Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System

J. William Lee
Department of Computer Science and Engineering
University of Washington

Technical Report 93-12-05

Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System

J. William Lee
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

A distributed shared memory system provides the abstraction of a shared address space on either a network of workstations or a distributed-memory multiprocessor. Although a distributed shared memory system can improve performance by relaxing the memory consistency model and maintaining memory coherence at a granularity specified by the programmer, the challenge is to offer ease of programming while maintaining high performance. Concord meets this challenge by carefully splitting responsibilities among the programmer, the compiler, and the runtime system. Concord has allowed a single programmer to port several real, large shared-memory parallel programs onto an Intel iPSC/2 in a few weeks and achieve reasonable speedup.

1 Introduction

A distributed shared memory system provides the abstraction of a shared address space across multiple processors — either a network of workstations or a distributed-memory multiprocessor. This abstraction frees the programmer from dealing with communication explicitly, making it easier to port and write parallel programs. However, building a distributed shared memory system that performs well for real scientific and engineering applications has proved to be difficult.

In this paper, we describe an approach that attempts to provide a practical way of writing parallel programs for real applications and running these programs efficiently on distributed-memory multiprocessors. The key in our approach is a careful division of responsibilities among the programmer, the compiler, and the runtime system.

Previous distributed shared memory systems have explored the benefits of various divisions of responsibilities. Amber [4], Orca [17], and Midway [2] can avoid introducing false sharing by maintaining coherence at a granularity specified by the programmer: the programmer specifies the granularity of coherence either by defining objects [4, 17] or by associating locks with program data [2]. DASH [14], Munin [3], and Midway [2] reduce the overhead of maintaining coherence via relaxed memory consistency models; these relaxed consistency models guarantee the shared memory to be consistent as long as the programmer writes correct programs, either by correctly synchronizing threads [7] or by correctly associating locks with program data [2].

Following the direction of Amber, Orca, and Midway, our approach also maintains memory coherence at a granularity specified by the programmer and guarantees the shared memory to be consistent only for correct programs. Because this approach avoids introducing false sharing and reduces the overhead of maintaining coherence, it can potentially deliver high performance. The major challenge in this approach, however, is to offer ease of programming while maintaining high performance. Defining objects in a parallel program to match the right size for high performance can be difficult, because objects are designed to represent logical entities in a program. Associating locks with program data can also complicate programming, because an error in the process may lead to inconsistent memory [2].

Our approach meets this challenge by carefully splitting the responsibilities:

- To simplify programming, the runtime system and the compiler cooperate to make it easier for the programmer to partition shared data at fine granularity. The runtime system imposes few restrictions on how the set of data within a single data partition can be laid out in memory: this set of data can be scattered in the virtual address space, and can also grow or shrink at runtime (e.g., a linked list). The latter is possible

because, in our approach, the programmer can specify a unit of sharing using a function that traverses the data in the unit of sharing.

- To help debugging under the relaxed memory consistency model, the compiler and the runtime system cooperate to provide runtime error checking. The runtime system provides primitives to check if accesses to a region of memory are allowed, and the compiler provides an option to insert a check before every access to shared memory. Runtime error checking has been proposed before [13], but we believe that we are the first to implement it in a general DSM system and report experience of using it for real applications.

- To further improve performance, the runtime system incorporates a new update-based consistency protocol and coalesces coherence messages using information from the programmer. The latter is done by providing asynchronous primitives to acquire synchronization objects. The programmer may bracket multiple asynchronous acquire primitives in a pair of runtime system calls, as long as the order of these acquire operations does not matter. With this information from the programmer, the runtime system coalesces coherence messages while processing these acquire operations in parallel.

Our prototype system, the Concord DSM system, supports an extended C language. It consists of a precompiler and a runtime system. Currently, Concord can run applications on either a network of workstations or an Intel iPSC/2.

The rest of this paper is organized as follows. The next section describes the design and implementation of Concord. Section 3 reports the performance of Concord and our experience of programming using Concord. Section 4 discusses our approach and compares it with related work. Section 5 concludes.

2 The Design and Implementation of Concord

Similar to previous systems in this class, Concord lets the programmer manage parallelism explicitly using threads. All threads share a single address space and synchronize with each other using the primitives provided by the system. Concord supports two types of synchronization objects: barriers, and a new type of synchronization object called *folders*.

