Portable Parallel Irregular Applications

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

Abstract

Software developers for distributed memory multiprocessors often complain about the lack of libraries and tools for developing and performance tuning their applications. While some tools exist for regular array-based computations, support for applications with pointer-based data structures, asynchronous communication patterns, or unpredictable computational costs is seriously lacking. In this paper we describe our experience with six irregular applications from CAD, Robotics, Genetics, Physics, and Computer Science, and offer them as application challenges for other systems that support irregular applications. The applications vary in the amount and kind of irregularity. We characterize their irregularity profiles and the implementation problems that arise from those profiles. In addition to performance, one of our goals is to provide implementations that run efficiently with minimal performance tuning across machine platforms, and our designs are influenced by this desire for performance portability. Each of our applications is organized around one or two distributed data structures, which are part of the Multipol data structure library. We describe these data structures, give an overview of some key features in our underlying runtime support, and present performance results for the applications on three platforms.

1 Introduction

This paper reports on several case studies of irregular parallel applications and the data structures within them. It also describes systems support in the form of a parallel data structure library, Multipol, 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 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

*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.

space. The design problem for these shared state structures is to maximize throughput of the operations to read and modify the structures. In some applications, a single data structure serves as both a scheduling structure and for storing computed values of the solution. Many scientific and engineering applications are simulations, rather than symbolic programs, but these may contain irregularities as well. For simulations, the underlying data structure is usually a grid over the physical domain, which may be a regular multi-dimensional mesh or an unstructured graph.

Our Multipol applications include an eigenvalue computation based on a divide-and-conquer algorithm [CRY94], an electromagnetics simulation kernel [CDG+93, Wen95], a symbolic algebra system [CY93a, CY93b, CY94], a timing level circuit simulator [Wen92, WY93, WY95], a solution to the phylogeny problem from computational genetics [Jon94, JY95], and a game tree search [WCD+95, Wen95]. 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, CCK92], there are no comprehensive data structure libraries, such as those that exist for uniprocessors. Multipol is such a library.

Multipol is designed to help programmers write irregular applications such as discrete event 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 balancing 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.

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, and Intel Paragon, and 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 remote memory than local 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 atomic threads, which we call *fibers*, support for efficient communication across architectural platforms, synchronization primitives including global snapshot facilities, and controlled scheduling. Our library-based approach offers several 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 extensible 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 applications and their key data structures. Section 3 introduces the common issues in designing high-performance, portable applications, and describes some of the solutions used in the Multipol library. Section 4 sketches the portable runtime system on which Multipol is built, Section 5 presents some performance data, and Section 6 draws some conclusions from our experience.

2 The Applications

In this section we give a brief description of our applications and the high level data structures that are replaced during parallelization. The purpose is to identify the main data structures and design trade-offs that shaped the Multipol design.

In each of the symbolic applications, 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. A common property of several of the solution structures is having some type of monotonicity property that allows out-of-date or incomplete versions to be used. Two of the applications are simulations, and in each of these cases locality as well as load balance are crucial to performance. For precise problem statements, alternate approaches, related applications work, and complete descriptions of our implementations, we refer the reader to more extensive papers on each application.

2.1 EM3D

The EM3D application computes the flow of electro-magnetic waves through a three-dimensional object [CDG+93]. Each object is modeled by an unstructured three-dimensional grid of convex polyhedral cells, which is called the "primary grid." A dual grid is defined with respect to the primary grid, having grid points at the centers of the primary grid's cells. The electric field is evaluated for the faces of the primary grid, while the magnetic field is evaluated for the faces of the dual grid. These field values form nodes in a bipartite graph, because the computation of a face uses data only from the faces of a different grid. The structure of the bipartite graph remains static throughout the computation.

The sequential algorithm consists of a series of alternating steps for computing the electric field and the magnetic field on the bipartite graph. The change in the electric field of a node is a linear combination of the magnetic field of its neighboring nodes, and is calculated during the odd numbered iterations. Similarly, the change in the magnetic field of a node is a linear combination of the electric field of its neighboring nodes, and is calculated in the even numbered iterations. Since the electric and magnetic field values are calculated in different steps, dependencies exist only between steps and never within a step.

Although the unstructured grid makes this an irregular application by some definitions, it is one of the most regular of the Multipol applications, because the graph is fixed at program startup time and the amount of computation per node is approximately the same. This allows for the kind of *inspector/executor* processing that is done in the Chaos system [DUSH94], although the EM3D kernel is based on a synthetic graph, so the inspector phase is unnecessary in our code. The computational kernel of EM3D reflects the structure of many synchronous simulations; it is a sequence of bulk-synchronous phases that alternately computes the states of the two

parts of the bipartite graph. The key issues in obtaining high performance are load balancing the graph nodes across processors, minimizing the number of cut edges between processors, and packing many small messages into fewer large ones. The graph partitioning problems are difficult, but all of these optimizations can be done in a preprocessing phase at runtime.

