1

RUNTIME SUPPORT FOR PORTABLE DISTRIBUTED DATA STRUCTURES

Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, Katherine Yelick

Computer Science Division, Department of EECS
University of California, Berkeley, California 94720
USA

ABSTRACT

Multipol is a library of distributed data structures designed for irregular applications, including those with asynchronous communication patterns. In this paper, we describe the Multipol runtime layer, which provides an efficient and portable abstraction underlying the data structures. It contains a thread system to express computations with varying degrees of parallelism and to support multiple threads per processor for hiding communication latency. To simplify programming in a multithreaded environment, Multipol threads are small, finite-length computations that are executed atomically. Rather than enforcing a single scheduling policy on threads, users may write their own schedulers or choose one of the schedulers provided by Multipol. The system is designed for distributed memory architectures and performs communication optimizations such as message aggregation to improve efficiency on machines with high communication startup overhead. The runtime system currently runs on the Thinking Machines CM5, Intel Paragon, and IBM SP1, and is being ported to a network of workstations. Multipol applications include an event-driven timing simulator [1], an eigenvalue solver [2], and a program that solves the phylogeny problem [3].

1 INTRODUCTION

Multipol is a library of distributed data structures for irregular applications such as discrete event simulation [1], symbolic computation [4], and search problems [3]. These applications have conditional control constructs and dynamic data structures that produce unpredictable communication patterns and computation costs. Multipol has data structures for both bulk-synchronous applications with irregular communication patterns and asynchronous applica-

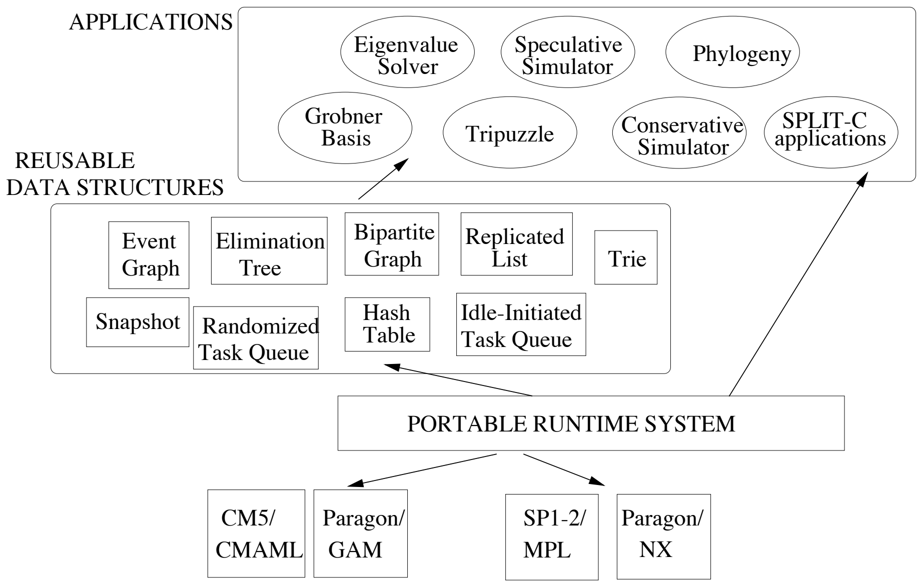

Figure 1 The Multipol architecture. The runtime layer is used to write the data structures, but can also be used directly by the applications.

tions, but in this paper we focus on the latter. Compiler analysis and runtime preprocessing, such as that used in PARTI [5], are not effective in asynchronous applications, since the computation patterns change dynamically. An overview of the Multipol library with its underlying runtime layer and example applications is depicted in Figure 1. In this paper, we explore the issues in developing irregular parallel applications and present the design of the Multipol runtime layer. We use example applications to highlight the design tradeoffs and justify our approach to latency hiding, scheduling, load balance, and communication.

The Multipol runtime system contains a thread system and a simple producer-consumer synchronization construct called a *counter* for expressing dependencies between threads. The threads also create opportunities for overlapping communication latency with computation, and for aggregating multiple remote operations in large physical messages to reduce communication overhead. In addition to threading support, the runtime system also provides a set of portable communication primitives for bulk-synchronous communication and asynchronous communication. Like TAM [6] and Nexus [7], the runtime system can be used as a compilation target, but it is primarily designed for direct programming by library and application programmers.

