Java-Based DSM with Object-Level Coherence Protocol Selection

Roumen Kaiabachev and Brad Richards

Computer Science Department
Vassar College
Poughkeepsie, NY 12604
{rokaiaba|richards}@cs.vassar.edu

ABSTRACT

This paper describes a Java-based distributed shared memory system (DSM) that simultaneously supports multiple coherence protocols. Coherence is enforced at the object level, and programmers can select the desired policy at object-creation time. Our work extends the Aleph Toolkit [1], a framework for distributed computing in Java. Since Aleph and our extensions are written in pure Java, the system is completely portable and can leverage immediately off of future improvements in the JDK. We demonstrate that a multiple-writer protocol that we added to the Toolkit outperforms existing Aleph protocols for some applications, and that a combination of protocols can provide performance superior to that resulting from the use of a single coherence scheme.

KEY WORDS

Distributed Shared Memory, Java, Coherence Protocols

1 Introduction

This paper describes a Java-based distributed shared memory system (DSM) that simultaneously supports multiple coherence protocols. Coherence is enforced at the object level, and programmers can select the desired policy at object-creation time. Our work extends the Aleph Toolkit [1]. Aleph offers several coherence protocols [2], but requires that a single scheme be selected to manage coherence for entire programs. Our work was motivated by the observation that no single protocol works best for all types of shared data [3]. We sought the same improvements for parallel Java programs that page-based DSM systems like Munin [4] have demonstrated in other domains, when matching appropriate coherence mechanisms with specific regions of shared memory based on patterns of access.

Our approach is similar to the Java DSM systems Jackal [5] and JavaParty [6], in that we use a distributed object paradigm instead of sharing arbitrary regions of memory. But Jackal and JavaParty both require custom compilers (the former to implement coherence mechanisms, the latter to handle constructs added to Java). Since Aleph and our extensions are written in pure Java, our system is completely portable, requires no special compilers or support, and can easily take advantage of improvements in the JDK. The Java/DSM [7] system creates a parallel Java environment by distributing a Java virtual machine across a collection of machines, but requires access to a commercial page-based DSM system, TreadMarks [8], and can suffer from false-sharing on some applications due to the larger coherence regions. Furthermore, none of the systems mentioned above offers simultaneous support for multiple coherence protocols.

The remainder of this paper is structured as follows. Sections 2 and 3 describe the Aleph Toolkit and our extensions. Performance results are presented in Section 4. Section 5 gives a detailed treatment of related work, and Section 6 summarizes our contributions and outlines future work.

2 Aleph

Distributed shared memory systems implement a shared global memory by transparently moving copies of data from processor to processor in response to shared-memory accesses. A coherence protocol ensures that parallel applications see a consistent view of memory by managing the replication and movement of this data. In general, a protocol ensures some form of consistency either by invalidating outstanding copies when a processor writes to a shared location, or by updating these copies with the new value.

The Aleph Toolkit provides a Java framework for shared-object distributed computing. It provides the ability to start threads on remote processors, and to communi-

```
GlobalObject g =
    new GlobalObject(new Vector(n));
Vector v = (Vector) g.open("w");
v.add(new Integer(3));
v.release();
```

Figure 1: Accessing a shared object in Aleph.

cate either via shared objects or message passing. The details of the object-sharing mechanisms are encapsulated by a `DirectoryManager` class to which all access requests are passed. Included with the Toolkit are three implementations of this class: The first is a conventional home-based protocol that invalidates outstanding copies of an object before modifications are permitted. Another implements the "Arrow" protocol [2], in which the directory for a given object contains a pointer to the processor thought to be holding the object. It may be that the processor pointed to has itself passed the object to another processor, in which case it stores a pointer to the new owner, etc. When an object is retrieved, pointers on all processors along the path to the object are reversed to reflect its new location. At most one copy of an object exists at a time, and it can therefore be modified at will. The third protocol is a cross between the home-based and arrow protocols that was found to often outperform both.

