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

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

Abstract

This paper presents experimental results that characterize the performance of 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 protocol piggybacks the version number information on the synchronization messages, it generates fewer messages than 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.

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any time a system replicates mutable data, it must solve the consistency problem: it must prevent tasks from accessing out of date replicas of shared data.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 These protocols send messages that eliminate or overwrite any out of date replicas whenever a task writes shared data.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 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.

The implementation of Jade, a portable, implicitly parallel programming language designed for exploiting task-level concurrency[5], uses an integrated synchronization and consistency protocol. It 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 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.

In this paper we present experimental results for several Jade applications running on the iPSC/860. We built versions of the Jade implementation that use invalidate, update and an integrated synchronization and communication protocols, then collected results for several Jade applications running under the different implementations. The 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.

Because of space restrictions, we have made no attempt to make this paper self-contained. A full version is available [6].

2 Experimental Results

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.

- **Version:** The Version implementation uses the object queue synchronization protocol and the version number consistency protocol.
- **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.
- **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. 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. 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.

We evaluate the Jade implementations by measuring the performance of several Jade applications running under each of the protocols. The applications are Water, which evaluates forces and potentials in a system of water molecules in the liquid state, Ocean, which simulates the role of eddy and boundary currents in influencing large-scale ocean movements and Panel Cholesky, factors a sparse positive-definite matrix. The full version of the paper also presents experimental results for String, which computes a velocity model of the geology between two oil wells.

2.1 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.

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 generated as a result of the algorithm that the Jade implementation uses to deallocated obsolete object replicas.

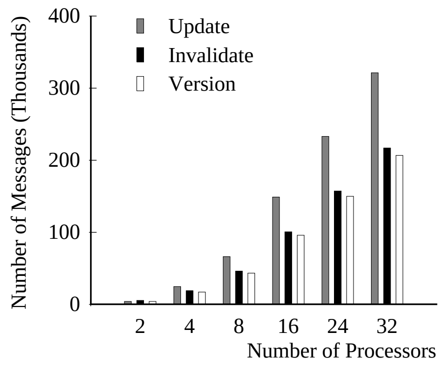

Figure 1: Number of Messages Sent for Ocean

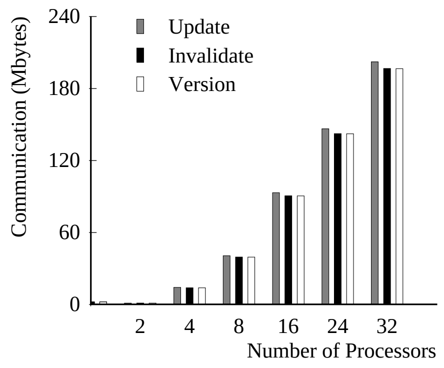

Figure 2: Communication Volume for Ocean

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.

2.2 Panel Cholesky

Panel Cholesky factors a sparse, positive-definite matrix. The computation decomposes the matrix into a set of panels. 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. 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.

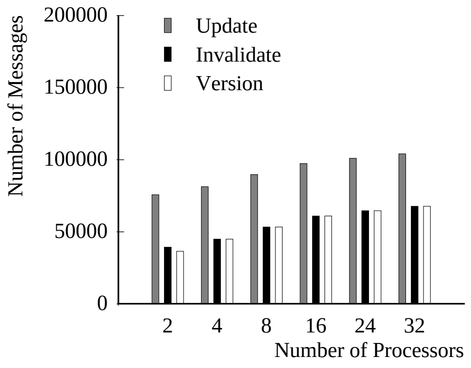

Figure 3: Number of Messages Sent for Panel Cholesky

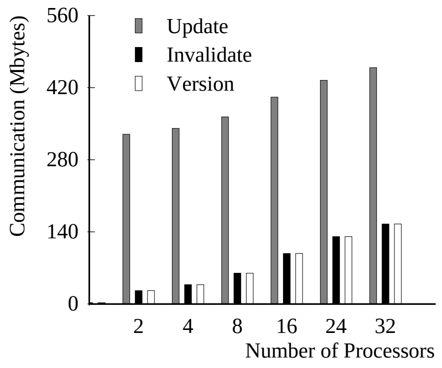

Figure 4: Communication Volume for Panel Cholesky

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.

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 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.

2.3 Water

The Water computation performs an interleaved sequence of parallel and serial phases. Each serial phase uses the results of the previous parallel phase to calculate the new molecule positions. Each parallel task reads the array containing the molecule positions and updates an explicitly replicated contribution array. At the end of the parallel phase the computation performs a parallel reduction of the replicated contribution arrays to generate a final contribution array.

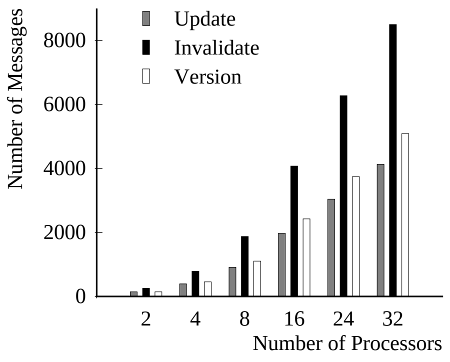

Figure 5: Number of Messages Sent for Water

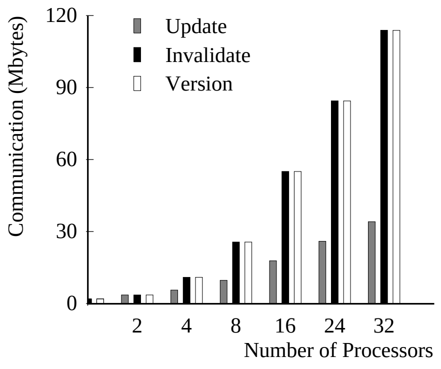

Figure 6: Communication Volume for Water

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. The reason for this behavior is that the Update implementation uses a single broadcast operation to transfer the new molecule positions, while the Invalidate and Version implementations use multiple point-to-point messages.

3 Adaptive Broadcast

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 both Water and String perform as well with a version protocol augmented with adaptive broadcast as they do with an update protocol.

4 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[4]. The Sprite file system also used a consistency scheme based on version numbers[3]. Both of these systems, like the Jade implementation, present the abstraction of mutable data and use version numbers as an internal implementation mechanism.

SAM [7] and VDOM [1], 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.

TreadMarks[2] 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.

Midway[8] 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.

5 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.

[6] M. Rinard. An integrated synchronization and consistency protocol for the implementation of a high-level parallel programming language. Technical Report TRCS95-25, Dept. of Computer Science, University of California at Santa Barbara, December 1995. http://www.cs.ucsb.edu/TRs/TRCS95-25.html.

[7] 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.

[8] 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.