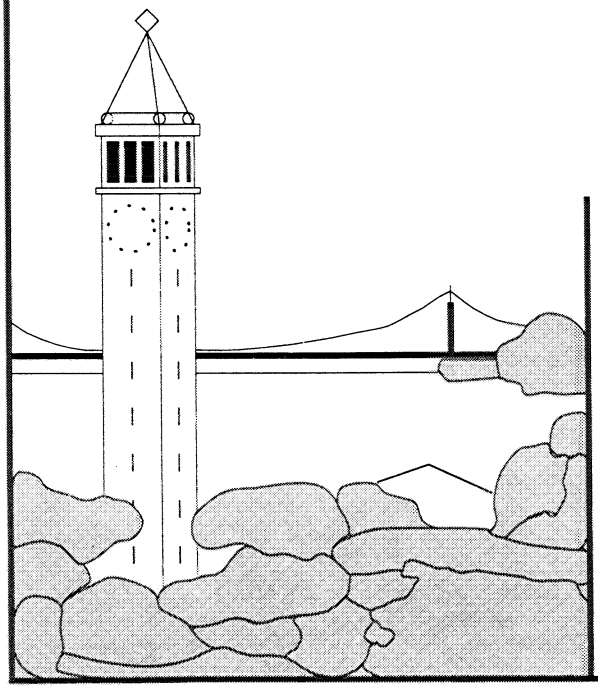
Multipol: A Distributed Data Structure Library

Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones,
Arvind Krishnamurthy, Chih-Po Wen, and Katherine Yelick



Report No. UCB//CSD-95-879

July 1995

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Multipol: A Distributed Data Structure Library

Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones,
Arvind Krishnamurthy, Chih-Po Wen, and Katherine Yelick
U.C. Berkeley, Computer Science Division
{soumen,deprit,ejim,jjones,arvindk,cpwen,yelick}@cs.berkeley.edu *

Abstract

Applications with dynamic data structures, unpredictable computational costs, and irregular data access patterns require substantial effort to parallelize. Much of their programming complexity comes from the implementation of distributed data structures. We describe a library of such data structures, Multipol, which includes parallel versions of classic data structures such as trees, sets, lists, graphs, and queues. The library is built on a portable runtime layer that provides basic communication, synchronization, and caching. The data structures address the classic trade-off between locality and load balance through a combination of replication, partitioning, and dynamic caching. To tolerate remote communication latencies, some of the operations are split into a separate initiation and completion phase, allowing for computation and communication overlap at the library interface level. This leads to a form of relaxed consistency semantics for the data types. In this paper we give an overview of Multipol, discuss the performance trade-offs and interface issues, and describe some of the applications that motivated its development.

1 Introduction

We have implemented a number of irregular parallel applications, including a symbolic algebra system [CY94], a timing level circuit simulator [WY93], a biological cell simulation [Ste94], a solution to the phylogeny problem [Jon94], and a electromagnetics simulation kernel [CDG+93]. In all of these projects, the key parallelization task was the development and performance tuning of distributed data structures. Although several languages and runtime systems support the development of such data structures [And93, BBG+93, FLR92, SK91, SL93], there are no comprehensive data structure libraries, such as those that exist for uniprocessors. Multipol is such a library. In this paper we give an overview of distributed data structures in general and Multipol in particular. We identify the common performance issues that arise and describe the implementation techniques used in Multipol.

Multipol is designed to help programmers write for irregular applications such as discrete simulation, symbolic computation, and search problems. These applications typically contain conditional control constructs and irregular, non-array data structures such as graphs or unstructured grids, which make the amount of computation in the program data dependent, leading to dynamic scheduling and load balance requirements. They also produce unpredictable communication patterns, for which runtime techniques must be used for enhancing locality and reducing communication costs. The Multipol data structures and runtime system provide such support.

*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 as a Research Initiation Award (number CCR-9210260), and as an Infrastructure Grant (number 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.

One of the key problems in any library effort of this kind is portability. Our primary targets are distributed memory machines such as Thinking Machines CM5, IBM SP1, Cray T3D and Intel Paragon, as well as networks of workstations. While at a functional level these platforms are very similar, the performance characteristics vary significantly. All of the machines have higher costs for accessing non-local memory than remote memory, whether this is done in hardware or in software, but the relative speeds of computation, the startup overhead of communication, the latency and the observed bandwidth all vary. The interface design and implementation of Multipol structures are aimed at coping with communication overhead and latency.

Multipol is implemented on a novel runtime layer that provides mechanisms for lightweight threading and communication without enforcing fixed scheduling policies or particular memory models. The threading model provides for fixed duration threads, which obviates the need for expensive thread preemption. The communication subsystem does message aggregation to amortize the communication startup on machines like workstation networks that have high message startup costs.

The approach taken by Multipol offers the following advantages. First, the data structures provide high level programming abstractions that can be reused across applications. The abstractions also help hide implementation details, which can be fine-tuned to match the communication and synchronization needs. Finally, the library defines a set of interface ideas, such as split-phase operations, and underlying implementation infrastructure that make it easily extendible to a wider class of data structures and applications.

The remainder of the paper is organized as follows. Section 2 gives an overview of the data structures in Multipol, describing their functionality and the applications that may use them. Section 3 introduces the common issues in designing high-performance, portable data structures. Section 4 sketches the portable runtime system on which Multipol is built, and Section 5 describes a few of the Multipol data structure implementations in greater detail. Sections 6 gives an overview of related work and in Section 7 we draw conclusions from our experience.

