization, the Jade implementation must be able to deallocate every replica of an object except the current primary replica stored at the current owner (which it must preserve to correctly execute the program). Jade uses a lazy deallocation algorithm for non-primary replicas of objects. Each processor has a target amount of memory dedicated to storing objects. When the amount of memory dedicated to object storage exceeds this target, the implementation deallocates that processor's non-primary replicas of objects (using a least recently accessed policy) until the amount of memory dedicated to objects drops below the target.

The implementation uses an eager deallocation algorithm for out of date primary replicas of objects. The current primary replica of a given task executes on the owner, there is no deallocation problem—the object becomes out of date when a task writes the object. If the task simply generates the new version by updating the existing primary replica. If the task executes on a processor with no replica of the current version, the implementation will send an object request message to the owner. When the owner receives this message, it forwards its current version to the processor that will execute the writing task. The owner then deallocates its replica, which will become out of date when the task generates the new version.

If the task executes on a processor that has a non-primary replica of the current version, it does not need to communicate with the current owner—it can simply generate the new version by updating its replica. But to enable the current owner to deallocate its replica (which will become out of date when the task executes), the implementation sends a deallocate message to the current owner. When the current owner receives the deallocate message it deallocates its replica.

5 The Three Jade Implementations

To experimentally evaluate the performance impact of using the version number consistency protocol instead of an update or invalidate protocol, we implemented three versions of the Jade implementation. Each uses a different consistency protocol.

5.1 Version

The Version implementation uses the object queue synchronization protocol and the version number consistency protocol. It finds the owner of each fetched remote object using the owner information maintained in the object queue.

5.2 Invalidate

The Invalidate implementation uses the same synchronization protocol as the Version implementation, but obtains the effect of a standard invalidate protocol by sending invalidate and invalidate acknowledge messages. When the current owner of an object either executes a task that will write the object or receives a deallocate message informing it that another processor will execute a task that will write the object, it sends invalidate messages to every processor with a replica of the object except the writing processor. When these processors receive the invalidate message they deallocate their version of the object, then send invalidate acknowledge messages back to the writing processor. The writing processor never blocks waiting for invalidate acknowledge messages to arrive. The performance impact comes from the overhead of processing the invalidate and invalidate acknowledge messages. In theory, an invalidate protocol may also improve the memory utilization. When a processor receives an invalidate message it deallocates the memory associated with the invalidated version of the object. The implementation can then use the memory for other objects. Without the invalidate message the processor may use more memory or generate object retransmissions by deallocating replicas of objects that it will access in the future.

5.3 Update

The Update implementation uses the same synchronization protocol as the Version implementation, but obtains the effect of a standard update protocol by sending update and update acknowledge messages. Every time a task finishes a write and generates a new version of the object, it sends an update message containing the new version of the object to every processor that ever accessed the object. If all processors accessed the object, the implementation sends the new version in a single broadcast message instead of multiple point to point messages. When each processor receives the updated object it sends an update acknowledge message back to the processor that generated the new version of the object. The writing processor never blocks waiting for update acknowledge messages to arrive.

An update protocol has the potential to either improve or degrade the overall performance. The protocol may waste communication bandwidth by sending new versions of objects to processors that never actually access the new versions. The update acknowledge messages also impose an additional message processing load on the system. On the positive side, eagerly updating versions of objects can eliminate object request messages and the latency associated with waiting for requested versions of objects to arrive. It is also much more efficient to broadcast a single message containing the new version of an object accessed by all processors than to send a separate point to point message to every processor.

6 The Jade Applications

We evaluate the Jade implementations by measuring the performance of several Jade applications running under each of the protocols. In any application-driven study the application set has a crucial impact on the final conclusions. 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 three complete 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 [6], which computes a velocity model of the geology between two oil wells, and 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 [16] contains variants of the Water, Ocean and Panel Cholesky applications. Figure 1 contains some basic application characteristics.

Application	Lines of Code	Serial Execution Time with no Jade Overhead in Seconds	Mean Task Size on 32 Processors in Seconds
Water	1471	2406.72	4.75
String	2941	19629.42	74.30
Panel Cholesky	2047	28.53	.0020
Ocean	3262	60.99	.0033

Table 1: Application Characteristics

In any application-based experimental evaluation the input data sets can be as important as the applications themselves. In each case we attempted to use realistic data sets that accurately reflected the way the applications would be used in practice. The data set for Water consists of 1728 molecules distributed randomly in a rectangular volume. It executes 8 iterations, with two parallel phases per iteration. These performance numbers omit an initial I/O and compution phase. In practice the computation would run for many iterations and the amortized cost of the initial phase would be negligible. The data set for String is from an oil field in West Texas and discretizes the 185 foot by 450 foot velocity image at a 1 foot by 1 foot resolution. It executes six iterations, with one parallel phase per iteration. The performance numbers are for the entire computation, including initial and final I/O.