Our design and implementation targets distributed memory architectures, including the Thinking Machines CM5, Intel Paragon, IBM SP1, and future networks of workstations. Such machines typically have high communication latency and overhead for variable-size messages, due to buffer allocation, copying, and on some machines kernel crossing. On these machines, it is common in bulk-synchronous applications to pre-allocate message buffers and pack many values into a single message, taking advantage of global communication information at compile time or at synchronization points at runtime. The Multipol runtime system takes this idea one step further, and aggregates messages "optimistically," even when there is no information about future messages going to the same processor. The runtime system is compact and has a universal interface across distributed memory platforms, which makes Multipol easy to port.

The rest of the paper is organized as follows. Section 2 describes the thread system for hiding latency and introduces *atomic threads* and *split-phase* interfaces, which are the basic programming abstractions in Multipol. Section 3 describes our support for application-specific schedulers. Section 4 explains our approach to load balance. Section 5 presents *dynamic message aggregation*, our solution to efficiently support asynchronous communication. Section 6 compares the Multipol runtime system with other runtime systems. Section 7 summarizes the paper and reports on the current status of the runtime system.

2 MULTIPOL THREADS

In this section, we describe the Multipol thread support for latency hiding and concurrent programming. The simple design of the thread system makes it extremely easy to port. The entire thread system is written in C, and can run on most machines without modification.

2.1 Latency Hiding

A typical Multipol application resembles a shared-memory parallel program, which consists of processes communicating via shared data structures. However, the logically shared data structure is physically distributed among the processors, and the distribution is explicitly controlled by the programmer for better locality and load balance. Since distributed memory architectures do not support a shared address space, accessing remote data requires communication with the remote processor, and is inherently more expensive than local accesses.

To save communication costs, the data structure designers usually adopt an *owner computes* rule, which often involves computation migration. For example, to perform a remote look-up operation on a distributed hash table, it is typically more efficient to migrate the operation to the processor where the bucket resides, than to fetch the data items of the bucket and perform local search. Although migrating computation saves communication overhead, each individual operation may have longer latency, since it turns shorter remote read and write operations into longer remote computation operations. The overall latency of a remote operation contains not only the network transport latency, but also the remote scheduling and computation delays. Therefore, it is essential for the runtime system to provide latency hiding mechanisms.

Operations on distributed data structures seldom have substantial local computation to hide the communication latency. For example, a hash table lookup cannot proceed locally without waiting for the reply. Therefore, the latency must be overlapped with the caller's computation, which requires breaking the traditional abstractions, since a data structure operation must return before it is complete. This is accomplished by providing *split-phase interfaces* for operations that may require communication. A split-phase operation returns after doing whatever local computation is necessary, but never waits for communication to complete. The caller is required to explicitly check for completion of the operation using other synchronization mechanisms. If the operation does not require communication, it behaves like a normal procedure call. Otherwise, it creates a separate thread of control and passes its local state to the new thread which awaits the reply from the remote processor.

Synchronization can be accomplished with continuation passing, which explicitly passes the continuation thread handle to the remote processor, or with synchronization data structures such as *counter*. A counter maintains a list of threads waiting for the counter to exceed certain values. Upon completion, a split-phase operation increments the counter, which starts all threads that become eligible for execution after the increment. Counters are usually used to synchronize threads that pipeline multiple operations, where the issuing threads need to know if some number of operations have taken effect on the data structure.

Multipol supports a simple thread system which requires the programmer to specify what local state to save. The programmer can use knowledge of the data structures or the application to reduce the state saving overhead. Without knowledge from the compiler or the programmer, the runtime system would have to make conservative assumptions and save the entire processor state and the stack frame.

2.2 Invoking Remote Computation

Communication layers such as active messages [8] provide mechanisms for implementing computation migration. Direct use of active messages as remote request handlers, however, presents three major problems for implementing general data structures other than simple memory cells:

- Most active message layers either assume a fixed number of arguments (e.g., 4 words [8, 9]), whose size is tailored to the network packet size, or require the programmer to pre-allocate remote memory for holding arguments [10]. The lack of flexibility makes it difficult to express dynamic communication patterns.

- Because active messages do not implement any flow control, they usually require the programmer to follow a request/reply protocol to avoid network level deadlock. For example, request handlers can only send reply messages, which severely restricts the code that can run as a handler. The protocol causes problems for data structures which may need to re-migrate computation, such as hash table lookups that require re-hashing.

- Active message handlers may execute whenever the network is serviced (by polling or interrupt). Therefore, a local thread loses atomicity whenever the network is touched. To guarantee atomicity, the programmer must either explicitly lock all data structures touched by the thread, which is overly general for short operations, or disable interrupt and polling. Disabling network service may cause congestion. It is also hard to enforce when modules or data structures are composed in a program that has no knowledge of their implementations. Finally, the programmer has no control over how the operations are scheduled (they always execute upon reception).

