An Integrated Approach to Distributed Shared Memory

Alan L. Cox, Sandhya Dwarkadas,
Pete Keleher, Willy Zwaenepoel
Department of Computer Science
Rice University, Houston, TX, U.S.A.

Abstract

We are developing an integrated user-compiler-runtime approach to programming distributed memory machines. Our approach is to provide a shared-memory programming paradigm through a global address space since shared memory is easier to program than message passing. We integrate the ability for user input through the use of high-level synchronization operations, runtime efficiency through the use of lazy release consistency and multiple writer protocols, and compiler support to assist in prefetching data and removing consistency overhead by switching to message passing directly when the communication patterns are predictable. Our optimizations allows us to take advantage of the benefits of bulk data transfer afforded by the underlying message-passing system and of combining data communication with synchronization. Our base system, which does not include any message-passing support or compiler analysis, achieves good speedups on 8 processors for several applications, including Successive Over-Relaxation (7.4), the Traveling Salesman Problem (7.2), and Genetic Linkage Analysis (5.9). We expect performance and/or usability to improve significantly with the specified optimizations for a wider range of applications.

1 Introduction

With increasing frequency, networks of workstations are being used as parallel computers. High-speed general-purpose networks and very powerful workstation processors have narrowed the performance gap between workstation clusters and supercomputers. Furthermore, the workstation approach provides a relatively low-cost, low-risk entry into the parallel computing arena. Many organizations already have an installed workstation base, no special hardware is required to use this facility as a parallel computer, and the resulting system can be easily maintained, extended and upgraded. We expect that the workstation cluster approach to parallel computing will gain further popularity, as advances in networking continue to improve its cost/performance ratio.

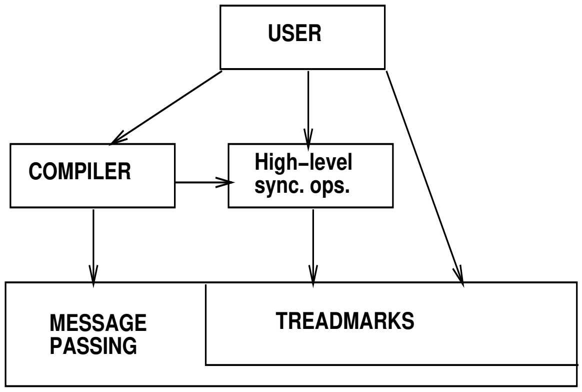

Figure 1: Integrated User-Compiler-Runtime System

The approach we provide to utilizing the workstation cluster is a shared-memory programming paradigm and a global address space. Performance is improved through an integrated user-compiler-runtime approach, as shown in Figure 1. Specifically, we integrate the ability for user input through the use of high-level synchronization operations, runtime efficiency through the use of lazy release consistency [10] and multiple writer protocols [5], and compiler support to assist in prefetching data and removing consistency overhead by switching to message passing directly when the communication patterns are predictable.

The user programs use a global address space and a shared memory interface. This shared memory in-

This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, and CDA-9310073, and by the Texas Advanced Technology Program under Grant 003604012.

terface includes a simple, yet powerful, user-level programming environment with high-level synchronization operations that expose intended communication requirements. These operations assist the runtime system in recognizing when communication is essential, and when the communication of consistency information can be delayed until a later synchronization point, or when it can be restricted to a subset of processes.

The user program is then analyzed by the compiler component, which determines when synchronization/consistency information can be avoided and replaced directly by messages. If this analysis is possible, communication can be reduced to the minimum possible for the given program. Failing the above, compile-time analysis can insert requests for prefetching or updating data when data access patterns are predictable, in order to combine data communication with synchronization wherever possible. These techniques reduce latency by cutting down on the number of misses incurred.

The fundamental principle behind the efficiency of our runtime environment is the use of a relaxed consistency model. We have a working runtime system in TreadMarks [11], a distributed shared memory system that runs on top of a network of workstations. TreadMarks uses a lazy implementation of release consistency to allow aggregation of data communication and to limit the communication of memory consistency information to synchronizing processes. Our implementation also limits the effects of false sharing by allowing multiple writers to a page [5].