Some fine-grained DSM systems rely upon compiler-generated access tests inserted ahead of references to shared data [9, 10, 5]. If, at runtime, the shared data is not found to be held locally, coherence mechanisms are invoked to obtain a copy of the data before the access is allowed to proceed. Aleph was designed to be completely portable, and therefore must operate without the support of a custom compiler. But without a compiler (or other custom tools) to insert access tests, the system has no means for detecting accesses to nonlocal data. Instead, Aleph requires that programmers manually acquire and release shared objects as shown in Figure 1, an approach similar to that used in CRL [11]. The `open` routine is responsible for obtaining a copy of the referenced object if necessary.

3 Extensions to Aleph

We have extended Aleph in three ways. First, Aleph was modified to allow the use of multiple protocols simultaneously. Second, a data-collection facility was added to aid our performance tuning efforts. It reports the average time to satisfy coherence requests as well as information about the number of times objects moved and why. Finally, a multiple-writer protocol was implemented and added to the three protocols in the Toolkit.

Supporting multiple protocols simultaneously requires relatively minor modifications. Instead of instantiating a single protocol at startup, the directory manager instantiates all four. The `GlobalObject` constructor shown in Figure 1 was then extended. It now takes a second parameter specifying which protocol is used to manage the newly-created global object. These changes result in a trivial performance penalty at startup, as the additional protocols are initialized, but no additional overhead at runtime. The data-collection mechanism increases runtime slightly, when used, but care has been taken to minimize its impact. Collected information is stored locally on each processor as it is obtained, and only combined once a program has completed. Thus, no communication between processors is required during collection.

Our multiple-writer protocol reduces the problem of false sharing by allowing writers to simultaneously modify copies of an object. (See Section 5.) Coherence is restored once all writers have returned their copies. As in TreadMarks [8] and other similar systems, overheads are reduced by creating local duplicates of shared objects ("twins"), and returning only the differences between modified objects and their twins.

4 Performance

All performance data was taken on a collection of identical SunBlade 1000 workstations with 750MHz UltraSPARC-III processors and 1 GB of RAM. No special networking technologies were used: Communication was via the standard Java RMI and serialization libraries, and the workstations were connected by 100 Mbps Ethernet. All code was run atop the Sun HotSpot 1.3.1 JVM.

Figure 2 illustrates the basic Aleph Toolkit performance on our system. The runtimes and speedups for a set of five benchmarks are shown. The Raytrace benchmark is distributed with the Aleph Toolkit. Scene specification data is read by all processors, each of which computes contiguous columns of the 500 by 500 output pixel array. We used an input scene consisting of 113 spheres of varying sizes and properties. Mandlebrot computes the mandlebrot set on a 1600 by 1600 pixel image. A static task decomposition is used, with columns being distributed to processors in a round-robin fashion. Matrix is a standard matrix multiplication benchmark. In our tests, it computes the product of two 800 by 800 input matrices. EM3D models the propagation of electromagnetic waves through objects in three dimensions [12]. The algorithm essentially performs SOR over a bipartite graph constructed at runtime. It requires a significant amount of communication, as the graph nodes are distributed across processors and need to share values with their neighbors on each iteration. CRC is a networking-inspired benchmark. The input consists of a collection of 6400 packets, each containing 256 bytes of data and a unique sequence number. Processors compute 32-bit CRC checksums over the packets, and update a corresponding status field in an output array.

While the speedups in Figure 2 fall short of ideal, it should be remembered that these results were generated using the standard RMI mechanism over a commodity 100Mbps Ethernet network. As expected, the performance