2 Overview of Data Structures

This section gives an overview of the Multipol data structures and the applications that use them. The data structures fall into three categories: *spatial structures* represent a collection of objects in a physical system; *association structures* represent a collection of key/value pairs, typically to store a set of computed values; *scheduling structures* represent a collection of tasks to be performed.

2.1 Spatial Structures

Many scientific applications simulate physical systems in which a set of objects are spread out in a spatial domain. In most cases, objects in the system have a physical coordinate, although in some cases objects are connected though logical channels without reference to a global coordination system. Spatial structures have natural locality, since objects nearby tend to have more effect than objects further away, and taking advantage of this locality proves critical to performance.

Bipartite Graph: Some algorithms gain efficiency and accuracy by using unstructured meshes, which place mesh points at physically significant locations, such as the face of a tetrahedron [Mad92], and use

more refined meshes in areas of greatest activity. The neighbor relation is still important, but the connectivity pattern is irregular. In an electromagnetics application called EM3D, the mesh is irregular but static [CDG+93]. In this case there are two set of mesh points, one with electric field values and one with magnetic field values. The graph is bipartite, and computation proceeds by reading values from one set of vertices and updating the neighboring values in the other set.

Oct-tree: Although locality is evident in most physical simulations, the effects between entities may not be limited to neighboring values. In the cell simulation, a $O(n^2)$ algorithm is used to compute the effects of cells on each other. In systems with gravitational or similar forces between entities, hierarchical data structures can be used to improve performance. The oct-tree is one example of such a structure. A three dimensional space is partitioned into eight equally sized cubes, and the process repeated on each sub-cube with more than one object, until each cube has at most one object [BH86]. The oct-tree is the hierarchy created from this decomposition.

Event Graph: The event graph provides flow-controlled, in-order message delivery in a static network. The nodes of the event graph represent computational processes that send and receive messages along the edges. The messages are called events because their order with respect to the same edge should be preserved. The event graph is used in a discrete event timing simulator called SWEC [LKMS91]. Discrete event simulations like SWEC can be parallelized using either optimistic or conservative scheduling. We have parallelized SWEC using optimistic scheduling [WY93] and focus here on the conservative simulation.

2.2 Association Structures

Association structures store a set of values without any relationship to a physical domain. These are common in search problems and are often built as dictionaries that store key/value pairs. There is no *a priori* notion of spatial locality in these data structures, but the data structures sometimes use caching to provide temporal locality for applications that can exploit it.

Hash Table: A common parallel programming paradigm requires that a set of processors cooperatively access a sparse address space. For boolean function manipulation, for example, this address space contains functions in the form of DAGs [BRB90]. For search problems, the processors generate sets of states in the space as they iterate over a subset of the space. The hash table provides a sparse address space distributed across the processor, with a well-behaved hash function providing a natural load balance. Each processor owns a local portion of the space, which it accesses efficiently, as well as a port into the global structure.

Trie: Memoizing intermediate solutions may speed up many types of search problems. The phylogeny problem[Fel82] uses this technique for determining the evolutionary history for a set of species based on their characters. The kernel of this algorithm explores the space of all subsets of characters searching for incompatibilities with a proposed evolutionary history. The search space may be pruned by ignoring the character sets that are either subsets of character sets already found compatible or supersets of character sets already found incompatible. Storing character sets in a trie provides an efficient implementation of the subset detection operation.

Replicated List: The replicated list provides efficient support for iterating over a set of objects, provided the elements in the list have small size, such as pointers. In particular, the notion of *reordered* and *incomplete* iterations are naturally supported. This involves application knowledge of the locality of the iteration space. E.g., if data elements for some iterations are remote, the application may choose to issue a prefetch and/or skip over these elements. The replicated list is built on top of a generic *object layer*, which implements a distributed shared memory by supporting variable size objects and a user-defined meaning of object consistency. It is the software generalization of shared memory protocols on Tempest [RLW94].

The replicated list has been used in the Gröbner basis algorithm, and applies to Knuth-Bendix style theorem provers [YG92]. Other potential uses are in Barnes-Hut type particle simulations. Critical to the use of the replicated list is understanding the commutativity and idempotence of the operations. While it is widely recognized that increased application control of shared memory operations improves performance [GLL+90, LLG+90], we have found that control at the read/write level of abstraction is insufficient.

2.3 Scheduling Structures

Task queue: Many task-parallel MIMD applications depend on some implementation of a distributed task pool to explore a task tree (or task DAG in the more general case). The task queue is an example of a generic interface providing basic functionality of this nature. This abstraction has been used in a symbolic algebra algorithm, a tridiagonal eigenproblem solver, a sparse Cholesky factorization program, and a computational biology program.

3 Performance Issues

The overheads of parallelization comprise the time the processors spend doing useless computation, i.e., computation that is not required by the sequential implementation, the time they spend in communication, and the time they are idle. Each type of overhead is reduced in Multipol code using a combination of the techniques outlined below.

3.1 Latency Masking

The latency of remote operations can cause idle time if the processor waits for the operation to complete. A remote operation simply reads or writes remote memory or executes a small remote procedure, for example, a lock acquisition. Thus, the term *latency* refers to both the message transit time and the time required for remote processing. The remote computation time is not necessarily overhead, but time spent waiting for completion is. The total latency can be quite large when the network is slow, when the application has highly irregular communication patterns that make it impossible to make optimal scheduling decisions, or when the remote requests require nontrivial computation time.