The data set for Ocean is a square 192 by 192 grid. For Panel Cholesky the timing runs factor the BCSSTK15 matrix from the Harwell-Boeing sparse matrix benchmark set[4]. The performance numbers only measure the actual numerical factorization, omitting initial I/O and a symbolic factorization phase. In practice the overhead of the initial symbolic factorization would be amortized over many factorizations of matrices with identical structure.

7 Experimental Results

We next present experimental results that characterize the performance of the application under each of the Jade implementations. We first describe how each application accesses data, then relate the data access behavior to the observed communication and execution time differences. All experiments are run on the iPSC/860 [1], a message passing machine consisting of i860 processing nodes interconnected by a hypercube network. The Jade implementation uses the NX/2 message passing library as its communication layer.

7.1 Ocean

The computationally intensive section of Ocean uses an iterative method to solve a set of discretized spatial partial differential equations. Conceptually, it stores the state of the system in a two dimensional array. On every iteration the application recomputes each element of the array using a standard five-point stencil interpolation algorithm.

To express the computation in Jade, the programmer decomposed the array into a set of interior blocks and boundary blocks. Each block consists of a set of columns. The size of the interior blocks determines the granularity of the computation and is adjusted to the number of processors executing the application. There is one boundary block two columns wide between every two adjacent interior blocks.

At every iteration the application generates a set of tasks to compute the new array values in parallel. There is one task per interior block; that task updates all of the elements in the interior block and one column of elements in each of the border blocks. Almost all of the communication therefore takes place when border blocks move between adjacent processors in response to each task's request to update the block.

We next turn to the experimental results. Figure 1 presents, for each of the three Jade implementations, the total number of messages sent during the Ocean computation as a function of the number of processors executing the computation. The Version and Invalidate implementations send an almost identical number of messages, which means that the Invalidate implementation sends almost no invalidate messages. Because each task updates all the border objects that it accesses, there is never more than one outstanding replica of the current version of each border block, and the border block updates generate no invalidation messages.

The Update implementation generates significantly more messages than the Version and Invalidate implementations. The extra messages are deallocate and update acknowledge messages. The Update implementation sends the new version of each border block to the adjacent processor every time the current processor finishes its write; the adjacent processor responds by sending an update acknowledge message back to the current processor. The latest version of the border block is therefore available at the adjacent processor when it starts its update. But even though the adjacent processor has the latest version when it starts its write, it is not the owner. It therefore sends a deallocate message back to the owner. The net result is that the Update implementation eliminates object request messages and some object fetch latency but, because of the update acknowledge and deallocate messages, sends more messages than the other two implementations. The update eliminates almost all object request messages because it almost always makes the latest version of the border block available at the adjacent processor when it starts its write.

Figure 2 presents the communication volume for each of the three implementations. The communication volume is the sum of the sizes of all messages sent during the execution of the application. Even though the Update implementation sends significantly more messages than the Version and Invalidate implementations, all three implementations generate approximately the same communication volume. The object messages that contain the border blocks dominate the overall communication volume and the small update acknowledge and deallocate messages which generate the increased message count for the Update implementation have a negligible impact on the overall communication volume.

Table 2 presents the overall running times under the three different implementations. Because the task management overhead is the limiting factor in the performance of this application, all versions exhibit comparable performance.