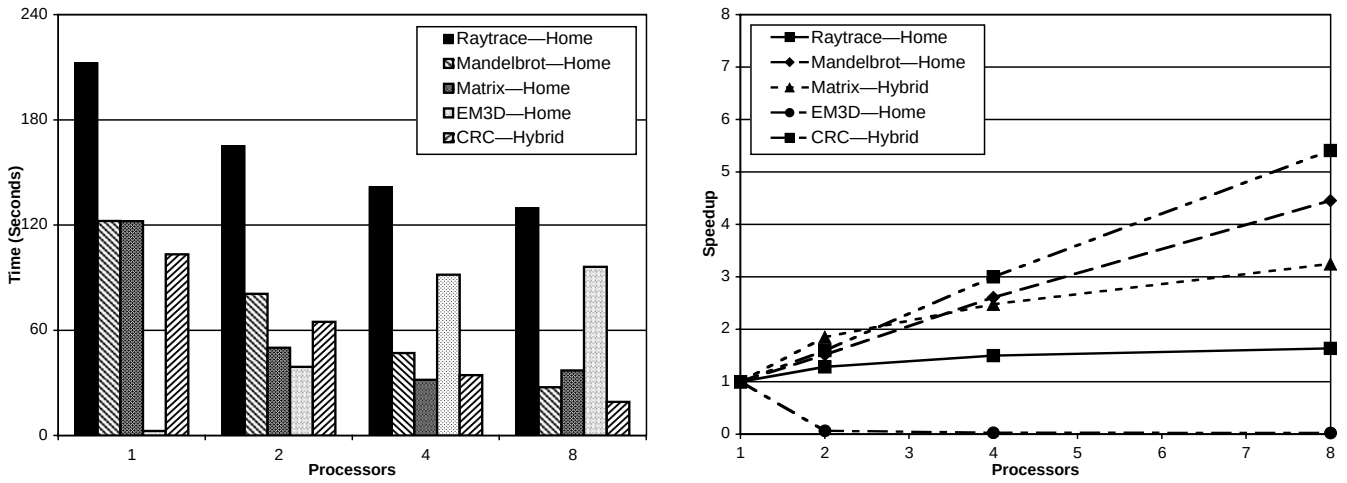

Figure 2: Basic Aleph performance.

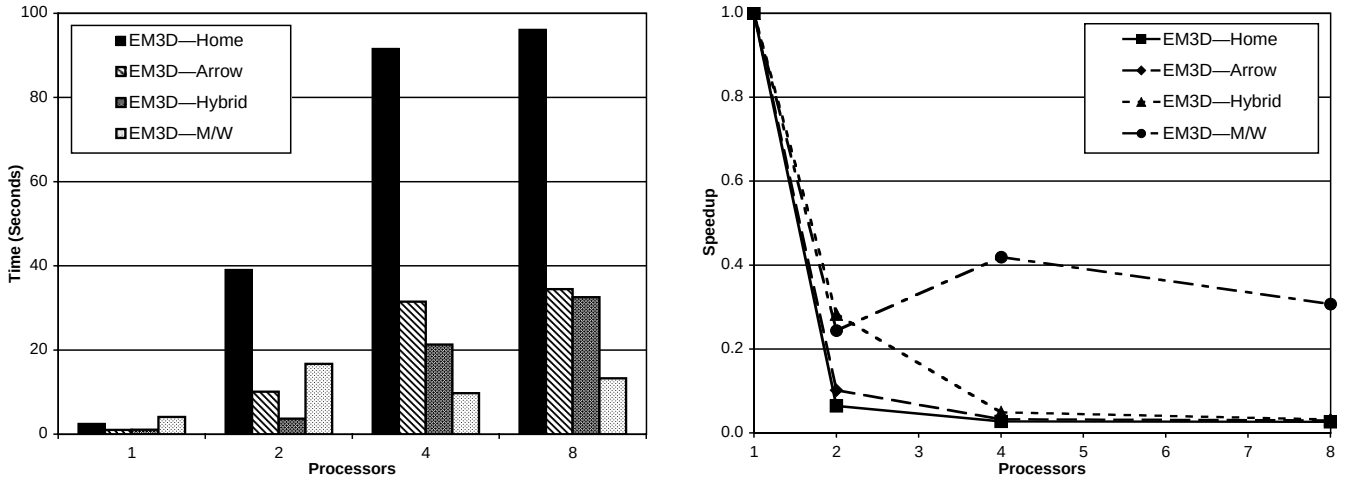

Figure 3: Effect of protocol on EM3D.