2.1 The Folder Abstraction

Briefly, a folder is a synchronization object that can be associated with a set of shared data. Folders not only provide a set of primitives to synchronize threads, but also guard shared data against accesses from threads: threads may access data associated with a folder only after it has acquired the folder explicitly.

The folder abstraction is designed mainly for two reasons. First, we'd like to have an abstraction that represents a logical unit of sharing, with few restrictions on how the data in the unit of sharing can be laid out in memory. Second, we'd like to have a synchronization object that not only supports multiple producer-consumer type of synchronization, but also fits well with the programming model of associating data with synchronization objects. This is important because some parallel programs may require synchronization more complicated than barriers and critical sections.

A folder represents a unit of sharing. The programmer may associate a folder with shared variables, subarrays, and functions that traverse through some shared data. The traversal function may walk through any program data structure, calling a built-in function to pick up the data associated with the folder along with the traversal. For convenience, we say a folder "contains" the data it is associated with.

As synchronization objects, folders extend locks with read and write primitives, versions, and "*yield*" write primitives. There are four basic folder primitives: acquiring a folder to read or write, and releasing a folder after reading or writing it. There are primitives to acquire a folder with a specific version, or a range of versions. There are also *yield* write primitives. A thread calling a yield write primitive waits for a certain number of threads to have read the current version of the folder before the thread grabs the folder to write.

Versions and yield write primitives support multiple producer-consumer type of synchronization. A consumer may request to read a folder with certain versions, waiting for the producers to write the data in the folder. A producer may request to write a folder using a yield primitive, waiting for the consumers to read the current version of the folder.

There are also two asynchronous acquire primitives that never block the calling thread. Programmers may put multiple asynchronous acquire primitives inside a pair of runtime system calls, *coalesceStart()* and *coalesceEnd()*, as long as the order of the acquire operations does not matter. The call *coalesceEnd()* returns when the calling thread has obtained all the folders

requested. These primitives are designed to tell the underlying system to coalesce coherence messages between the pair of runtime system calls. It is difficult for a DSM system to coalesce coherence messages automatically, because when the DSM system is processing an acquire request, the system does not know when the next acquire will occur and if the system can process the next acquire in parallel.

2.2 The Consistency Model

The consistency model supported by Concord, *Concord consistency*, is similar to entry consistency [2]: a thread is guaranteed to access the most up-to-date data in a folder after it has obtained the folder. Due to the asynchronous acquire primitives, Concord consistency is more "relaxed" than entry consistency: In Concord, a thread is guaranteed to obtain a folder either (1) at the point where the thread calls a normal acquire to obtain the folder, or (2) at the point where the thread calls *coalesceEnd()* after calling an asynchronous acquire to obtain the folder.

In order for the underlying system to guarantee the shared memory to be consistent, the programmer must also follow two rules in associating folders with program data. First, folders that are passed between threads concurrently must be disjoint. Second, a thread associating a folder with a set of data must have the right to write the set of data.

Essentially, Concord guarantees that the shared address space is always consistent for correct programs. Ensuring that a program is correct, however, is one of the major difficulties in programming using Concord. This difficulty is alleviated by help from the language, the compiler, and the runtime system.

2.3 The Language

The programming language, High C, augments C with a few language constructs and built-in data types and primitives. The new built-in data types include threads, folders, and barriers. The major work in designing High C was to provide succinct language constructs for associating program data with folders. We describe two language constructs below.

The programmer may associate data with a folder simply by "assigning" the folder with a list of variables, subarrays, or traversal functions enclosed in a pair of "<" and ">" symbols. For example, the following code segment associates a folder `f` with a variable `class`, a sub-block of an array `students`, and a traversal function `speakers()`. The parameter `hd` is computed when this statement is executed, and the resulted value is passed to `speakers()` each time the underlying system decides what `f` contains.

```
f = < class, students[4:5], speakers(hd) >;
```

Because scientific and engineering programs often use arrays, High C provides a way to partition an array in a single statement. For example, the following code segment partitions a two dimensional array of size `(XN×XB)×(YN×YB)` into `XN×YN` blocks of size `XB×YB`. Here, the operator "<−" initializes each individual array element with the expression following the operator, and the temporary index variables `i` and `j` implicitly go through the range of each dimension of the array.

```
float  data [XN*XB][YN*YB];
folder envelope [i:XN][j:YN] <-
  < data[i*XB:(i+1)*XB-1][j*YB:(j+1)*YB-1] >;
```

There is no pre-defined scheme for data partitioning. This approach is in contrast to the approach of High Performance Fortran [9]. Our approach attempts to provide the programmer with a few simple mechanisms for partitioning shared data.