Techniques such as pipelining remote operations and multithreading can be used to hide latency. Even on a machines like the CM-5 with relatively low communication latency, the benefits from message overlap are noticeable: message pipelining of simple remote read and write operations can save as much as 30% [KY94] and overlap of higher level operations in the Gröbner basis application saves about 10%.

On workstation networks with longer hardware latencies and expensive remote message handlers, the savings should be even higher.

The latency hiding techniques require the operations be nonblocking, or *split-phase*. In Multipol, operations that would normally be long-running with unpredictable delay are divided into separate fixed-length threads. Multipol operations execute local computation and may initiate remote communication, but they never wait for remote computation to complete. Instead, long-running operations take a synchronization counter as an argument, which the caller can use to determine if the operation has completed. This leads to a relaxed consistency model for the data types, which is weaker than either sequential consistency [Lam79] or linearizability [HW90]. A operation completes sometime between the initiation and synchronization point, but no other ordering is guaranteed.

Several applications can take advantage of relaxed consistency models. For bulk-synchronous problems such as EM3D [CDG+93], cell simulation [Ste94], and n-body solvers, data structure updates are delayed until the end of a computation phase, at which point all processors wait for all updates to complete. In Gröbner basis and the phylogeny application, which have characteristics of a search problem, the set of "found" values are stored in a lazily updated structure.

3.2 Locality

Locality is crucial when communication cost is large. One way to improve locality is to reduce the volume of communication. Techniques for reducing communication can be either static or dynamic. Static techniques include partitioning, which attempts to divide up the data set into loosely dependent partitions among the processors, and replication, which keeps a copy of mostly-read data on each processor. Dynamic techniques include caching, which maintain multiple copies of the data depending on the its runtime usage. For these techniques, relaxed consistency may be used to further reduce communication.

Many applications can take advantage of these relaxed data structures because there is no strict ordering on updates. In the phylogeny application and Gröbner basis problem, not only are updates to the global set of results lazy, but each processor keeps partially completed cached copies of this set. This yields a correct, albeit different, execution than the sequential program [CY93, Jon94].

3.3 Communication Cost Reduction

Some communication cannot be avoided, but its cost can be reduced by minimizing the number of messages (as opposed to the volume) and by using less expensive unacknowledged messages. For machines like the Paragon and workstation networks, which have high communication start-up (known as α in the $\alpha - \beta$ cost model), the former is very important. Many small messages are aggregated into one large physical message to amortize the overhead. Several other systems, including Parti and LPARX, also use message aggregation. Even for machines such as CM5, which have small hardware packets and therefore a nearly fixed overhead per word, it may still be advantageous to aggregate messages to reduce the amount of flow-control communication for sending arbitrary-sized messages which cannot fit into a machine packet. Message aggregation can be performed statically by the programmer, or dynamically by the runtime system.

The second technique for reducing communication cost is to avoid acknowledgement traffic. Acknowledgements may consume a significant fraction of available bandwidth when the messages are small. In

the hash table, a factor of 2 in performance was gained when split-phase inserts with acknowledgements were replaced by batches of inserts followed by periodic global synchronization points.

3.4 Multi-ported Structures

In addition to communication overhead, many parallel applications lose performance on the local computation. Languages that support a global view of distributed data structures, for example, may incur costs from translating global indices into local ones [Ste94] or from checking whether a possibly remote address is actually local [CDG+93]. Message passing models in which objects cannot span processor boundaries avoid these overheads, but lose the ability to form abstractions across processors. We propose a compromise, which each data structure has both a local view, which refers to the sub-object that is on the processor, and the global view, which refers to the entire distributed data structures. For example, many of the data structures allow for iteration over the local components of the object, and for operations that modify or observe only at the local data. In this sense, the data structures are *multi-ported*: each processor has its own fast handle to a structure, while access to the global structure is also permitted.

3.5 Load Balance

Load balance of data structures requires that the data be spread evenly among the processors to avoid hot spots. Scheduling involves the assignment of tasks to processor to keep all processors busy. There is typically a trade-off between locality and load balance which can be resolved used either static or dynamic techniques. For data structures with high remote access costs, static load balance and scheduling techniques such as the owner-compute rule can be used to reduce communication. For data structures with little remote access cost, dynamic strategies such as randomization can be used to increase processor efficiency. A mixed strategy where dynamic scheduling is combined with locality considerations is also possible.

4 The Runtime Layer

A Multipol program consists of a collection of threads running on each processor, where the number of physical processors is exposed so that the programmer can optimize for locality. Multipol threads serve two purposes. They are invoked locally to hide the latency of split-phase operations and can also be invoked remotely to perform asynchronous communication. The Multipol runtime system provides support for thread management, as well as a global address space spanning the local memory of all processors. In this section, we describe the runtime support in Multipol.

4.1 Overview of Multipol threads

Multipol threads are designed to facilitate the composition of multiple data structures, and the porting of the runtime system. This section describes the features of Multipol threads and explains our design decisions.

Multipol threads run *atomically* to completion without preemption or suspension. Atomicity of thread execution reduces the amount of locking required, and makes it easy to implement common read-modify-write operations. Since threads are not preempted, spinning is prohibited – to suspend

a computation awaiting the result of a long latency operation, the thread that issues the operation explicitly creates a *continuation* and passes the required state. The issuing thread then terminates, and its continuation thread can be scheduled to resume the computation when the result becomes available. Synchronization between the continuation thread and the completion of the operation is achieved by waiting for a `counter` to exceed a given value.

