Draft: To appear in Parallel Symbolic Languages and Systems, October 1995

Parallel Data Structures for Symbolic Computation

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

Abstract

Symbolic applications often require dynamic irregular data structures, such as linked lists, unbalanced trees, and graphs, and they exhibit unpredictable computational patterns that lead to asynchronous communication and load imbalance when parallelized. In this paper we describe several symbolic applications and their parallelizations. The main problem in parallelization of each application was to replace the primary data structures with parallel versions that allow for high throughput, low latency access. In each case there are two problems to be solved: load balancing the parallel computation and sharing information about the solution as it is being constructed. The first problem is typically solved using a scheduling data structure, a stack, queue, or priority queue in sequential programs. The difficulty in parallelizing these structure is the trade-off between locality and load balancing: aggressive load balancing can lead to poor locality. The second problem of storing the solution depends much more on the type of solution, but range from simple scalar values to sets or tables. These structures use partitioning, full replication, or dynamic caching in their parallelizations.

In sequential programming environments, common data structures are often provided through reusable libraries. We have built a parallel analog to these libraries, called Multipol. Multipol support irregular and asynchronous applications, including symbolic applications, discrete event simulation, and adaptive algorithms. The performance issues in Multipol include masking remote latency, elimination of communication, load balance, performance portability across machines, and local node performance.

1 Introduction

This paper reports on several case studies of parallel symbolic applications and the parallel data structures within them. It also describes systems support in the form of a parallel data structure library that makes such applications easier to develop. In our experience, each application contains a small number of data structures that need to be replaced when developing a parallel version; in symbolic applications, these often fall into two categories. The first category is scheduling structures such as stacks, queues, or priority queues. These structures hold data items that represent the set of tasks to be computed, so

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-92-C-0026 and F30602-95-C-0136, by the Department of Energy grant DE-FG03-94ER25206, and by the National Science Foundation grants CCR-9210260, CDA-8722788, and CDA-9401156. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

the key issue in parallelization is to load balance these tasks without destroying locality properties or violating dependencies between tasks that lead to incorrect semantics or unnecessary work. The second class of data structures hold shared information about the current approximation to or partial version of the final solution. In search problems, for example, this might be a representation of the best solution so far or cut-off values to prune the search space.

In contrast to these symbolic applications, most scientific and engineering applications are physical simulations in which at least one of the key data structures is the representation of the physical domain. Some examples include fluid flow simulation with a regular or unstructured mesh placed over the fluid, n-body simulations with a set of particles in physical space, or event-driven simulation with loosely-coupled processes are connected by logical channels over which events are passed. Even the most irregular simulations problems have one important advantage over symbolic applications: locality properties in the physical domain domain lead to locality in the parallel implementation; entities near each other in space are more likely to affect each other than ones far away. In symbolic applications, locality may become a design concern, but does not usually arise from a physical notion of locality.

In this paper, we report on our experience with several parallel applications on both shared and distributed memory machines. First, we show that even the challenging class of symbolic applications can result in effective use of parallel machines, including distributed memory multiprocessors. Since one of the key problems is parallel data structure development, we have developed a data structure library called Multipol. We give an overview of some of the Multipol structures and the underlying system concepts that enable portability and high performance.

2 The Applications

In this section we give a brief description of our symbolic applications and the high level data structures that are replaced during parallelization. In each case we identify two data structure requirements: a *scheduling* structure to hold the set of tasks being generated and a *solution* structure to hold an approximation to or partial version of the final result. In two cases, the same data structure will play both roles. The applications were not chosen for their similarity, but a common property is that all of the solution structures have some type of monotonicity that allows out-of-date or incomplete versions to be used.

For precise problem statements, alternate approaches, related work and complete descriptions of our implementations, we refer the reader to more extensive papers. Most of the applications were written for distributed memory multiprocessors, but as a means of comparison, we also discuss two shared memory applications: the Knuth-Bendix procedure and term matching.