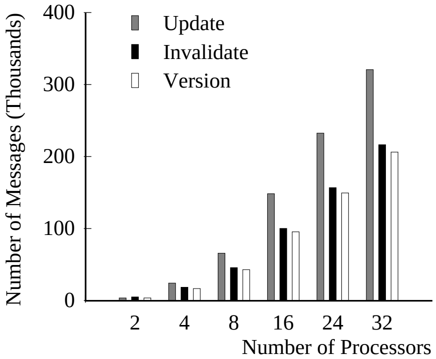

Figure 1: Total Number of Messages Sent for Ocean

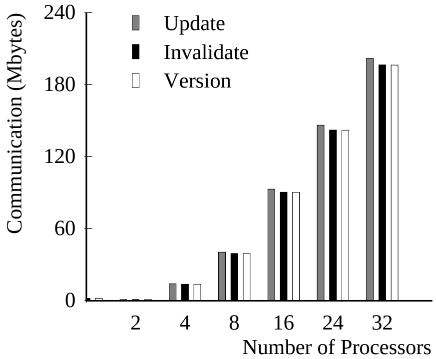

Figure 2: Total Communication Volume for Ocean

	1	2	4	8	16	24	32
Version	63.14	65.54	28.73	19.11	25.68	39.99	55.71
Invalidate	63.22	66.54	28.79	19.14	26.29	40.40	56.71
Update	78.30	65.53	28.11	18.19	26.54	40.67	56.20

Table 2: Execution Times for Ocean (seconds)

7.2 Panel Cholesky

Panel Cholesky factors a sparse, positive-definite matrix. The computation decomposes the matrix into a set of panels. Each panel contains several adjacent columns. The algorithm generates two kinds of tasks: internal tasks, which modify one panel, and external tasks, which read one panel and modify another panel. The computation generates one internal task for each panel and one external task for each pair of panels with overlapping nonzero patterns. The tasks that access a given panel are sequenced as follows. All external tasks that modify the panel execute before the internal task, which in turn executes before all external tasks that read the panel. All of the internal and external tasks that modify the same panel execute on the same processor. All panels are initialized on the processor that executes the main thread of control. The first time a processor writes a panel it fetches the panel from this processor.

We next turn to the experimental results. Figure 3 presents, for each of the three Jade implementations, the total number of messages sent during the Panel Cholesky computation as a function of the number of processors executing the computation. The Version and Invalidate implementations send an identical number of messages, which means that the Invalidate implementation sends no invalidate messages. Because all of the tasks that modify a panel execute before all of the tasks that only read the panel, the implementation never replicates a panel until the computation has performed all of its writes to the panel.

The Update implementation sends substantially more messages than the Version and Invalidate implementations. The elevated message count is caused by superfluous update messages. Every time a task completes, the Update implementation sends an update message containing the new version of the modified panel to the main processor. Because the main processor reads no updated version until the end of the computation, the vast majority of the update messages are wasted.

Figure 4 presents the communication volume for each of the three implementations. The communication volume is the sum of the sizes of all messages sent during the execution of the application. The difference in the number of messages sent translates into an even larger relative difference in the total communication volume. Panel Cholesky illustrates the major weakness of an update protocol—it may dramatically increase the overall communication overhead by sending superfluous update messages. Table 3, which contains the overall execution times for this application, shows that these superfluous update messages significantly degrade the performance. On

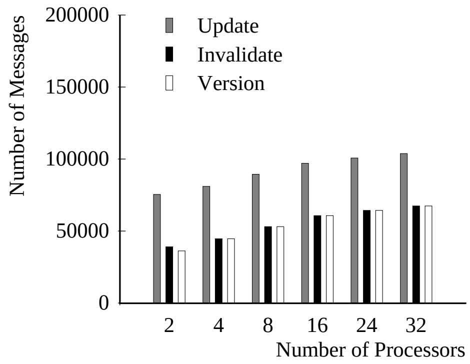

Figure 3: Total Number of Messages Sent for Panel Cholesky

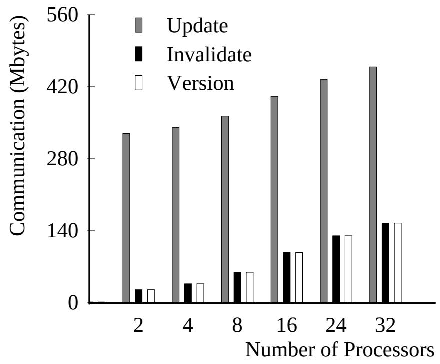

Figure 4: Total Communication Volume for Panel Cholesky

the iPSC/860 the task management overhead is the limiting factor in the performance of the Update and Invalidate versions. The computation therefore exhibits no performance increase as the number of processors increases.