Because the programmer explicitly specifies the state to be passed to the continuation, there is no need to implement a machine dependent thread package for saving the processor state and managing separate stacks. Our approach improves the portability of the runtime system, and may have lower thread overheads for machines with large processor states.

The runtime system provides a two-level scheduling interface for threads. The programmer can write custom schedulers to schedule the data structure or application threads. The runtime system, for example, uses a FIFO scheduler for interprocessor communication, and applications such as discrete event simulators can have their own priority based schedulers. The top-level system scheduler guarantees that each custom scheduler is called once within finite time, and the frequency of calls can be configured by the programmer.

The scheduling interface localizes scheduling decisions to the custom schedulers, which can be individually fine-tuned for performance. It also facilitates the composition of data structures, or the addition of new runtime support. The scheduling policy used by one data structure can be changed without introducing anomalies, such as unexpected livelock or deadlock, into other parts of the program.

The Multipol threads are designed for direct programming, in contrast to compiler-controlled threads such as TAM [CSS+91], in that Multipol provides more flexibility such as arbitrary size threads and custom schedulers. A set of macros can be used to facilitate programming. These macros make the Multipol programs resemble sequential programs with blocking operations (as opposed to thread continuations with split-phase operations).

4.2 The Multipol communication primitives

The runtime system supports two types of communication primitives: remote thread invocation and bulk accesses of the global memory. A thread may be invoked on a remote processor to perform asynchronous communication, such as requesting remote data, or to generate more computation, such as dynamically assigning work to processors. Invoking a remote thread is a non-blocking operation which returns immediately, and its completion guarantees that the remote thread will be invoked in finite time. The programmer can also use bulk, unbuffered *put* and *get* primitives to access remote data. The put and get operations are split-phase operations which use a counter to synchronize the calling computation when all data arrive at the destination.

The runtime system aggregates messages to improve communication efficiency for programs that generate many small, asynchronous messages. These messages are accumulated into large physical messages to better amortize the communication start-up overhead. Experiments with SWEC show that message aggregation can reduce the running time by as much as 50% on machines such as the IBM SP1.

4.3 Porting the Multipol runtime system

Porting the runtime system consists of defining the machine dependent constants (such as the number of processors available), installing custom schedulers for the architecture, and implementing the network primitives. Two principal network primitives need to be ported: the barrier synchronization of all processors (blocking), and a nonblocking *store* primitive. The store primitive transfers a block of data from a local buffer to a remote address and invokes a remote handler upon completion. Store is nonblocking, returning when the operation can be guaranteed to complete in finite time. The data is unbuffered, and the primitive takes a counter as input to inform the sender when the local buffer can be reused. Ports of the runtime system to active message and send/receive architectures exist.

5 The Data Structures

This section describes a few of the Multipol data structures, their interfaces, implementations, and performance characteristics.

5.1 Hash Table

The Multipol hash table provides three general types of operations – synchronous, distributed and local. The synchronous operations `create` and `destroy` must be called on all processors. The distributed operations (`insert`, `lookup`, `testNset`) are called locally on each processor and may access the distributed structure through the local port. Strictly local operations allow the processor to iterate over, count, and clear its local portion of the hash table. Note that the local operations perform no synchronization, and their interaction with concurrent distributed operations is undefined. Thus, the multi-ported nature of the hash table give good performance at a cost of weaker semantics.

The hash table distributes the buckets over the processors, and uses chaining to resolve collisions. `Insert` and `lookup` are implemented using an owner-computes rule – keys and data are sent to the processor managing the bucket. The hash table uses the runtime primitives to support split-phase operations. For `insert`, `lookup` and `testNset`, the user may synchronize a thread on the result by specifying a counter. `Insert` become unacknowledged if the counter is omitted – `sync` may then be used to detect termination of all outstanding operations for bulk synchronous algorithms.

The hash table demonstrates two performance techniques in Multipol – latency masking and used of one-way communication. As an example, we coded a search problem which finds solutions for a simple puzzle. The algorithm iterates between a current set of states and a new set generated by valid moves. For the problem size we considered, about 100,000 positions are explored.

We compare three implementations. In the naive blocking algorithm, each processor iterates over the positions in its local set and performs a blocking `insert` into the hash table. On a 32-node CM5, this version takes 27.9 seconds. To pipeline the inserts, the processors wait on the synchronization counters at the end of the iteration. With pipelining, the running time is reduced to 16.4 seconds. A further optimization is to eliminate acknowledgement traffic by using a global synchronization point at the end of each iteration. The search now takes only 11.0 seconds. See Figure 3 for a complete speedup curve on the search problem.

5.2 Replicated list

The replicated list, like the hash table, holds a set of values for search-like applications. The creation and destruction routines are similar to those on the hash table, and there are append and remove operations, which are long-latency operations similar to hash table `insert`. Like the hash table, the list also has an iterator for traversing the local elements. The difference is that the local iterator on the hash table produces disjoint subsets of the elements, whereas the replicated list iterator does not. In the replicated list, each local iteration produces a subset of all the elements, whose union across processors is the complete set.