Our experimental platform is a network of 8 DECStation-5000/240s running Ultrix and using a 100Mbps ATM network. We achieved good speedups on 8 processors for several applications including Successive Over-Relaxation (SOR - 7.4), the Traveling Salesman Problem (TSP - 7.2), and Genetic Linkage Analysis (ILINK - 5.9) [11]. The TreadMarks implementation is portable: it runs on Alpha/OSF/1, SPARC/SunOS 4.x, and RS/6000/AIX 3.x as well.

The rest of the paper is organized as follows. Section 2 discusses the advantages of using a single global address space and provides an overview of the consistency model used in our system. Section 3 motivates and describes the high-level synchronization operations provided to the user. Section 4 describes our proposed compile-time support and the tradeoffs involved. Section 5 presents the performance of several applications on our system and discusses the suitability of the integrated approach. Section 6 describes related work. Section 7 summarizes our conclusions.

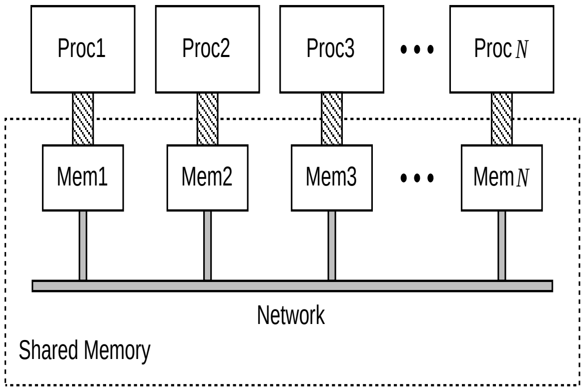

Figure 2: Distributed Shared Memory

2 Single Global Address Space

Various software systems have been proposed and built to support parallel computation on workstation networks, e.g., tuple spaces [2], and message passing [19]. TreadMarks is a *distributed shared memory* (DSM) system [17]. DSM enables processes on different machines to share memory, even though the machines physically do not share memory (see Figure 2).

The shared memory approach is attractive since most programmers find it easier to use than a message passing paradigm, which requires them to explicitly partition data and manage communication. For programs with complex data structures and sophisticated parallelization strategies, we believe this to be a major advantage. With a global address space, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values.

2.1 Lazy Release Consistency Model

The provision of memory *consistency* is at the heart of a DSM system: the DSM software must move data among the processors in a manner that provides the illusion of globally shared memory. The consistency model seen by an application or user determines when modifications to data may be expected to be seen at a given processor.

Traditionally, *sequential consistency* [15] has been the model provided to users of shared memory machines. Sequential consistency, in essence, requires that writes to shared memory are visible “immediately” at all processors. The IVY system [17] implemented this model. The disadvantage of this model is that it can result in an excessively large amount of communication since modifications to shared data must be sent out immediately in the form of invalidates or updates.

The model we provide to the user is *release consistency* (RC) [8] and TreadMarks uses a *lazy release consistent* (LRC) implementation [10]. Further details on TreadMarks may be found in [11].

RC is a relaxed memory consistency model that addresses the above-mentioned efficiency problems of sequential consistency. In RC, synchronization operations are made explicit and categorized into *acquires* (corresponding to getting access to data) and *releases* (corresponding to providing access to data). RC allows the effects of non-synchronizing shared memory accesses to be delayed until a subsequent *release* synchronization by the same processor is performed. This optimization is based on the assumption that any other process will expect to see the *release* before looking for previously modified data.

The LRC algorithm used by TreadMarks delays the propagation of modifications to a processor until that processor executes an *acquire* since the process will access any modified data only after the *acquire*. At that time, the last releaser piggybacks a set of *write notices* on the lock grant message sent to the acquirer. These write notices describe the shared data modifications that precede the *acquire*. The acquiring processor then invalidates the pages for which the incoming write notices represent modifications that it does not already know about in order to ensure that these modifications will be brought in when the data is accessed. Thus, by limiting communication to the synchronizing processes, transmission of unnecessary consistency information and data is avoided.