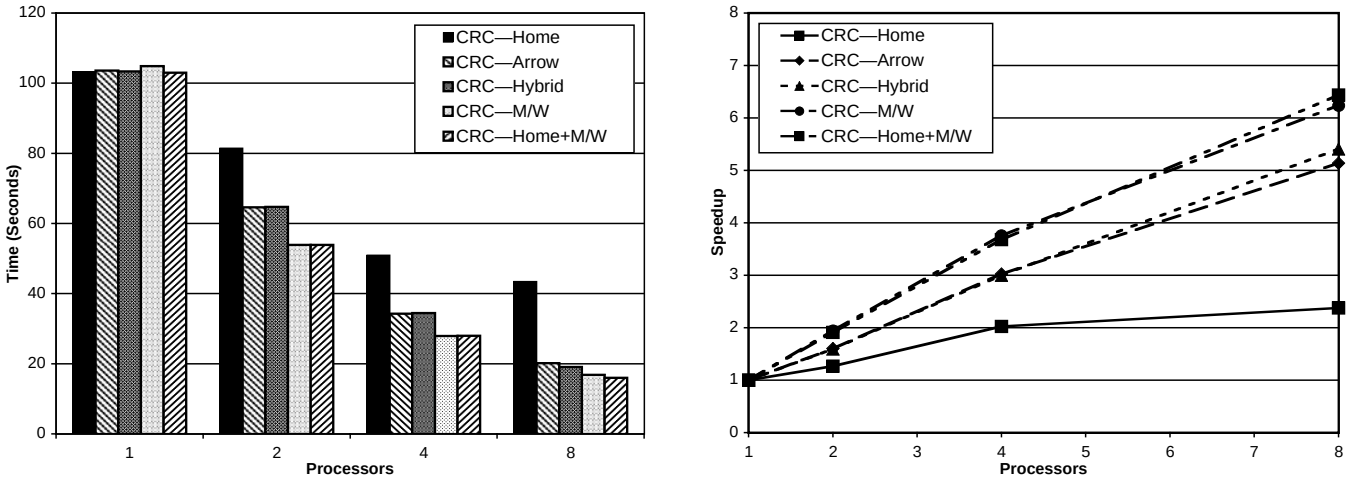

Figure 4: Effect of protocol on CRC.

for EM3D lags behind the other four benchmarks due to increased communication requirements. Speedup for Raytrace and Matrix is limited by the volume of data that must be transmitted across the network.

The benchmarks' performance depends upon the coherence protocol managing their shared data. In preparing the results for Figure 2, we ran series of tests measuring benchmark performance atop each of the basic Aleph protocols. We chose to use the results from the protocol giving the best performance on eight processors. Figures 3 and 4 show more detailed results for two of the benchmarks. In addition to the three Aleph protocols, Home, Arrow, and Hybrid, results are shown for our multiple-writer protocol. The choice of protocol clearly has a substantial effect, especially as the number of processors increases.

Note that for EM3D, the best performance is obtained when coherence is managed by the multiple-writer protocol. While it still scales poorly on our collection of workstations, performance on eight processors is dramatically improved by the multiple-writer protocol. Instead of sharing individual nodes from its graph data structure, EM3D shares *groups* of nodes to amortize communication costs over multiple nodes. This results in contention for node groups during the update computation — exactly the situation in which a multiple-writer protocol would be expected to improve performance.

Our multiple-writer protocol also improves the performance of the CRC benchmark. The execution time on eight processors drops by roughly 13% from that obtained with Hybrid, leading to a pronounced increase in speedup. This improvement is due to reduced contention for the result array, which is being modified by processors as they discover undamaged packets in the input data. The writes can be performed simultaneously and without contention via the multiple-writer protocol. This benchmark also demonstrates the potential benefit of managing portions of the shared data with *different* coherence protocols. For the final data point in Figure 4, the CRC benchmark was configured such that the input data was managed using the Home protocol, while the result array used the multiple-writer protocol. The Home protocol allows the source data to be efficiently read-shared, while results can be written into the output matrix concurrently and without contention with the multiple-writer protocol. The 5% decrease in execution time on eight processors leads to an improved speedup as well.

One might expect this multiple-protocol treatment to increase the performance of Matrix as well, but its run time is dominated by the time required to move relatively large arrays across the network. There is also very little contention for the result array by writers, since each processor obtains the result just once and writes its entire contribution to the array. Thus, using a multiple-writer protocol, alone or paired with Home, does not result in a performance gain for the Matrix benchmark.

5 Related Work

Ivy [13], the first software DSM system, took advantage of virtual memory paging mechanisms to present the illusion of a global address space shared across machines. Since the unit of sharing is an operating-system page, unnecessary contention for pages can result: Processors attempting to access distinct portions of the same page must compete for ownership. TreadMarks [8] showed that these false sharing effects can be significantly reduced through a combination of relaxed consistency models [14] and multiple-writer protocols like the one we have implemented for Aleph. Munin [4] allowed multiple protocols to be used simultaneously and, as we have, demonstrated performance improvements as a result.