2.1 Knuth-Bendix

The Knuth-Bendix procedure is used in automatic theorem proving systems based on ordered equations called *rewrite rules*. Given a set of rewrite rules, the procedure computes a new set that is in some sense *complete*, which allows for easy and efficient proofs in the resulting system [KB70]. The procedure may also fail to terminate or may halt unsuccessfully, although the reasons for these behaviors and techniques to help avoid them are beyond the scope of this discussion. The computation involves addition of new rewrite rules, called *critical pairs*, and the simplification of existing rules. New rules are discovered by considering each pair of existing rules: a potential new rule is computed and simplified using existing rules, and if it survives the simplification process it is added to the set.

The creation and simplification of new rules constitutes the bulk of the work, so the data structure that holds pairs of rules is the primary scheduling structure in the parallel implementation [YG92]. The ordering of pairs within this scheduling queue is quite flexible, although a fairness must be guaranteed (pairs cannot be left in the queue forever) and if some rule pairs are favored over others, unnecessary work can be avoided. The parallel solution is a task queue: each processor has its own queue of pairs, ordered by heuristics, and when a processor runs out of tasks it steals them from another processor's queue [RV89]. Even on a four processor shared memory machine, a centralized task queue proved to be a bottleneck, so a distributed version is used.

As the computation proceeds, the set of rewrite rules progresses toward completeness, so the set itself is the approximate solution. The elements of the rewrite rule set need to satisfy a property of pairwise irreducibility, which is stronger than the usual uniqueness property on elements of a set, and introduces the need for concurrency control. The representation of choice for the set is replication, or at least partial replication, because the set is much more frequently read than written. When new rules are being reduced, some or all of the existing rules from the set are needed, thereby generating frequent reads. Fortunately, most rules reduce to something trivial and are thrown away, so write accesses demands are low. Because we used a shared memory machine for the Knuth-Bendix, the underlying hardware handled replication, whereas in the distributed memory Gröbner basis code described below, replication is done explicitly within the set data structure.

2.2 Term Matching

Matching procedures take two logical terms as inputs, a *pattern* and a variable-free *target*, and it produces a substitution of terms for variables that makes the two inputs equal. If no such matching substitution exists, the procedure signals a failure. Term matching arises in logic programming, term rewriting, and other rule-based systems and represents a class of algorithms that traverse trees or graphs, although in practice most terms being matched are too small to justify large scale parallelization.

The sequential matching algorithm is recursive, so the program stack is the scheduling structure to be parallelized [Yel90]. As in Knuth-Bendix it is replaced by a task queue, in this case using order of insertion as the priority, i.e., each local queue is FIFO. The solution to a matching problem is a substitution — a mapping from variables to terms — that starts out as the identity mapping and grows to the final result or becomes an over-constrained failure value if one variable is assigned multiple values. The ratio of reads to writes is lower than for the Knuth-Bendix rewrite rule set, so replication is not a clear choice. On a distributed memory machine we might choose a partitioned or hash table or one with dynamic caching, but on shared the shared memory machine we use a centralized structure and leave caching decisions to the underlying hardware.

2.3 Eigenvalue

Our first distributed memory application is not symbolic, but is a classic example of a search problem, and therefore falls into the category of interest for our study. We also have implementations of other search problems, including the over-used n-queens example and traveling salesman problem, but the basic principles are the same. The scheduling structure in each case is some type of task queue that hold nodes from the search tree. The solution structure may be something a simple as a single value: in branch and bound algorithms, the bound represents a current approximation to the solution.

The *bisection algorithm* for computing the eigenvalues, an algorithm used in the ScaLAPACK library, is also a search problem [DDvdGW93]. A symmetric tridiagonal $N \times N$ real matrix is known to have N real eigenvalues and it is easy to find an initial range on the real line containing all eigenvalues. Then, given a real number x, it is possible to calculate how many of the N eigenvalues are less than x. This primitive can be used to successively subdivide the real line and locate all eigenvalues to arbitrary precision.