Two design decisions improve performance of the list. First, the list is replicated fairly aggressively: it is consistent except when a processor has an outstanding split-phase operation. Complete replication is acceptable when the list is used to store only global pointers or object ID's (as against actual values). In the Gröbner basis application, for example, the list is use to store the final answer, which is small relative to the total data that is accessed during the computation. Second, the list sits on top of a general distributed object library (described below) which manages the global address space. Thus, while the view of the list of object identifiers is usually consistent, the cached objects associated with them are communicated lazily. The list is designed for a relatively smaller set of large objects.

5.3 Object layer

The object layer is a general abstraction for caching contiguous objects. Because of its generality, the interface is rather large, so only a few of the operation are outlined here. While the set of routines is rather complicated, especially the sequencing requirements between operations, this must be compared with the even more complicated synchronization protocols that are hidden within the layer. The complexity of the interface comes from its generality and from our desire to provide useful mechanisms rather than fixed memory consistency policies. We expect most uses of this data structure to be hidden within another distributed structure like the replicated list, hash table, or trie.

`Mol_create` creates a new object and registers it into the object space. A unique ID for this object (recognized eventually on all processors) is created and returned. This is a long running routine, so a counter is taken as input. Objects are read locally using `Mol_read_acquire` and `Mol_read_release`. This style is needed for atomic version upgrades and garbage collection. `Mol_valid_sync` is a long-running operation to validate a given object, which may be stale or unavailable. `Mol_modify` and `Mol_destroy` are (long-running) operations to modify or delete an object. They bump input counters when the remote operation completes.

High read throughput is achieved by replicating the object identifiers aggressively. However, the application dictates how aggressively the actual objects are cached. The Gröbner basis application makes use of such flexibility by working with a subset of the data and allowing the copies to become out of date. One disadvantage of providing this flexibility in software is that the overhead of a cache hit is prohibitive for small objects. The application writer must evaluate this cost for the object size of interest.

5.4 Event Graph

The event graph is a data structure with some spatial locality from the application domain. In addition to basic construction and destruction functions, the event graph has operations to send an event along a set of outgoing edges, receive events from an incoming edge, and a global *snapshot* function, which presents a consistent global view of the data structure to the program. The operations all have split-phase interfaces for latency masking.

The events are stored at the receiving processors, so that all `receive` operations can be handled quickly. To reduce the communication and memory requirements, we combine events that are sent to different graph nodes on the same processor. The space for caching is pre-allocated and the number of readers is known from the static graph structure. Experiments with SWEC timing level circuit simulator show that 5% to 20% of the events can be found in the cache for a small circuit.

Sending an event requires that the buffer space be available on all fanout edges. We replicate the flow-control state of the buffer at the receiving and the broadcasting processors. The replicas are conservative summaries of the true fanout buffer states, so that most events can proceed without requiring flow control. Experiments with SWEC show that as much as 64% to 98% of the sending operations can complete without blocking, depending on the size of the buffer space. The amount of memory allocated to the event graph can be adjusted to fit different machines or applications. Since the event graph is the only distributed data structure in SWEC, allocating more memory generally leads to better performance. Experiments show that the running time can be reduced by 30% when the edges can hold 64 instead of 16 outstanding events.

After a processor accepts an event, flow control messages must be sent to free up buffer space. Rather than sending a flow-control message for each event, we provide an interface whereby the programmer can explicitly issue an update after some number of events are received. By aggregating such update messages, their number can be reduced by 10% to 50%.

By combining all these techniques, the conservative parallel SWEC kernel simulating a small circuit with 129 subcircuits shows speedup up to 3.65 and 5.30 on the 8-node Paragon and SP1 respectively, and 9.90 on a 32-node CM5 (See Figure 4 for the speedup curves). These numbers are based on a simulation, where it is easier to control and measure the effects of individual optimizations. However, the data structure was motivated by a parallel implementation of SWEC using optimistic scheduling, which shows a speedup of 55 for a large circuit on a 64-node CM5[WY93].

5.5 Oct-Tree

To provide an efficient implementation of an oct-tree, two important issues need to addressed: how to spread the oct-tree across processors to obtain good load balance without sacrificing locality, and how to decrease the cost of accessing remote nodes.

For good load balancing, the tree needs to be partitioned across processors so that all processors compute equal numbers of inter-particle interactions. To preserve locality, it is beneficial to partition the tree so that each processor gets complete sub-trees. These two goals unfortunately conflict. Warren and Salmon describe an algorithm for mapping an oct-tree across processors. Our goal in Multipol is to decrease the cost of the access operations for a given layout. In particular, we decrease the number of

remote accesses as well as decrease the cost of a remote access by reordering the inter-particle interactions. The standard per-processor algorithm is as follows:

```
for each particle P owned by me
    for all particles R that P needs to interact with
        Interact P with R
```

We reverse the loops, which has the advantage that each remote particle R is fetch once and used multiple times. However, there are some important considerations in obtaining an efficient implementation of this high level description. R should be fetched only if there is some local particle P that requires the remote value. Storing the local particles in a multi-list data structure allows us to locate all the particles that need to interact with R without having to search through the entire list of local particles.

Our approach guarantees that the number of remote accesses made is the minimum required for the entire computation. We also reduce the cost of a remote accesses by making the operation split-phase, and pre-fetching across iterations.

5.6 Task Queue

The task queue provides dynamic load balancing of a set of value, which are typically structures used to identify tasks. The three main functions of the task queue object are to inject a new task, extract an existing task, and detect quiescence of the system. From the performance perspective, a more important internal functionality is load balancing. In addition to creation and destruction operations, the task queue has functions to insert and remove tasks. The removal operation is a long-running operation with a synchronization counter, allowing the processor to perform other work if the requested task is unavailable. The only other long-running operation is for termination detection, since this requires global agreement among the processors.