2.4 The Compiler

The High C compiler supports the extended language constructs and runtime error checking. Our prototype High C compiler is modified from *gcc*. It translates a High C program into a C++ program and then invokes a normal C++ compiler to generate the executable code. Below we outline a few source-to-source transformations performed by the High C compiler.

To support the operator "<−" that initializes individual array element, the High C compiler generates nested loops, with the number of nested loops equal to the number of dimensions of the array.

To support runtime error checking, the compiler provides an option to insert error checking primitives before every access to shared memory. The compiler transforms each reference to shared memory into a comma expression and inserts an error checking primitive into the comma expression.

To support traversal functions with arbitrary parameters, the compiler generates an envelope function for each traversal function. This is necessary because the underlying runtime system can only call functions with fixed number of parameters through generic function pointers. The envelope function calls the traversal function, taking two fixed size parameters: one points to a data structure that packs all the parameters of the traversal function, and the other indicates the size of this data structure.

2.5 The Runtime System

The Concord runtime system consists of a user-level thread package, an interrupt-driven message passing platform, and a set of runtime routines. These runtime routines manage synchronization, maintain memory coherence, support error checking primitives, allocate memory in the global shared heap, map user-level threads onto processors, coalesce coherence messages, and handle marshaling, unmarshaling of folders. The message passing platform provides a machine-independent interface on top of both Unix sockets and the iPSC/2 message passing primitives. The runtime system contains about 16,000 lines of C++ code, most of which are machine independent.

2.5.1 Synchronization Management

Concord implements two kinds of synchronization primitives: barriers and folder primitives. In our implementation of barriers, threads on the same processor rendezvous with each other via a count variable. User-level thread systems on different processors rendezvous with each other in a way similar to the tournament barrier [8]. The synchronization of folder primitives is combined with the management of memory coherence, as described below.

2.5.2 Coherence Management

Concord maintains memory coherence in two levels. First, Concord ensures the consistency of folder data structures. Each processor detects uninitialized folder data structures using an approach similar to Amber [4]. Second, through the consistent folder data structures, Concord maintains the replication and consistency of program data using both an invalidation-based and an update-based consistency protocol.

Concord's invalidation-based consistency protocol is similar to existing invalidation-based consistency protocols. Each local folder has two states: valid and invalid. A processor with a valid folder not only has the most recent data, but also has the right to read it. For each folder, each processor maintains a "best guess" of the *owner* of the folder, the processor that is writing the folder or wrote the folder the last time. A read acquire operation that finds a valid local folder satisfying the request simply increments a count. Otherwise, the request is forwarded to the owner, possibly going through a chain of "best guesses." The owner maintains a list of folder requests in the increasing order of the smallest version requested, with requests for an arbitrary version always at the head of the list. A write acquire operation always contacts the owner. It invalidates all valid copies of the requested folder before it grabs the folder to write.

Unlike in a page-based invalidation consistency protocol, in Concord, processors do not always acknowledge invalidations immediately; a processor only acknowledges an invalidation of a folder when the number of threads reading the folder on the processor reaches zero.

Concord only uses its update-based consistency protocol under the direction of the programmer. When a runtime system call informs the system that a folder has a stable sharing pattern, Concord switches the consistency protocol of the folder to the update-based protocol. The owner of the folder then starts to remember all the processors that have requested the folder. In the reply to these requests, the owner promises all these processors to update their local copy of the data. When a read acquire operation finds a valid local folder that can satisfy the request, it increments a count and returns. Otherwise, the read acquire operation simply waits on its local processor for either the next update or a version that can satisfy the request. As in the invalidation-based consistency protocol, each write acquire operation still invalidates all other valid copies of the requested folder. A write release operation eagerly sends the new version of the folder to all the processors sharing the folder and sets all these copies as valid. These update messages are not acknowledged.

This update-based protocol is correct. Because write acquire primitives invalidate other valid copies of a folder, the synchronization of folder primitives is ensured. Furthermore, this protocol avoids the pitfall described by Bal and Tanenbaum [1] because an update of a folder can only cause an event in the computation of a processor when a thread on the processor acquires the folder.

When a folder has a stable sharing pattern, this update-based protocol can be more efficient than the invalidation-based protocol for two reasons. First, new data is propagated sooner. Second, readers do not send messages to request folders.