In our experience, asynchronous applications require higher level programming abstractions than a SPMD model (one thread per processor) with active messages. The Multipol runtime system provides *atomic threads* as the basic programming abstraction, and unlike active message handlers, messages from the network can be accepted without affecting atomicity. Atomic threads run to completion without preemption or suspension. Except for a non-blocking restriction, which forbids the atomic threads from spinning on a condition and requires that each thread terminates in finite time, they are completely general computation constructs and can be created by remote processors. We use atomic threads as building blocks for higher-level programming constructs in Multipol data structures as well as applications. A restricted subset of active messages are used by the runtime system to send data over the network or to create remote atomic threads. Above the runtime level, all communication is done using the runtime system primitives, and not through direct access to active messages or other message layers.

The overhead of scheduling and thread management can be reduced when lack of atomicity does not affect the correctness of the program. For such applications, the programmer can use a preemptive scheduler (see Section 3) for selected threads. Such threads may preempt the running computation when the network is serviced.

3 SCHEDULING

There are many applications where the scheduling policy has a significant impact on performance. In the Gröbner basis application, for example, there are two types of tasks, one of which must be scheduled at a higher priority to keep the memory utilization and total work low [4]. In this discussion, we use Parswec, a speculative timing simulator [11] and Tripuzzle, a state space search program that counts the number of unique solutions for the tripuzzle problem [12].

In Parswec, a digital circuit is decomposed into subcircuits that are distributed among the processors, and the simulation proceeds speculatively using an algorithm similar to Timewarp [13]. A separate thread is created for each subcircuit to simulate its state. These threads are ready to start any time (subject to storage constraints), since they can speculate on the input values. However, some threads are more likely to lead to redundant work than others. Therefore, it is imperative to give higher priorities to threads that are more likely to be useful work. Also, the thread scheduling priorities must change as the simulation progresses. For example, a subcircuit that is rolled-back in time due to incorrect speculation should be given higher priority to avoid an avalanche of roll-backs. If these scheduling policies are not properly enforced, the overall running time can easily increase by more than two-fold. Therefore, applications such as Parswec not only require sophisticated control over scheduling, but also require access to the scheduler's data structure.

In contrast, in the Tripuzzle application a simple scheduling policy is sufficient, and being less complicated, it has lower scheduling overhead. The performance of Tripuzzle improves by 20% to 30% if we use a cheap scheduler that sacrifices atomicity with respect to the hash table operations, which is not required by Tripuzzle.

To accommodate application-specific scheduling policies, our runtime system allows the programmer to use customized schedulers, which are data structures with two operations: *deposit* and *select*. When a thread is ready for execution, the runtime system invokes the designated *deposit* operation to store the thread in the scheduler's internal data structure. The programmer also registers the *select* operation with the runtime system, which periodically executes the operation to choose the next thread to dispatch. In addition to the user-written schedulers, Multipol also provides common schedulers such as a FIFO and a priority queue.

Separating out schedulers as independent data structures also eases performance tuning, because scheduling decisions are localized to the implementation of the schedulers. Scheduling decisions are some of the hardest design decisions to make in advance, and by separating the scheduling abstraction we allow application programmers to select a scheduler or write their own very late in the development process. The runtime system guarantees that each registered deposit operation is executed exactly once within finite time of being registered, and the frequency of call can be configured by the programmer. The scheduling of different modules or data structures can then be tuned separately without affecting other parts of the program.

4 LOAD BALANCE

Many irregular applications can be naturally decomposed into parallel tasks. These applications include the phylogeny problem [3], which uses a parallel branch and bound algorithm to search for the largest character subset that forms a perfect phylogeny tree, a divide-and-conquer eigenvalue algorithm, and the Gröbner basis problem, which reduces sets of polynomials in parallel with respect to a growing basis.

Runtime systems such as Cilk [12] adopt a built-in load balancer and treat threads not only as a latency-hiding mechanism, but also as units of load balance which can be migrated freely. This approach suffers the same problem as providing fixed scheduling policies – different applications require different load balancing policies. For example, the phylogeny program observes a 5-fold increase in running time if tasks are migrated randomly for load balance. The significant increase in running time is due to the loss of locality, which is important for effective pruning of the search space. There are also applications where the loss of locality incurs additional communication overhead.