On an access fault, a page is validated by bringing in the necessary modifications to the local copy in the form of *diffs*. A diff is a run-length encoding of the changes made to a single virtual memory page. The use of diffs avoids the necessity of sending entire pages on every access miss and also multiple writers to access a page at the same time.

3 High-level Synchronization Operations

The available parallelism in a program and the data necessary for a certain computation are sometimes intuitively obvious to the user, while requiring significant compiler analysis or runtime overhead. In order to expose this information, we provide users with high-level operations that specify more clearly any intended ordering of data accesses, or any necessary synchronization. By making the intended use explicit, we can tailor the implementation of each operation for its intended use, thereby achieving reduction in latency and communication overhead. We therefore identify a small number of high-level operations often used by programmers, and provide support for these. Examples of these include the above-mentioned task queues and accumulation operations (see [1] for further details).

Access to shared data in a shared memory program is mediated through synchronization operations. Locks, barriers, and condition variables are the most common primitives provided to the user. One example of the use of locks is to implement work queues. The work queue imposes an ordering relation between parts of the execution of the different processes. The work done / modifications made by the process doing an `enqueue` must be visible to the process that does the corresponding `dequeue`. The use of locks entails that the enqueuing or dequeuing process will actually synchronize with all other processes that have previously accessed the work queue, resulting in unnecessary communication. This communication can be avoided through the use of high-level task queues.

Another common operation is the accumulation of values into a variable. While mutual exclusion is implied by this operation, no synchronization may be implied. If the user can expose the intended use of the operation, communication can be significantly reduced by advancing the accumulation operation to a later synchronization point.

4 Compile-time Support

Another component in our integrated approach to an efficient distributed shared memory system is compile-time analysis. Section 2.1 describes a protocol by which pages that are modified result in invalidations at other processors with which a process communicates. This can result in a large number of miss messages, two or more per page.

Ideally, one would like to bring in all data accessed by a process asynchronously without the process having to miss on the page / request the data. One solution to this is to keep track of all data requested by any processor, and based on the knowledge of what the processor has accessed in the past, update the relevant pages. This *hybrid* protocol [6] is an attempt to achieve most of the advantages of both a pure update and a pure invalidate protocol. Updates are sent only for the modifications present at the releaser, and invalidations for the remaining modifications. Like the invalidate protocol, the hybrid requires modifications to be propagated only from the processor that last released the lock to the acquiring processor, thereby reducing lock acquisition latency. All the updates can be piggybacked on the lock grant message, achieving the same number of messages for lock acquisition as the invalidate protocol. Furthermore, the amount of data exchanged is smaller than for the update protocol. The data transmitted by the releaser is also the most recently modified, and hence most likely to be accessed by the acquiring processor.

While the runtime system can optimize communication by keeping track of past access patterns, it cannot predict changes in future access patterns, which can result in large amounts of excess data being brought in. Compile-time analysis can collect information about the accesses made by a process between acquires and supply hints to the runtime system during execution. At the time of the *acquire*, the prefetch hints supplied by the compiler are "piggy-backed" on the *acquire* request message. Bulk data transfer for efficient coarse-grained data sharing can then be taken advantage of in the releaser's reply, with a more accurate estimate of the data required by the acquirer. When the releaser allows the *acquire* request to succeed, it piggy-backs the modifications it has for the requested pages, and sends invalidation notices for the rest (Figure 3). The acquirer is not required to wait for the updates, but continues computation, stalling only if it accesses a page that has not yet been updated. Pages that are not present at the releaser are pushed asynchronously by the runtime system while the process continues execution (see Figure 3). Asynchronous prefetching reduces the potential problem of larger acquire latencies due to data shipping at the time of an *acquire*. This results in a reduction in the number of miss messages as well as provides the ability to overlap the communication and computation requirements of an application.

Mechanisms such as the *data access descriptor* [3] provide compact representations of data access and traversal information, and will be used to efficiently determine the required access information. However, irregular access patterns can result in complex runtime analysis of the pages required to be prefetched. In such situations, allowing normal consistency actions to proceed by default may result in the best overall performance.