The update-based consistency protocol incorporates an optimization to further exploit the stable sharing pattern. In the update-based consistency protocol, each processor keeps track of how many times a folder is read between invalidations. According to this information, a processor may voluntarily give up the right to read a folder in a read release operation, assuming the folder is read equal number of times after each update. In this way, the next writer of the folder will not have to invalidate the copy of the folder on the processor.

2.5.3 Marshaling and Unmarshaling Folders

When Concord transfers a folder, it marshals the folder into a message buffer, sends the message, and unmarshals the folder at the destination node. If a folder is associated with a function, Concord uses the function to both marshal and unmarshal the folder. This is done by switching the role of the built-in *include()* function called along the traversal. When Concord marshals a folder, the *include()* function copies the included program data into a message buffer. When Concord unmarshals a folder, the *include()* function copies data from the message buffer into its place in the virtual address space, possibly using the virtual address specified in program data previously unmarshaled along the traversal.

2.5.4 Coalescing Coherence Messages

To coalesce coherence messages, on each processor, Concord allocates a request registry and a reply registry. Whenever they are "open," they capture outgoing folder requests and replies, coalesce them, and send several requests or replies in a single message. Concord always marshals a folder into and unmarshals a folder from a message buffer directly, avoiding a potential extra copy.

The registries are opened whenever there is a good opportunity to coalesce messages. In particular, the *coalesceStart()* operation always opens the request registry. The *coalesceEnd()* operation always flushes out all the request messages.

2.5.5 Supporting Error Checking Primitives

When error checking is turned on, each thread in Concord keeps track of the folders it has acquired to either read or write. An error checking primitive checks accesses to a range of memory against this list of folders, in the same way as marshaling and unmarshaling folders.

3 Performance

This section evaluates the performance of Concord and reports our experience of using Concord to program applications, including three applications from the SPLASH benchmark: Water, Barnes-Hut, and LocusRoute [16]. These three applications are real scientific and engineering applications, as claimed by their creators. Two of them, Barnes-Hut and LocusRoute, have irregular data structures. They represent the kind of applications for which it is very difficult to write message passing programs.

Application	Lines of code in original programs	Code increase in High C
Water	1500	18%
Barnes-Hut	2750	52%
LocusRoute	6400	16%

Table 1: The sizes of the original SPLASH benchmark programs and the amount of code increased.

3.1 Programming Experience

The porting of the three SPLASH benchmark programs took a few weeks for a single programmer. In general, the derived High C programs neither modify the major data structures nor change the structure of computation. The major modification to the original programs is to use folders to block shared data. Table 1 shows the sizes of the three SPLASH benchmark programs and the amount of code added in the High C programs. The percentage of code added to Barnes-Hut is more than that for the other two programs because Barnes-Hut is a very complicated program: it re-partitions a tree of bodies at the beginning of each iteration of its main loop.

Our experience suggests the following conclusions. First, the runtime error checking is essential for programming under the Concord relaxed memory consistency model. Error checking helps debugging: it is extremely difficult to debug a parallel program without knowing that some threads access stale data. Error checking also helps the programmer to understand the sharing of a complicated program. This is because error checking can often identify accesses to shared memory that the programmer is not aware of initially. Runtime error checking may slow down a program considerably. But this is usually not a problem: programmers typically debug scientific and engineering programs with a small problem size and increase problem size only for production runs.

Second, programming in High C is not too difficult, especially for programs that have a straightforward way to partition the shared data, such as Red/Black SOR, Water, and LocusRoute. For these programs, once the programmer understands how threads share data, adding the extra code required for the relaxed memory consistency model is often mechanical. The language support reduces the possibility of making errors, and the runtime error checking makes errors easy to detect.

Third, programming in High C is easier than writing message passing programs. In High C, threads

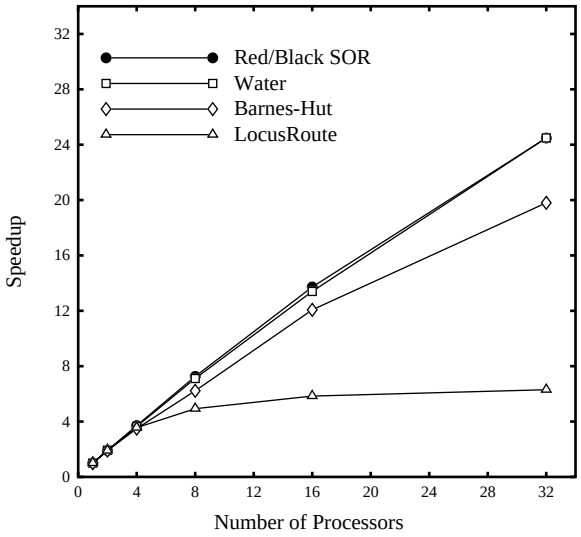

Figure 1: Speedup of applications programmed using the Concord DSM system.