	1	2	4	8	16	24	32
Update	54.57	185.97	147.70	153.07	152.40	151.35	153.14
Invalidate	37.25	51.74	32.03	32.58	34.85	37.14	39.20
Version	37.09	49.91	32.07	32.80	35.26	37.03	40.19

Table 3: Execution Times for Panel Cholesky (seconds)

7.3 Water

The Water computation performs an interleaved sequence of parallel and serial phases. In the steady state the parallel phases compute the intermolecular interactions of all pairs of molecules; each serial phase uses the results of the previous parallel phase to update an overall property of the set of molecules such as the positions of the molecules. Each parallel task reads the array containing the molecule positions and updates an explicitly replicated contribution array. Replicating this array at the language level allows tasks to update their own local copy of the contribution array rather than contending for a single copy. At the end of the parallel phase the computation performs a parallel reduction of the replicated contribution arrays to generate a final contribution array.

Figure 5 presents the total number of messages sent for Water. Figure 6 presents the communication volume. The Invalidate implementation sends substantially more messages than the Version implementation, but the difference has no significant effect on the communication volume. The large object messages dominate the overall communication volume and the small invalidate messages have a negligible effect on the overall communication overhead. The Update implementation sends fewer messages than the other implementations and generates substantially less communication volume. Different communication algorithms for the object containing the molecule positions cause the observed difference between the Update and the other two implementations.

We first consider the behavior under the Version and Invalidate implementations. At the beginning of each parallel phase each processor requests the new molecule positions from the main processor, which executed the previous serial phase. The main processor responds by serially sending the new molecule positions to each requesting processor. The Update implementation, on the other hand, uses a single broadcast message to transfer the new molecule positions to all other processors. The tasks in the new parallel phase can then execute immediately, instead of having to wait for the implementation to serially fetch the new molecule positions. The communication behavior of Water is perfectly suited for an update protocol. Every processor reads every version of every object

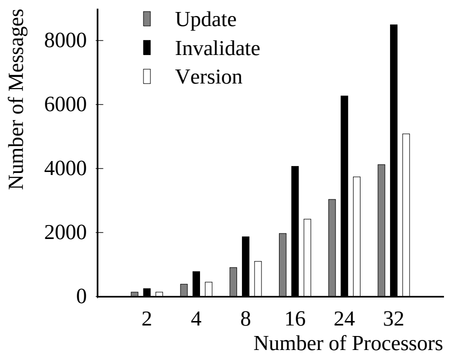

Figure 5: Total Number of Messages Sent for Water

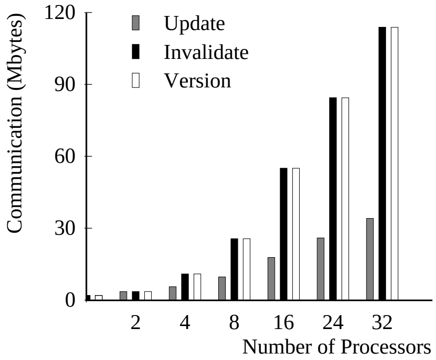

Figure 6: Total Communication Volume for Water

that the update protocol sends. The use of the update protocol eliminates object request messages, reduces the number of messages that send the same object to different processors, and minimizes the amount of time tasks spend waiting for remote data to arrive. Table 4, which presents the overall execution times, shows that the use of an Update protocol substantially improves the overall performance.

	1	2	4	8	16	24	32
Update	2439.95	1224.17	618.98	315.13	165.08	117.22	91.17
Invalidate	2458.70	1234.65	625.89	323.74	180.86	141.65	123.40
Version	2462.82	1236.85	626.98	323.94	180.72	141.47	123.14

Table 4: Execution Times for Water (seconds)

7.4 String

String[6] computes a velocity model of the geology between two oil wells. Like Water, String performs a sequence of interleaved parallel and sequential phases. The parallel phases trace rays through a discretized velocity model, computing the difference between the simulated and experimentally observed travel times of the rays. After tracing each ray the computation backprojects the difference linearly along the path of the ray. Each task traces a group of rays, reading an array storing the velocity model and updating an explicitly replicated difference array that stores the combined backprojected difference contributions for each cell of the velocity model. At the end of each parallel phase the computation performs a parallel reduction of the replicated difference arrays to generate a final difference array. Each serial phase uses the final difference array generated in the previous parallel phase to generate an updated velocity model.

The qualitative communication behavior of String is similar to the qualitative communication behavior of Water. The velocity model object in String generates the same communication and invalidation pattern as the molecule position object in Water. As in the Water application, the communication pattern in Water is perfectly suited for an update protocol. We next present the experimental results for String and determine if the qualitative similarities translate into quantitative similarities.