A parallel implementation of bisection can use a static subdivision of the initial range, but this has poor parallel efficiency if the eigenvalues are clustered, because the work load is not balanced [DDR94]. A solution is to use a task queue with load balancing for the scheduling structure. Because our machine target is now a distributed memory multiprocessor, locality is a more obvious concern, but for bisection, the tridiagonal matrix is relatively small and can be statically replicated, so the only data associated with each task is the endpoints of their interval. A simple randomized scheduler sends each task to a random processor upon insertion. This scheduling structure is called `RandQue` in Multipol; it has poor locality properties if there is an advantage to executing tasks on the processor that created them, but is adequate for the bisection algorithm [CRY94].

The intervals stored in the `RandQue` act as the approximate solution as well as the scheduling structure. As the intervals shrink, the approximation improves until a solution of the desired accuracy is obtained. We will observe this phenomenon of a single data structure playing both roles in one other application, the Tripuzzle.

2.4 Gröbner Basis

The Gröbner basis program is a completion procedure, like Knuth-Bendix, but it manipulates polynomials rather than rewrite rules and is guaranteed to terminate successfully. From a high level, the computation is very similar: for each pair of polynomial a new polynomial is computed; if the new polynomial is shown to be an linear combination of existing ones it is eliminated, otherwise it is added to the set.

The concurrency control in Gröbner is somewhat simpler than in Knuth-Bendix, but the key difference in our implementations is the use of distributed memory for Gröbner basis. A representation for the polynomial pairs is not difficult, since this data structure was already partitioned in the shared memory case. The `RandQue` provides good load balance assuming locality is not a concern. The basis of polynomials has the same performance requirements as the set of rewrite rules in Knuth-Bendix — reads are much more frequent than writes — but the underlying hardware no longer provides automatic replication. One software option is full replication, whereby each polynomial is broadcast when added to the basis. However, polynomials can be large and the processors are not synchronizationed, so this disrupts the computation and leads to poor processor utilization. Instead, we use software caching with a Multipol data structure called `ObjLayer`. Object caching avoids false sharing and fragmentation problems of hardware caches, but has higher address translation overhead. It also has an advantage of flexibility: we use a consistency protocol that is specific to the data structure, and make scheduling decisions based on cache state. For example, when new polynomials are added or old ones simplified, other processors may have stale or incomplete copies of the basis. Fortunately, this does not prevent them from doing useful work. When a processor finds a polynomial that appears to be new, i.e., did not reduce to zero, it locks the basis, obtains a consistent copy of all elements, performs one final check on reducibility, and finally adds the polynomial.

The locking solves the consistency problems and enforces the uniqueness of elements, but it can lead to performance bottlenecks as processors wait for locks. To avoid these overheads, we use multi-threading. If a processor cannot acquire the basis lock to perform the desired task, it suspends the current work and picks up something unrelated. Multi-threading support is built into the Multipol runtime layer and is done at user level to avoid the costs of saving complete thread contexts. As with caching, some hardware designers would place multi-threading support into the hardware. This may lower the cost of threading operations, but gives up some flexibility. In our approach, the library designer can decide whether a cache miss may be ignored (because the value is not essential) or should result in a change of context.

2.5 Phylogeny Problem

The problem of determining the evolutionary history for a set of species, known as the *phylogeny problem*, is fundamental to molecular biology. Evolutionary history is typically represented by a *phylogeny tree*, a tree of species with the root being the oldest common ancestor and the children of a node being the species that evolved directly from that node. Each species in a set is represented by a set of traits or character values. One technique for solving the phylogeny problem, called *character compatibility*, is to search through the power set of characteristics to see which ones are in a sense consistent. The notion of consistency in this problem is the existence of a particular kind of phylogeny tree called a *perfect phylogeny tree*. The specifics are not important. What is important is the structure of the search space (a power set) and the following property of the perfect phylogeny trees: if none exists for some set of characters S (the set is inconsistent), then none exists for any superset of S.

Although the search space has a known structure, the above property allows for unpredictable pruning, which leads to load imbalance. The `RandQue` data structure is not appropriate in this application, because there locality is important. However, work stealing, using the `TaskStealer` data structure in Multipol, provides load balance that is almost as good as `RandQue`, but with better locality. (Work stealing is provably optimal, in the same sense that randomized task sharing is [BL94].) The basic difference is that `TaskStealer` leaves tasks on the processor that created them until another processor becomes idle. The `TaskStealer` was originally used in the Eigenvalue and Gröbner basis problems, until the simpler `RandQue` proved to work as well [CRY94].