Application	Problem size
Red/Black SOR	512 × 512 data points
Water	512 water molecules
Barnes-Hut	8192 bodies
LocusRoute	3817 wires & 20 channels

Table 2: The problem sizes of the measured runs.

communicate through shared variables, the meanings of which are determined statically. In a message passing program, threads communicate by sending messages, the meanings of which are often determined by the order of the messages.

3.2 Application Performance

Figure 1 shows the speedup of the four applications using 1 to 32 processors. The problem sizes of the measured runs are reported in table 2. The performance is measured on an Intel iPSC/2.

The iPSC/2 we used is a relatively slow machine. Its 386-based processor ranks about 0.3 MFLOPS and 2-3 VAX MIPS. A round trip short message on the iPSC/2 takes about 700 microseconds. Compared to DASH [14], the iPSC/2 processor is about 10-30 times slower. Sending a round trip short message on the iPSC/2 is about 170 times slower than fetching a remote cache line on DASH in the worst case.

As shown in the figure, all applications achieve good speedup except LocusRoute. Compared with the measured runs of these three applications on the DASH prototype [14], our measured runs used the same problem sizes for Water and LocusRoute, and a smaller problem size for Barnes-Hut. Although the relative performance of communication on the iPSC/2 is much lower than that on DASH, our performance results for Water and Barnes-Hut are comparable with the results on the DASH prototype. The speedup of LocusRoute is, however, poorer.

application	reduction in # of messages	Reduction in runtime
Red/Black SOR	71%	10.1%
Barnes-Hut	34%	8.4%
LocusRoute	1.4%	1.4%

Table 3: The advantage of using asynchronous acquire primitives

Red/Black SOR, Water, and Barnes-Hut achieve good speedup mainly because the folders allocated in these programs capture correctly the information about sharing. This information allows Concord to avoid false sharing and perform "scattered-gather," transferring data scattered in memory in a single message. The traversal functions turned out to be very useful. Water uses a traversal function to associate a folder with certain components of a block of molecules. Barnes-Hut uses a traversal function to associate a folder with certain components of a subtree of bodies.

LocusRoute performs poorly mainly because LocusRoute has low locality and the iPSC/2 provides relatively slow message passing primitives. LocusRoute routes wires on a chip, accessing a cost array while making routing decisions. On the average, it only takes 52 milliseconds of CPU time to route a wire. During this time, on 32 processors, a processor sends on the average 73 messages due to low locality in accessing the cost array.

3.3 Effect of Asynchronous Acquire Primitives

Table 3 shows the effectiveness of using asynchronous acquire primitives in three applications when they run on 32 processors. The results depend on the applications because some applications have more opportunities to coalesce messages than others. On the average, for these four applications, the use of asynchronous acquire primitives reduces the number of messages by 27% and the runtime by 5%.

3.4 Effect of the Update-based Consistency Protocol

For Red/Black SOR, using the update-based consistency protocol instead of the invalidation-based consistency protocol reduces the number of messages by 41% and the runtime by 9.4%. But the update-based consistency protocol is not very useful for the three SPLASH benchmark applications because these applications are very dynamic.

4 Discussion and Related Work

There is a large body of literatures about distributed shared memory systems [15, 14, 4, 17, 6]. Below we compare our approach with Munin [3], TreadMarks [12], and Midway [2]. We also discuss briefly the tradeoff between our approach and the approach of compiling data-parallel languages such as High Performance Fortran [9]. The latter has received considerable support from both academia and industry.

Munin implements release consistency and employs multiple consistency protocols. It incorporates an update-based multiple-writer protocol that uses the *diff* operation to find out the updates within a page. This approach reduces the impact of false sharing, but eager update-based protocols may send more messages than necessary for dynamic programs. TreadMarks alleviates this problem by implementing release consistency lazily, reducing communication considerably for dynamic programs [12].