Figure 7 contains the total number of messages sent for String. Figure 8 contains the overall communication volume. Although the Update implementation sends fewer messages and generates less communication volume than the other two implementations, the difference is less dramatic than for Water. Part of the difference is caused by size of the velocity model object relative to the other communicated objects. Another contributing factor is that the first few parallel phases of String read different velocity model objects. Because all processors must read an

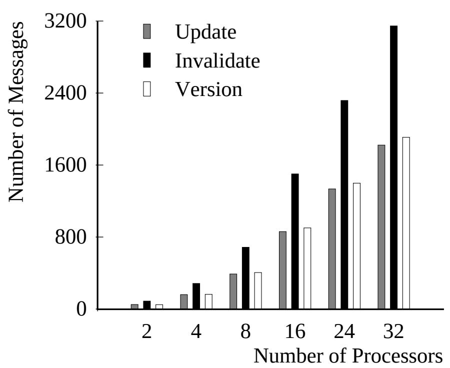

Figure 7: Total Number of Messages Sent for String

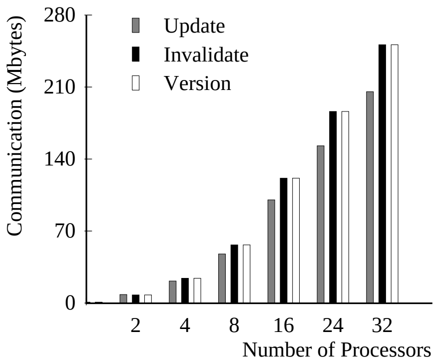

Figure 8: Total Communication Volume for String

object before the Update implementation broadcasts new versions, the Update implementation serially transfers the new velocity model to all processors for the first few parallel phases. Table 5 contains the overall execution times, which are similar for all implementations. The final factor that reduces the relative performance impact of an Update protocol for String is that String has a much smaller communication to computation ratio than Water.

	1	2	4	8	16	24	32
Update	18904.08	9484.33	4786.00	2421.24	1256.29	868.11	677.41
Invalidate	18875.04	9495.69	4778.88	2421.96	1256.72	876.82	691.96
Version	18858.15	9474.06	4788.05	2421.82	1254.74	876.53	692.23

Table 5: Execution Times for String (seconds)

8 The Adaptive Broadcast Optimization

Our applications experience led us to develop a hybrid implementation that combines the advantages of the Update and Version implementations. The largest potential performance improvement associated with the use of an update protocol comes from using a single broadcast message to distribute objects accessed by all processors. The challenge is therefore to develop an algorithm that broadcasts objects when appropriate, but does not suffer from the potential excessive communication problems characteristic of update protocols.

Our final protocol starts with the standard version based protocol, but tracks how processors access objects. If all processors ever access the same version of a given object, the implementation switches to an update protocol for that object by broadcasting each new version of the object to all processors. In practice this adaptive broadcast protocol works well—it degrades the performance of none of our applications and, as Tables 6 and 7 illustrate, both Water and String perform as well with a version protocol augmented with adaptive broadcast as they do with an update protocol.

9 Related Work

The idea of using version numbers for consistency and concurrency control is fairly old. The NAMOS system used a version-based scheme for concurrency control and consistency in a distributed database[10]. The Sprite file system also used a consistency scheme based on version numbers[9]. Both of these systems, like the Jade implementation, present the abstraction of mutable data and use version numbers as an internal implementation mechanism.

	1	2	4	8	16	24	32
Update	2439.95	1224.17	618.98	315.13	165.08	117.22	91.17
Adaptive Broadcast	2459.82	1232.12	623.50	318.63	166.69	119.30	91.75

Table 6: Execution Times for Water with Adaptive Broadcast(seconds)

	1	2	4	8	16	24	32
Update	18904.08	9484.33	4786.00	2421.24	1256.29	868.11	677.41
Adaptive Broadcast	17398.11	9483.71	4765.06	2413.20	1248.43	864.99	676.82

Table 7: Execution Times for String with Adaptive Broadcast(seconds)

SAM [15] and VDOM [5], two more recent systems in the field of parallel computation, expose the concept of version numbers directly in the programming model. When a SAM or VDOM task writes an object it provides a number for the newly generated version of the object. When a task reads an object, it specifies the number of the version it needs to access and synchronizes on the generation of that version.

Forcing the programmer to manage the version number calculation can impose an onerous programming burden. SAM addresses this problem by allowing the program to access the latest version of an object (at the cost of extra communication to find the number of the latest version) and by providing libraries that encapsulate the version number calculation for common data structures with common access patterns.