The choice of a data structure for holding a partial solution is more difficult in Phylogeny than in the other search problems, because we need a representation of the result (success or failure) of every node searched so far. Fortunately, because the structure of the search space, this information can be compressed using a trie, but the trie should ideally be shared between processors. We use a lazily replicated trie with global synchronization points for making the shared copies consistent.

2.6 Tripuzzle

The Tripuzzle problem is to compute the set of all solutions to a single player board game. The parallelism comes from considering a set of moves simultaneously, and the solution is the set of all moves that result in a winning game. The parallel algorithm is bulk-synchronous. At each step, all processors look at a set of resulting boards from the previous step and compute the set of legal moves. As in the Eigenvalue problem, a single data structure is use to load balance the computation and to store the current approximate solution. The data structure is a partitioned hash table: if the same board is found from two different series of moves, they will clash in the hash table and be collapsed; the hashing function distributes elements of the hash tables across processors and therefore also acts as a load balancer. The processors look at the local portion of their hash table when computing the next step.

3 Multipol Design

Multipol is a publicly available library of distributed data structures, designed for distributed memory multiprocessors. One of the first priorities in any software support for large scale multiprocessors has to be performance. In Multipol, the design of clean interfaces and portability are also high priorities, although some compromises are made to the interface to satisfy performance demands. 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, JY95].

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 Multipol 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).

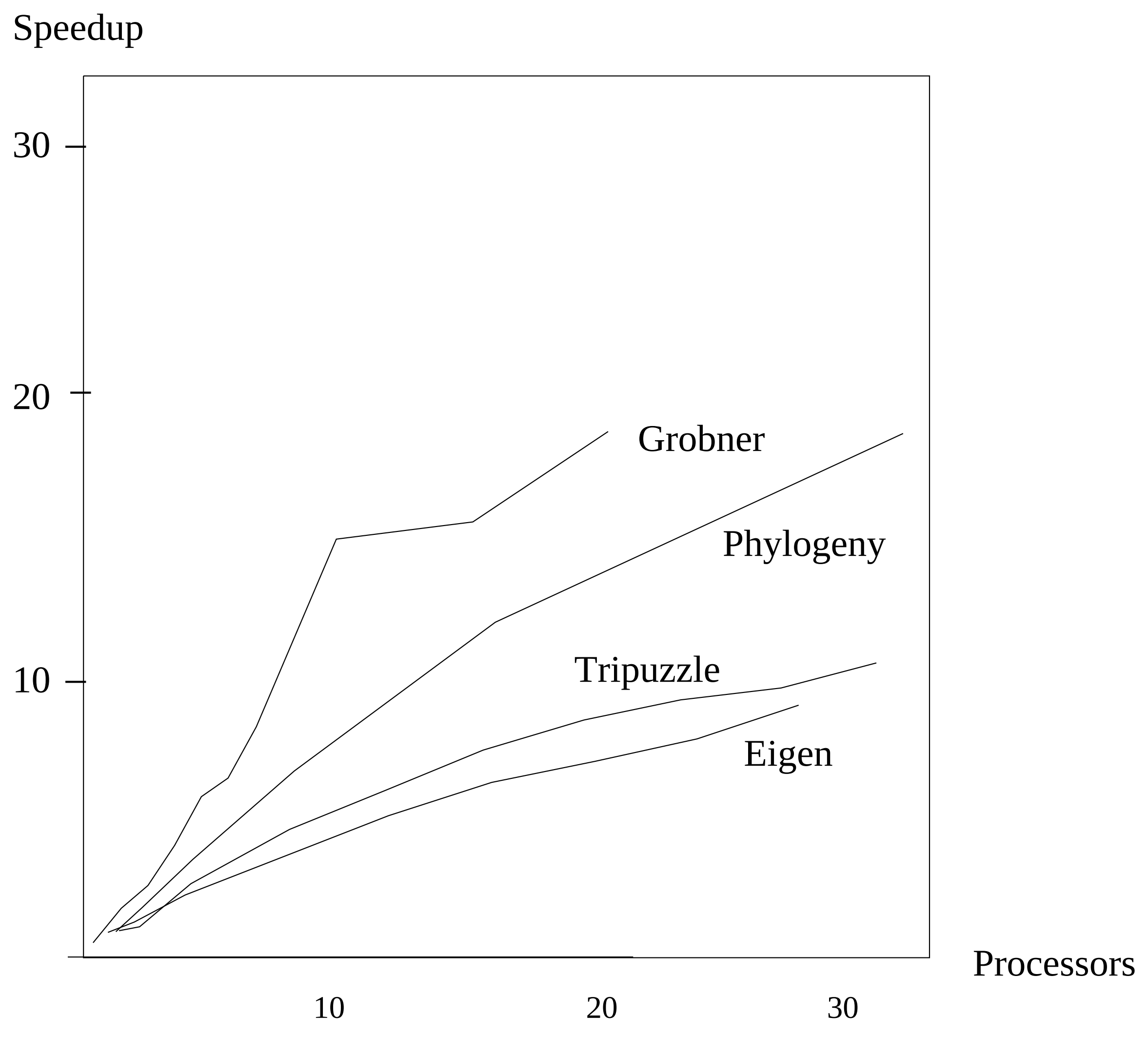
4.2 The Multipol communication primitives