To further improve efficiency in this environment, the underlying message passing must be used directly

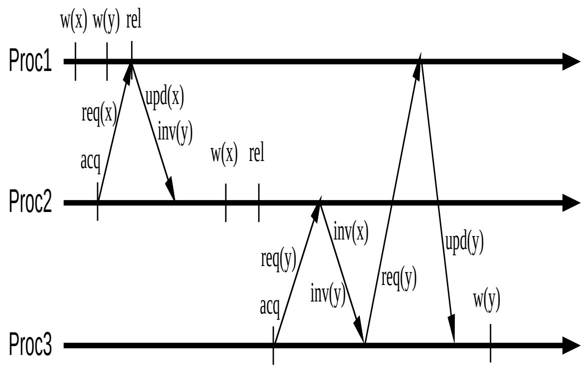

Figure 3: Compiler-based Update / Prefetch

wherever possible and the overhead involved in consistency maintenance avoided. Since messages are expensive, the other advantages afforded by this technique are bulk data transfer, combining data transfer with synchronization, and avoiding the effects of false sharing. In SOR (Successive Over-Relaxation), since all communication is nearest-neighbor, and the data to be communicated is known exactly, the messages necessary are easily generated for this program (by an automatic parallelization tool) and the overhead of consistency maintenance can be avoided. Our approach is to free the programmer from having to worry about the details of message passing and have the compiler extract any predictable communication patterns. Thus, integrating message passing in the global address space provides the advantage of the ability to exploit compiler advances. The use of a global address space also simplifies compiler-based analysis, since it avoids the necessity for local to global name space translation.

5 Performance

In this section, we present results for our runtime system without any compile-time optimizations or user-level hints / high-level operations. The system used to evaluate TreadMarks consists of 8 DECstation-5000/240 workstations, each with a 40Mhz MIPS R3000 processor, a 64 Kbyte primary instruction cache, a 64 Kbyte primary data cache, and 16 Mbytes of memory. The workstations are connected to a high-speed ATM network using a Fore Systems TCA-100 network adapter card supporting communication at 100 Mbits/second. In practice, however, user-to-user bandwidth is limited to 25 Mbits/second. The ATM interface provides a high

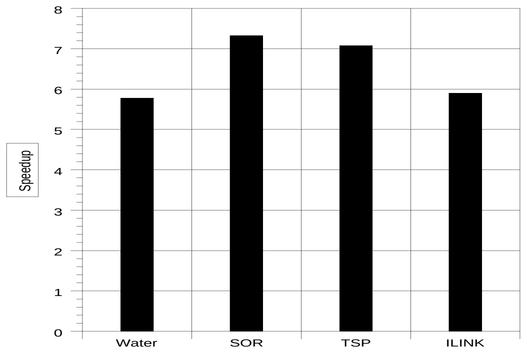

Figure 4: TreadMarks - Speedup on 8 Processors

aggregate bandwidth because of the capability for simultaneous, full-speed communication between disjoint workstation pairs. The workstations run the Ultrix version 4.3 operating system. TreadMarks is implemented as a user-level library that is linked in with the application program. No kernel modifications are necessary.

Figure 4 presents the speedups for four applications on 8 processors. The applications presented are ILINK, SOR, TSP, and Water.

ILINK, from the LINKAGE package [16], is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. We present results for the CLP input data set (see [7] for more details). Red-Black Successive Over-Relaxation (SOR) is a method for solving partial differential equations. We ran SOR for 100 iterations on a 2000×1000 matrix. TSP solves the traveling salesman problem using a branch-and-bound algorithm. We used a 19-city problem as input. Water, a modified version of the program from the SPLASH suite [18], is a molecular dynamics simulation. We present results for Water for a run with 288 molecules and 5 time steps.

5.1 Discussion

The speedups presented in Figure 4 indicate that the base system performs reasonably well for most of the applications presented. Each of the applications presented were also chosen to demonstrate the suitability of our integrated approach and to indicate how performance my be improved.