TreadMarks[7] is a page-based software distributed shared memory system that exploits the flexibility of release consistency to combine synchronization and invalidation messages. Whenever a processor acquires a lock it sends a lock acquire request message to the last processor to hold the lock. When the request arrives and the holder releases the lock, the holder sends a lock acquire response message to the requestor. The lock acquire request message contains enough information so that the holder can calculate which pages are out of date at the requestor. The holder piggybacks a list of these pages on the lock acquire response message sent to the requestor. The requestor invalidates these pages before proceeding beyond the lock acquire operation.

An evaluation of an alternate protocol that piggybacks update messages on the lock acquire response message instead of invalidate messages found that the invalidate version exhibited superior performance[7]. We conjecture that part of the reason for this performance difference is that TreadMarks does not explicitly associate data with synchronization objects. The TreadMarks implementation may therefore be unable to calculate a reasonably precise approximation to the data that the requestor will access, and the update version will communicate an excessive amount of data.

Midway[17] is a software distributed shared memory system that allows programmers to associate data objects with synchronization objects such as locks. The Midway implementation further divides each object into software cache lines and tracks cache line modifications. When a processor acquires a read lock, the implementation transfers the updated cache lines for the lock's object to the processor acquiring the lock. The message that grants the lock contains the updated cache lines. The write lock acquisition algorithm is similar except that the implementation must invalidate existing read locks. Like the Jade implementation, the Midway implementation exploits the explicit association of data and synchronization to combine consistency and synchronization messages. But Jade's version-based consistency algorithm allows it to eliminate the invalidation messages that Midway uses for consistency control. The tradeoff is that every Jade task that reads an object must acquire a version number from the object queue. Multiple Midway tasks, on the other hand, can read cached data without interacting with a central synchronizer.

10 Conclusion

Parallel systems with separate synchronization and consistency protocols may generate excess message traffic. We have presented an alternative: the integrated synchronization and consistency protocol used in the implementation of Jade. By using version numbers as a consistency mechanism, the Jade implementation is able to merge the consistency protocol into the synchronization protocol. The end result is an integrated protocol that minimizes the overall message traffic.

We have presented experimental results that compare the performance of several Jade applications under three protocols: the version number protocol, an update protocol and an invalidate protocol. The results show that the invalidate protocol may send substantially more messages than the version number protocol. The additional messages, however, do not significantly degrade the performance—the communication overhead is dominated by the large messages that transfer objects between processors.

For some applications the update protocol generates excessive message traffic that substantially degrades the performance. For other applications the update protocol improves the performance by broadcasting versions of objects accessed by all processors. Motivated by our application experience, we have developed a hybrid protocol. This protocol realizes the performance improvements that broadcasting offers, but avoids the performance degradation associated with excessive message traffic.

We believe that many of the concepts and techniques presented in this paper will prove to be applicable in other parallel systems. For example, implementations of parallel object-oriented languages would benefit from hybrid protocols that exploit hardware support for broadcast. We also expect future parallel languages to become increasingly based on a data-oriented paradigm in which programmers participate in the parallelization process by providing information about how parts of the program access data. Locality hints in COOL [3] and shared regions [14] are two examples of this trend. The implementations of these languages will be able to exploit their knowledge of how the program accesses data to employ efficient integrated synchronization and consistency protocols similar to the one presented in this paper.

References

[1] R. Berrendorf and J. Helin. Evaluating the basic performance of the intel iPSC/860 parallel computer. *Concurrency: Practice & Experience*, 4(3):223–240, May 1992.

[2] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, October 1991.

[3] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[4] I. Duff, R. Grimes, and J. Lewis. Sparse matrix problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

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

[6] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pages 82–85, 1990.

[7] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical Report Rice COMP TR93-214, Rice University, November 1993.

[8] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[9] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[10] D. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1978.

[11] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994. http://www.cs.ucsb.edu/ martin.

[12] M. Rinard and M. Lam. Semantic foundations of Jade. In *Proceedings of the Nineteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 105–118, Albuquerque, NM, January 1992.

[13] M. Rinard, D. Scales, and M. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[14] H. Sandu, B. Gamsa, and S. Zhou. The shared regions approach to software cache coherence on multiprocessors. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–238, San Diego, CA, May 1993.

[15] D. Scales and M. S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

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

[17] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for a distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.