It is infeasible to build a general load balancer in the runtime system, since it is very difficult to predict the impact of locality on performance. We separate out load balancing policies from the Multipol runtime system and put the functionality in data structures such as a distributed task queue [14]. The programmer can then select different data structures or implement new ones to tailor the load balancing policy to a particular application. Furthermore, Multipol threads do not migrate, because they are used only for latency hiding or message aggregation (described in the next section).

5 EFFICIENT COMMUNICATION

Besides the thread system, the Multipol runtime system also provides a communication layer that is portable across a variety of distributed memory architectures. Two types of communication are supported: bulk-synchronous and asynchronous. The bulk synchronous communication primitives include *put* and *get* operations, which are split-phase versions of remote read and write. Asynchronous communication primitives include *remote threads*, which start a thread on a remote processor with a variable number of arguments, and *store*, which is similar to remote threads except that it requires the user to pre-allocate buffer space for holding the arguments.

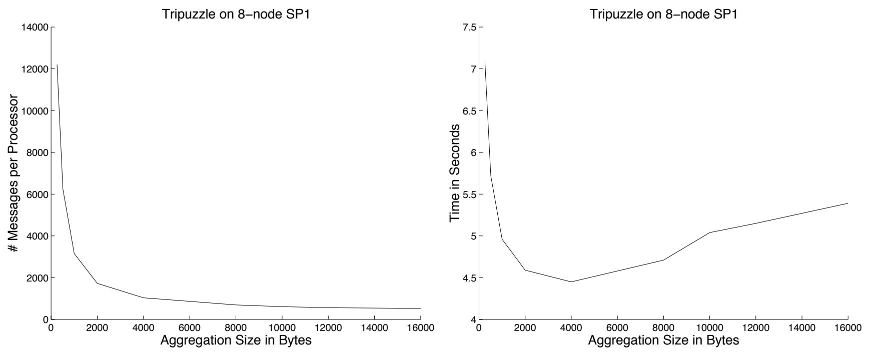

Figure 2 Effect of aggregation on Tripuzzle. The running time of Tripuzzle first decreases due to better communication efficiency, and then increases due to synchronization delay.

Many irregular applications have unknown numbers of concurrent operations, each taking some arbitrary number of arguments. Such communication patterns have low bandwidth characteristics on distributed memory architectures, which perform better for pre-allocated or large messages. The Multipol runtime system dynamically aggregates small, asynchronous messages to improve communication performance. For example, asynchronous remote thread calls are copied into a message buffer, which is sent to the destination processor in bulk when the accumulated size exceeds a certain threshold, or when the local processor runs out of work. The extent of aggregation depends on the available concurrency in the application. To allow more aggregation, the application may need to increase its level of multithreading beyond what is sufficient for latency hiding.

Message aggregation amortizes the communication startup overhead over large volumes of data. However, it introduces copying overhead, and increases the latency of remote operations. Increase in the latency may increase idle time because of the additional delay in synchronization, which is demonstrated by Tripuzzle (Figure 2). Tripuzzle enumerates the states in the state space, and its execution consists of constructing a series of hash tables that record all the possible states of a particular depth. The results showed that aggregating up to 4K bytes improves performance due to the sharp reduction in the number of physical messages. However, the benefit is offset by the increase in idle time, when the processors wait for all updates to a hash table to complete. The tradeoff results in a optimal aggregation size at about 4K byes.

The increase in latency may also generate more redundant work for speculatively parallel applications such as Parswec, which is illustrated in Figure 3. Figure 3 shows that a moderate aggregation size (1K bytes) reduces the running time by more than 20% due to the reduction in the number of physical messages. However, the running time increases by 7% when the aggregation