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 a circuit simulation application and the Tripuzzle example show that message aggregation can reduce the running time by as much as 50% on machines such as the IBM SP1.

5 Performance

The speedups of the distributed memory applications are shown in the following figure. These numbers were all taken on a 32 processor Thinking Machines CM5 multiprocessor. The performance of these applications, like many symbolic problems, is highly dependent on the input, and there is no obvious notion of problem scaling.



6 Conclusions

We have described several parallel symbolic applications and shown that common programming techniques, software caching, replication and dynamic load balancing can be used across applications, and in some cases the data structures themselves can be re-used. We identified two types of data structures that are common in symbolic applications, one used for load balancing and another used for sharing partial solutions.

The Multipol library fills the gap in the parallel software tools for programming irregular applications on distributed memory machines. We have identified some of the primary performance issues in the Multipol design, namely, locality, load-balance, latency hiding, and communication elimination, and gave an overview of our solution based on a multi-threaded runtime layer. The split-phase interfaces in Multipol are a concession to performance demands, and 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.

The irregular applications described here represent some of the more challenging problems for parallelism. We believe that the library approach is a good compromise between hand-coded, machine-specific applications, and approaches based entirely on high level languages and compilers.

References

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Thirty-Fifth Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, November 1994.

[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.

[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. (Also available as Technical Report UCB/CSD 91/594, CS Div., University of California at Berkeley).

[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.

[DDR94] J. Demmel, I. Dhillon, and H. Ren. On the correctness of parallel bisection in floating point. Tech Report UCB//CSD-94-805, UC Berkeley Computer Science Division, March 1994. available via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/csd/csd-94-805, file all.ps.

[DDvdGW93] J. Demmel, J. Dongarra, R. van de Geijn, and D. Walker. LAPACK for distributed memory machines: the next generation. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1993.

[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.*

[JY95] J. Jones and K. Yelick. Parallelizing the phylogeny problem. In *Supercomputing '95*, December 1995. To appear.

[KB70] Donald E. Knuth and Peter B. Bendix. *Simple Word Problems in Universal Algebras*, pages 263–297. Pergamon, Oxford, 1970.

[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 multi-process programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[RV89] Eric Roberts and Mark Vandevoorde. Work crews: An abstraction for controlling parallelism. Technical Report 42, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1989.

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

[Yel90] Katherine A. Yelick. *Using Abstraction in Explicitly Parallel Programs.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA 02139, December 1990. Also appeared as MIT/LCS/TR-507, July 1991.

[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.