There is an enormous literature on dynamic load balancing and we have two implementations: one that uses a simple randomized load balancing scheme [CRY94] and another that uses heuristics for locality along with round-robin task pushing. Both of these work well for the applications in which locality is not a major concern, including Gröbner basis, the phylogeny problem, and a symmetric eigenvalue code [Sin]. However, more sophistication in task assignment will likely be required for other applications, that have partitioned data sets.

6 Related Work

Several research efforts are related to ours: general libraries and systems software, application specific libraries, customized shared memory software, and systems for performance portability.

Libraries and software systems. Recent research has yielded a variety of runtime support like the Concert system [KC93], the Chare kernel [SK91, KK93], and the compiler-controlled threaded abstract machine [CSS+91], and programming languages like COOL [CGH], Concurrent Aggregates [And93], pSather [FLR92] and pC++ [BBG+93]. As language-based approaches, they encapsulate default policies for data and task assignment. When tasks involve dynamic data structures, execution time and data access patterns cannot be predicted by the compiler. Since no policy is likely to be universally applicable,

user overrides are also permitted. Thus, these systems provide mechanisms but no general understanding of application performance with respect to data structure design and implementation, or how to address these issues in a new application.

Another thread of work has been the development of application-specific distributed data structures, including irregular grids [BSS91], the LPARX library for writing adaptive mesh codes [Bad91], B-trees [WW90], sets [Dal86] and oct-trees [WS93]. Some early Fortran versions of irregular grid code can be improved by message aggregation using the PARTI library [BSS91]. In contrast, we look at significantly more irregular, dynamic, and even non-deterministic problems.

Shared memory semantics. The programmer's most intuitive notion of shard memory is sequential consistency, which specifies that accesses to shared memory are an arbitrary merge of accesses from all the processors, but each has atomic effect [Lam79]. Release consistency relaxes this by permitting limited reordering of the processor order of accesses, with hardware instructions to commit all outstanding writes when the program needs this property to proceed [LLG+90]. Adve and Hill have designed a formalism and expressed these and other variants of shared access specification [AH90]. Weihl's thesis makes a case for reordering accesses more aggressively than possible in hardware by using more application information [Wei88]. We have carried this idea even further in the Gröbner basis program [CY93, CY94], the Oct-tree code, and the Phylogeny program.

Portable parallel programming. Alverson [AN93] introduced the notion of effectively portability and developed the runtime system Chameleon, which separates the abstract interface of the data structures from their implementation for portability. Crowl [CL94] developed the programming language Natasha, which supports "forall" like control abstractions and multiple implementations of the abstractions. The programmer explicitly selects the implementation using annotations in the source programs. Neither Alverson nor Crowl addressed the composition of multiple data structures. Brewer [Bre94] proposed a systematic approach to tuning parallel programs based on statistical sampling. However, his work addressed the sequential composition of algorithms, but not the parallel composition of irregular data structures.

7 Conclusions

The Multipol library fills the gap in the parallel software tools for programming irregular applications on distributed memory machines. In this paper we have identified some of the primary issues in the design of distributed data structures – namely locality, load-balance, latency hiding, and communication elimination. All of these performance issues are complicated by the variations across architectures, which make it difficult to effectively manage resources. This lead to the development of portable runtime layer and data structures that tune themselves to the architecture.

The irregular applications described here represent some of the more challenging problems for parallelism. The library approach proves to be a good compromise between hand-coded applications that are machine-specific and pure language and compiler approaches, which give too little control to the application programmer. The split-phase interfaces are a concession to performance demands. While they complicate the interface from the client's perspective, they significantly improve performance on

distributed memory machines. The use of one-way communication eliminates acknowledgement traffic and is a significant performance enhancement for data structures with small messages. The multi-ported aspect of the structures allows the users to switch between global and local views, providing the abstraction of the former and performance of the latter.

References

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering–a new definition. In *17th International Symposium on Computer Architecture*, April 1990.

[AN93] G.A. Alverson and D. Notkin. Program structuring for effective parallel portability. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[And93] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.

[Bad91] S. B. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. Stat. Comput.*, 12(1):145–157, 1991.

[BBG+93] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Maloney, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel system. In *Supercomputing '93*, pages 588–597, Portland, Oregon, November 1993.

[BH86] J. Barnes and P. Hut. A hierarchical o(nlogn) force-calculation algorithm. *Nature*, 324:446, 1986.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, Orlando, FL, June 1990.

[Bre94] Eric Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization.* PhD thesis, Massachusetts Institute of Technology, September 1994.

[BSS91] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory multiprocessors. *Concurrenty: Practice and Experience*, pages 159–178, June 1991.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing '93*, pages 262–273, Portland, Oregon, November 1993.

[CGH] Rohit Chandra, Anoop Gupta, and John L. Hennesy. Data locality and load balancing in COOL. submitted for publication.

[CL94] Lawrence A. Crowl and Thomas J. LeBlanc. Parallel programming with control abstraction. *ACM Transactions on Programming Languages and Systems*, 1994.

[CRY94] Soumen Chakrabarti, Abhiram Ranade, and Katherine Yelick. Randomized load balancing for tree-structured computation. In *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[CSS+91] 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.