Compared with Concord, both Munin and TreadMarks provide a programming model that is much closer to the conventional shared-memory programming model. But Concord may potentially achieve better performance because the programmer associates data with folders. Concord's update-based consistency protocol also differs from Munin's update-based consistency protocol in two ways: (1) Concord's protocol also synchronizes threads. (2) While Munin flushes all updates when releasing a lock, Concord only flushes a folder when releasing the folder. Unfortunately, due to differences in processor and network speed, it is difficult to compare our performance results with theirs.

Although Concord requires the programmer to do more work, Concord also helps the programmer in debugging programs. The runtime error checking can identify in some cases race conditions, a common type of program error that is often considered as one of the most important barriers in programming shared-memory programs [11, 5]. Thus it is difficult to determine if it is more difficult and how much more difficult it is to program in Concord than either in Munin or TreadMarks.

Midway is probably the existing DSM system that is closest to our approach. Midway reduces the overhead of maintaining coherence via entry consistency, which requires the programmer to associate program data with locks. To simplify programming, Midway supports multiple consistency models. Midway also lets a compiler insert code to manage logical time-stamps for programmer-specified "cache" lines. This approach can reduce the size of coherence messages. But because a processor has to update a logical time-stamp each time the processor writes shared data, this approach may increase the computation inside the inner loop of a parallel program. Nevertheless, it is difficult to assess the cost of managing logical time-stamps from the limited performance results published about Midway [2].

Unlike Midway, Concord attempts to simplify programming through the language, the folder abstraction, and the runtime error checking. The folder primitives include not only lock primitives, but also primitives for multiple producer-consumer type of synchronization.

Concord always transfers a complete folder. This scheme may send more data than necessary, and it relies on the programmer to partition data at a fine granularity to achieve good performance. On the other hand, this scheme avoids managing Midway's logical time stamps, and locking at a fine granularity can also increase the parallelism in a program.

High Performance Fortran (HPF) is a data-parallel language. It augments Fortran with parallel loops and data distribution annotations. Programming in HPF can be easy because the program annotations are treated as hints. Reasonable performance has been achieved by compilers for relatively small and static data-parallel programs [10]. Achieving good performance for larger and more dynamic programs may be harder, because runtime analysis causes overhead and static analysis can be difficult for large programs due to interprocedure analysis and symbolic computation.

Compared to HPF, our programming language, High C, may be harder to program for data-parallel applications. But High C programs are not limited to data-parallel programs. High C programs can potentially perform better because the programmer expresses parallelism explicitly.

5 Conclusion

Concord is one attempt to provide a distributed shared memory system practical for real scientific and

engineering applications. By carefully dividing responsibilities among the programmer, the compiler, and the runtime system, Concord has allowed a single programmer to port several real, large shared-memory parallel programs onto an Intel iPSC/2 in a few weeks and achieve reasonable speedup.

Our performance measurement and programming experience suggest the following conclusions. First, with appropriate support from the runtime system, the compiler, and the language, it is not too difficult for the programmer to partition shared data logically at a fine granularity. The performance gain can be significant. Second, runtime error checking is essential for programming under some relaxed memory consistency models. Third, both Concord's update-based consistency protocol and asynchronous acquire primitives can improve the performance of some applications significantly.

Acknowledgments

The author is grateful to Ed Lazowska, Brian Bershad, Ed Felten, and Mike Feeley for useful conversations and helpful comments. This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer.

References

[1] H. E. Bal and A. S. Tanenbaum. Distributed Programming with Shared Data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, Oct. 1988.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON '93)*, pages 528–537, Feb. 1993.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[4] J. S. Chase, F. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Dec. 1989.

[5] J.-D. Choi and S. L. Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *Proceedings of the third ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pages 145–154, Apr. 1991.

[6] M. J. Feeley and H. M. Levy. Distributed Shared Memory with Versioned Objects. In *Proceedings of the 1992 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 247–262, Oct. 1992.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[8] D. Hensgen, R. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0 DRAFT, Jan. 1993.

[10] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary Experiences with the Fortran D Compiler. In *Proceedings of Supercomputing '93*, pages 338–350, Portland, Oregon, Nov. 1993.

[11] A. H. Karp and R. G. B. II. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, 5(5):52–66, Sept. 1988.

[12] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–131, Jan. 1994.

[13] M. S. Lam and M. C. Rinard. Coarse-Grain Parallel Programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pages 94–105, Apr. 1991.

[14] 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 Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[15] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[16] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[17] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, pages 10–19, Aug. 1992.