ILINK is an application with complex data structures. Understanding the underlying genetics is necessary in order to effectively load balance the program. Therefore, this program is not directly amenable to automatic parallelization. The use of a single global address space and shared memory greatly improved the ease of parallelization of this program. Using accumulation operations further improves efficiency and ease of use [1].

TSP is a program that uses a work queue approach to generating and performing its tasks. The program has a shared, global queue of partial tours. Each process gets a partial tour from the queue, extends the tour, and returns the results back to the queue. This approach is not directly amenable to automatic loop-level parallelism, and hence our shared memory approach is the most suitable. In addition, high-level task queues can be employed to reduce the consistency information supplied when synchronizing on the task queue.

The SOR program divides the matrix into roughly equal size bands of consecutive rows, assigning each band to a different processor. Communication occurs across the boundary between bands. SOR is a data-parallel program that is easily analyzed and automatically parallelized. As is intuitively obvious, experiments have shown that performance for this application is slightly improved by automatically generating message passing code for this application. Hence, our approach of integrating message passing in the shared memory global address space environment is essential to extract the maximum performance benefits possible.

The original Water program obtains a lock on the record representing a molecule each time it updates the contents of the record. We modified Water such that private and shared variables are more clearly separated and each processor instead uses a local variable to accumulate its updates to a molecule's record during an iteration. At the end of the iteration, it then acquires a per processor lock and applies the accumulated updates to the molecules owned by each processor at once. The simpler approach of updating the contents of the record directly can be used without loss in performance by using the high-level accumulate primitive [1]. Using compiler-analyzed prefetching would also benefit this application by replacing multiple miss messages with a single update.

Molecular dynamics programs such as Water can be automatically parallelized using runtime support such as CHAOS [4]. However, depending on the dynamic nature of the irregular data access pattern, there is a tradeoff between the use of the default runtime virtual memory support and compiler-based approaches.

These examples show that no single approach to parallelization is suitable for all programs. Different programs benefit from different optimizations and different parallelization strategies. With our integrated approach, we will combine the advantages of several of these strategies in a single framework.

6 Related Work

Implicit parallelism, as in HPF [9], relies on user-provided data distributions which are then used by the compiler to generate message passing code. This approach is suitable for data-parallel programs, such as SOR. Programs exhibiting dynamic parallelism, such as TSP, are not easily expressed in the HPF framework.

Two approaches to integrating message-passing and shared-memory in a hardware coherent environment are Alewife [13] and FLASH [14]. Both try to exploit the advantages of bulk data transfer afforded by message passing.

Carlos [12] is a distributed shared memory system that integrates support for message passing with and without coherence. A system of this nature could provide the hooks necessary for our user and compiler optimizations.

7 Conclusions

In summary, we describe an integrated user, run-time and compile-time approach to providing an efficient and usable programming environment for distributed memory machines. We exploit the use of a relaxed memory consistency model and present techniques to improve the estimate of data access patterns currently used by our lazy release consistent DSM system. We have presented results to demonstrate the performance of our runtime system, and described the compiler-based and user-based optimizations to our system. We expect performance and/or usability to improve significantly with the specified optimizations.

References

[1] S.V. Adve, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Replacing locks by higher-level primitives. Technical Report TR94-237, Rice University, 1994.

[2] S. Ahuja, N. Carreiro, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[3] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[4] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. Technical Report 13, ICASE, Hampton, September 1990.

[5] 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, October 1991.

[6] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.

[7] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

[8] 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, May 1990.

[9] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[11] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[12] Povl T. Koch and Robert J. Fowler. Integrating message-passing with lazy release consistent distributed shared memory. To appear in the First Operating Systems Design and Implementation Symposium, 1994.

[13] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the 1993 Conference on the Principles and Practice of Parallel Programming*, May 1993.

[14] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. To appear in Proceedings of the 21st Annual International Conference on Computer Architecture, April 1994.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[16] G.M. Lathrop, J.M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proceedings of National Academy of Science*, 81:3443–3446, June 1984.

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

[18] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[19] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency:Practice and Experience*, 2(4):315–339, December 1990.