[CY93] Soumen Chakrabarti and Katherine Yelick. On the correctness of a distributed memory Gröbner basis computation. In *Rewriting Techniques and Applications*, Montreal, Canada, June 1993.

[CY94] Soumen Chakrabarti and Katherine Yelick. Distributed data structures and algorithms for Gröbner basis computation. *Lisp and Symbolic Computation*, 1994. To appear.

[Dal86] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* PhD thesis, California Institute of Technology, Pasadena, California, March 1986.

[Fel82] J. Felsenstein. Numerical methods for inferring evolutionary trees. *Q. Rev. Biol*, 57:379–404, 1982.

[FLR92] J.A. Feldman, C.C. Lim, and T. Rauber. The shared-memory language psather on a distributed-memory multiprocessor. In *Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, Boulder, CO, September 1992.

[GLL+90] Kaourosh Gharachorloo, Daniel Lenoski, James Laudon, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, pages 463–492, July 1990. A preliminary version appeared in the proceedings of the 14th ACM Symposium on Principles of Programming Languages, 1987, under the title: *Axioms for concurrent objects.*

[Jon94] Jeff Jones. Parallelizing the phylogeny problem. Master's thesis, University of California, Berkeley, Computer Science Division, December 1994.

[KC93] Vijay Karamcheti and Andrew Chien. Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Supercomputing'93*, Portland, Oregon, November 1993.

[KK93] L. V. Kalé and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System based on C++. Technical Report UIUCDCS-R-93-1796, University of Illinois, Urbana, IL, March 1993. Also in OOPSLA'93.

[KY94] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel spmd programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LKMS91] S. Lin, E. Kuh, and M. Marek-Sadowska. SWEC: A stepwise equivalent conductance simulator for cmos vlsi circuits. In *Proc. of European Design Automation conference*, February 1991.

[LLG+90] Daniel Lenoski, James Laudon, Kaourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th International Symposium on Computer Architecture*, pages 148–159, 1990.

[Mad92] Niel K. Madsen. Divergence preserving discrete surface integral methods for maxwell's curl equations using non-orthogonal unstructured grids. Technical Report 92.04, RIACS, February 1992.

[RLW94] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: User-level shared memory. In *21st International Symposium on Computer Architecture*, April 1994.

[Sin] Dhillon Inderjit Singh. The bisection eigenvalue algorithm. Unpublished manuscript.

[SK91] 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.

[SL93] Daniel J. Scales and Monica S. Lam. A flexible shared memory system for distributed memory machines. Unpublished manuscript, 1993.

[Ste94] Stephen Steinberg. Parallelizing a cell simulation: Analysis, abstraction, and portability. Master's thesis, University of California, Berkeley, CA, December 1994.

[Wei88] William E. Weihl. Commutativity-based concurrency control for abstract data types. Technical Report MIT/LCS/TM-367, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[WS93] M.S. Warren and J.K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93*, pages 12–21, Portland, Oregon, November 1993.

[WW90] William E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the Symposium on Parallel and Distributed Processing*, December 1990.

[WY93] Chih-Po Wen and Katherine Yelick. Parallel timing simulation on a distributed memory multiprocessor. In *International Conference on CAD*, Santa Clara, CA, November 1993.

[YG92] Katherine A. Yelick and Steven J. Garland. A parallel completion procedure for term rewriting systems. In *Conference on Automated Deduction*, Saratoga Springs, NY, 1992.

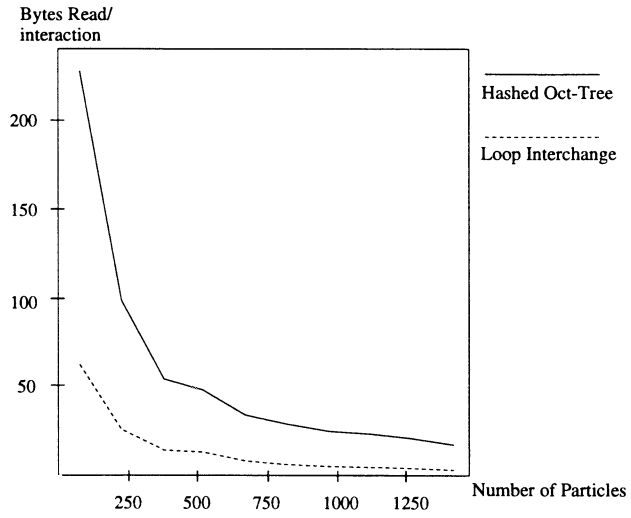


Figure 1: Comparison of the Hashed Oct-Tree approach suggested by Warren and Salmon with our Loop interchange approach based on bytes read per inter-particle interaction.

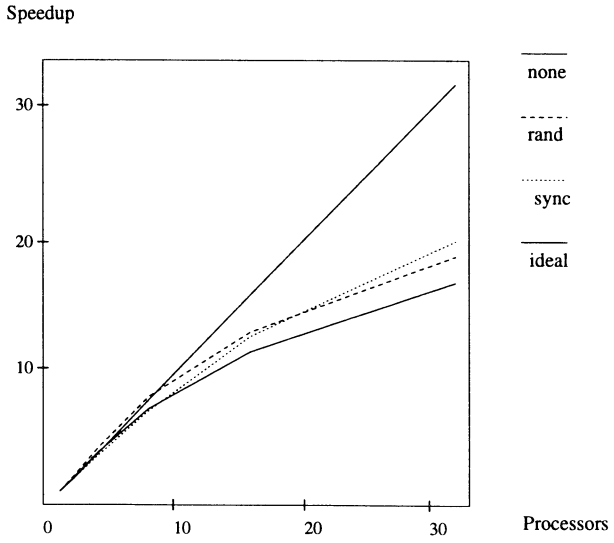


Figure 2: Speedups for the Phylogeny problem using the three implementations of the Trie.