Another approach to reducing false sharing is to enforce coherence on smaller units of memory. Fine-grained DSM systems like Shasta [9] and Blizzard [10] use compiler-inserted access tests to determine when coherence actions must be invoked. These systems can manage coherence on essentially arbitrarily-sized regions as a result. CRL [11] relies upon annotations inserted by the programmer to determine when shared data is being referenced, as does Aleph. (Recent work has shown that fine-grained systems are not necessarily superior to sophisticated page-based systems, however [15].)

More closely related to our work are systems like Midway [16] and Orca [17] that manage coherence at the object level. This granularity is a good match for object-oriented languages, and can produce better performance since sharing occurs on meaningful units of data instead of arbitrary regions. Neither of these systems supports Java, however.

Related Java DSM systems include Jackal [5], JavaParty [6], and Java/DSM [7]. Jackal supports coherence at the object level, but relies on compiler-inserted access tests to invoke coherence actions as necessary. The compiler optimizes away as many of these tests as possible, then compiles directly to executable code (not Java bytecode). Unlike our system, Jackal only supports a single coherence scheme — a multiple-writer home-based protocol. JavaParty extends the language with the keyword `remote`, which is used to declare globally-shared objects. A custom compiler is thus required. A sophisticated runtime system migrates objects to improve performance, but multiple coherence schemes are not supported. Instead of running separate Java virtual machines on each processor, the Java/DSM system distributes a single JVM across a collection of machines. The system is built atop TreadMarks, and is therefore a page-based system at the lowest level. (It also requires that Java/DSM users have access to TreadMarks — a commercial product.)

6 Summary

This paper describes a Java-based DSM system that allows multiple coherence schemes to be used simultaneously, and

selected at the object level. It extends the Aleph Toolkit, a distributed programming framework written in pure Java, and is therefore completely portable and can leverage immediately off of future improvements in the JDK. This is a key distinction between our work and related Java systems, as is the fact that our system allows multiple protocols to be used simultaneously. To our knowledge, no other Java DSM system can make the same claim. We have implemented a multiple-writer protocol, and demonstrated that it outperforms existing Aleph protocols for some applications. Performance results in Section 4 also show that using a combination of protocols can produce performance superior to that resulting from the use of a single coherence scheme. Future work includes implementing additional protocols and benchmarks, experimenting with more efficient RMI schemes [18], and extending the system to automatically select appropriate protocols at runtime.

Acknowledgements

The authors thank Susan Hert for her valuable feedback on earlier drafts of this paper. This work was supported in part by NSF MRI grant #0079466.

References

[1] Maurice Herlihy. The Aleph toolkit: Support for scalable distributed shared objects. In *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, January 1999.

[2] Maurice Herlihy and Michael P. Warres. A tale of two directories: implementing distributed shared objects in Java. *Concurrency: Practice and Experience*, 12(7):555–572, 2000.

[3] S.J. Eggers and R.H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proc. of the Third Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 257–270, April 1989.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

[5] Ronald Veldema, Rutger F. H. Hofman, Raoul Bhoedjang, and Henri E. Bal. Runtime optimizations for a java DSM implementation. In *Proceedings of the ACM Java Grande Conference*, pages 153–162, 2001.

[6] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in java. *Concurrency: Practice & Experience*, 9(11):1225–1242, November 1997.

[7] Weimin Yu and Alan L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice & Experience*, 9(11):1213–1224, 1997.

[8] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[9] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 174–185, Cambridge, Massachusetts, 1996.

[10] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[11] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, pages 213–228, December 1995.

[12] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in split-c. In *Supercomputing*, pages 262–273, 1993.

[13] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[14] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the 2st IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.

[15] B. Buck and P. Keleher. Locality and performance of page- and object-based DSMs. In *Proc. of the First Merged Symp. IPPS/SPDP 1998)*, pages 687–693, 1998.

[16] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[17] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.

[18] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for java. In *Proceedings of the ACM Java Grande Conference*, June 1999.

[19] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, 1996.