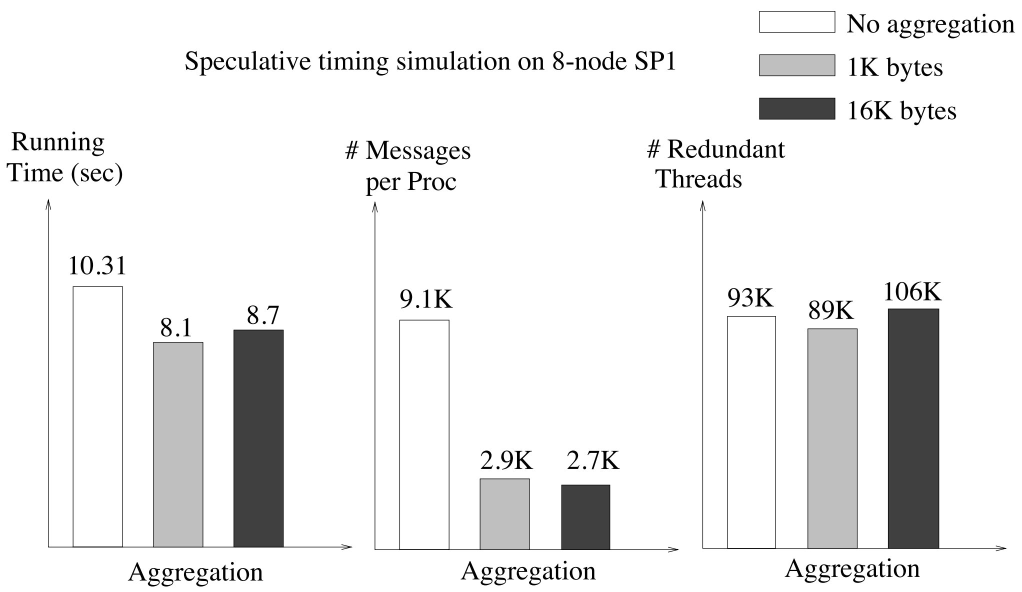

Figure 3 Effect of aggregation on Parswec. The running time of Parswec first decreases due to better communication efficiency, and then increases due to redundant work.

size changes from 1K bytes to 16K bytes due to a proportional increase in redundant work.

6 RELATED WORK

Past research has produced a variety of runtime systems such as TAM [6], the Chare kernel [15] Cilk [12], and Nexus [7]. TAM threads are similar in spirit to the Multipol atomic threads in that they are used to hide latency. However, since TAM is designed as a compilation target, the threads are statically allocated within the scope of an activation frame, and a fixed scheduling policy is used to maintain locality of threads in the same frame. The threads in Cilk are also atomic, but unlike the Multipol threads, they can be migrated for load balance. Both Cilk and Chare kernel have built-in load balancers. Nexus is designed to support heterogeneous computing, and is built on top of standard thread packages, which are more heavy weight. TAM and Cilk programs are synchronized in a data-flow fashion, while most Multipol programs communicate and synchronize via shared data structures. PARTI [5] performs runtime message aggregation, but it does not handle dynamic communication patterns where runtime preprocessing techniques cannot be applied. We have not found any runtime system that allows customized schedulers, or performs dynamic message aggregation.

7 CONCLUSIONS AND CURRENT STATUS

The design of the Multipol runtime system is motivated by our experiences in parallelizing irregular applications. Specifically, we have identified the following important features:

- Split-phase interfaces for latency hiding.
- Atomic threads as the basic programming abstraction.
- Customized schedulers for application-specific scheduling.
- Dynamic message aggregation for better communication performance.

Currently, the runtime system exists for CM5, Paragon, and SP1. Ports to network of workstations and tools for performance tuning are currently being developed.

Acknowledgements

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Department of Energy grant DE-FG03-94ER25206, by the National Science Foundation Grants CCR-9210260 and CDA-8722788, The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

REFERENCES

[1] Chih-Po Wen and Katherine Yelick. Portable parallel asynchronous simulation on distributed memory architectures. In *Internation Conference on Parallel Processing*, 1995. To appear.

[2] Soumen Chakrabarti and Abhiram Ranade and Katherine Yelick Randomized Load Balancing for Tree Structured Computation In *IEEE Scalable High Performance Computing Conference*, 1995.

[3] Jeff Jones. Exploiting parallelism in the perfect phylogeny computation. Master's thesis (TR-95-869), University of California, Berkeley, Computer Science Division, December 1994.

[4] Soumen Chakrabarti and Katherine Yelick. Distributed data structures and algorithms for Gröbner basis computation. *Lisp and Symbolic Computation*, 1994.

[5] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory multiprocessors. *Concurrency: Practice and Experience*, pages 159–178, June 1991.

[6] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991.

[7] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuccke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1991.

[8] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, 1992.

[9] Eric A. Brewer and Robert D. Blumofe. Strata: A multi-layer communication library. To appear as a MIT Technical Report, February 1994.

[10] David Culler, Kim Keeton, Lok Tim Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, and Kristin Wright. The generic active message interface specification. Unpublished, 1994.

[11] Chih-Po Wen and Katherine Yelick. Parallel timing simulation on a distributed memory multiprocessor. In *International Conference on CAD*, Santa Clara, CA, November 1993. An earlier version appeared as UCB Technical Report CSD-93-723.

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming*, 1995.

[13] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[14] Chih-Po Wen. The distributed task queue user's guide. Unpublished, 1994.

[15] Wei Shu and L.V. Kalé. Chare kernel – a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1991.