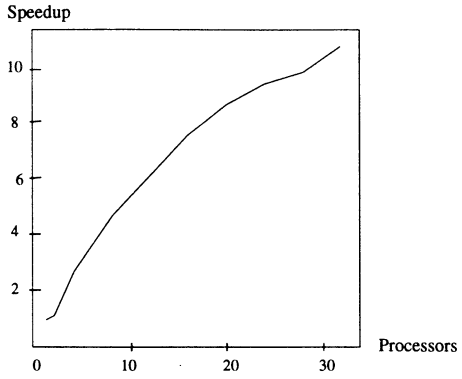


Figure 3: Speedups for the triangle puzzle using the hash table.

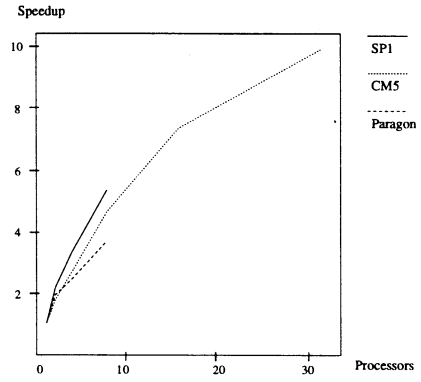


Figure 4: Speedups for the parallel SWEC kernel using the event graph.

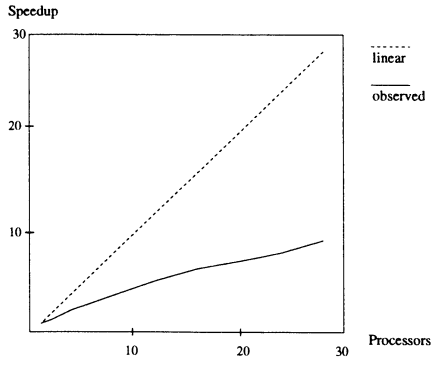
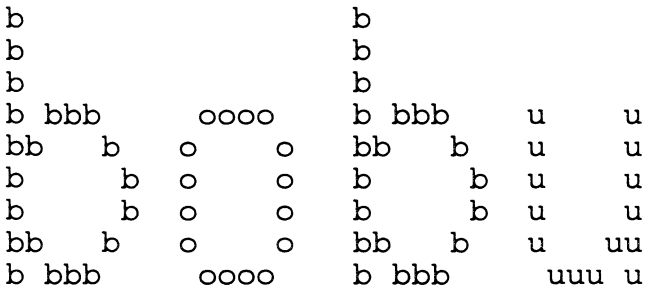
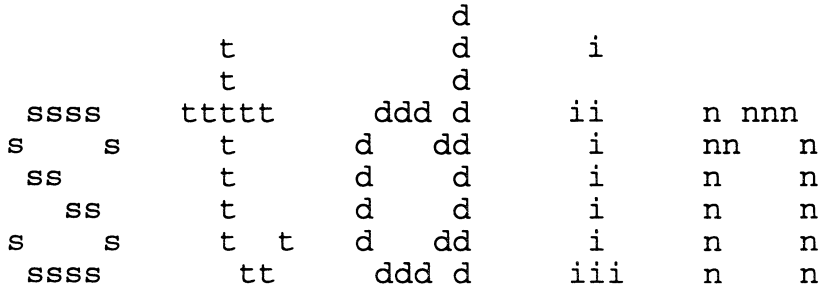
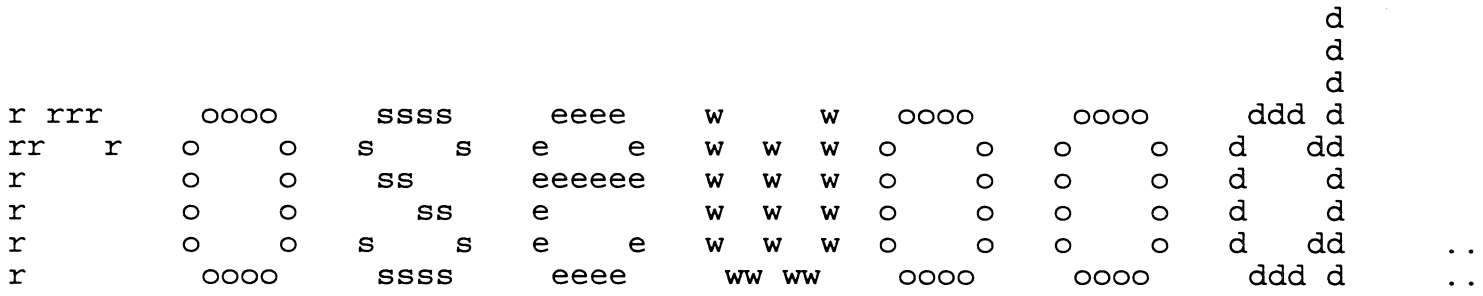


Figure 5: Speedups for the eigenvalue problem using the task queue.







Job: stdin
Date: Tue Nov 14 09:40:39 1995