2.2 CSWEC

The CSWEC application is a classic example of discrete event simulation [WY95]. The program simulates the voltage output of combinational digital circuits, that is, circuits without feedback signal paths. The program partitions the circuit into loosely coupled subcircuits that can be simulated independently within a time step. The time step size is determined independently for each subcircuit based on its current state. At the end of a time step, if the subcircuit's state cannot be extrapolated linearly from its previous state within some error margin, the new state is propagated to its fanout subcircuits. The propagation of subcircuit state is called an *event*. The event-driven approach significantly reduces the computation and communication required for the simulation.

We adopt a *conservative* approach to parallel asynchronous simulation [CM81], although in related work we describe a speculative version of this simulator which is more complex, but useful for circuits with feedback loops [Wen92, WY93]. The program uses a distributed *event graph* data structure to represent the circuit. Each graph node corresponds to a input signal port or a output voltage point of a subcircuit. The directed edges in the event graph are essentially communication channels for multicasting event messages from the source nodes to their sink nodes. The data structure guarantees the event messages are delivered in the order sent, and the maximum number of outstanding event messages do not exceed a specified threshold (called the *capacity*). A thread is created for each subcircuit on the processor that owns the part of the event graph representing the subcircuit. The threads communicate and synchronize via the event graph data structure, which encapsulates the connectivity structure of the circuit. The programmer can tune the capacity to trade off the parallelism of the simulation and the amount of memory used.

The semantics of the graph multicast operation is weakened to reduce synchronization overhead. When the multicast call returns, the data structure guarantees that the event message will eventually be delivered, but the completion of the call does not necessarily imply that the event message has been received. The weak semantics eliminates the hand-shaking between the sending and the receiving nodes for acknowledging event messages. The application can call a data structure primitive to force the delivery of all event messages, for example, when taking a snapshot of the graph to perform deadlock detection.

The event graph also takes advantage of the graph structure to optimize communication performance. For example, identical event messages that are sent to different nodes on the same processor are collapsed into one physical message to reduce communication. The control messages for managing message buffers are also collapsed in a similar manner. The collapsing of messages significantly reduces the amount of communication if the nodes have a large number of fanouts, which is common in large circuits.

2.3 Eigenvalue

The Eigenvalue program is a classic example of a search problem. We also have implementations of other search problems, including the 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 holds nodes from the search tree. The solution structure may be something as simple as a single value: in branch and bound algorithms, the bound represents a current approximation to the solution.

The *bisection algorithm*, an algorithm used in the ScaLAPACK library [CDPW92], is a search-based method for computing the eigenvalues of symmetric tridiagonal matrices. 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 a distributed memory multiprocessor, locality is an 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 pair of endpoints of each interval. A simple randomized scheduler pushes each task to a random processor upon insertion. This scheduling structure has, in a sense, provably optimal performance, but poor locality properties if there is an advantage to executing tasks on the processor that created them. Given the replicated matrix, locality is not a concern in the bisection algorithm [CRY94].

The intervals stored in the task queue 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 used for solving systems of nonlinear algebraic equations. The problem is: given one set of polynomials, compute another set that has the same roots but is, in a technical sense, simpler. It has two large data structures: the current set of polynomials and a queue in which all pairs of all polynomials are placed. From a high level, the computation is very similar. For each pair of polynomials, a new polynomial is computed; if the new polynomial is shown to be an linear combination of existing ones — a computation that is performed by simplification relative to the set of existing polynomials — the new one is eliminated. If is does not simplify to zero, it is added to the set, and all pairs that can be created with it are added to the pair queue.

The creation and simplification of new polynomials constitutes the bulk of the work, so the data structure that holds pairs of polynomials is the primary scheduling structure in the parallel implementation. The ordering of pairs within this scheduling queue is quite flexible, although orderings can have a significant effect on the total amount of work performed. The pair queue is implemented as a task queue,

in which each processor's portion ordered by heuristics, but no global ordering is maintained. The set of polynomials is read much more frequently than it is written, since reduction of polynomials requires reading the set and most of the polynomials will reduce to zero. 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 synchronized, so this disrupts the computation and leads to poor processor utilization. Instead, we use software caching. 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 is done at user level to avoid the cost 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. An important optimization in the sequential program is to keep track of all inconsistent sets, and before computing a perfect phylogeny tree for a new set, we check whether the set or any of its subsets have been found inconsistent.

Although the search space has a known structure, the above property allows for unpredictable pruning, which leads to load imbalance. Aggressive task pushing is not appropriate in this application, because the child tasks of a given task are closely related to their parent, and often need the failure information computed by

the parent to avoid rendundant computation. Work stealing, a variation on the task pushing data structure, provides load balance that is almost as good as pushing, but with better locality. (Work stealing is provably optimal, in the same sense that randomized task pushing is [BL94].) The basic difference is that stealing leaves tasks on the processor that created them until another processor becomes idle.

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. Each node in the search space is a subset of all of the characters, represented as a bit string, and a binary trie is built using the bit strings as keys. Using the trie, we can easily look for subsets as well as identical sets, and we compress the trie as it logically grows by storing only minimal subsets. Ideally, the trie should be shared between processors, but an out-of-date trie will result in extra work without changing the program correctness. We therefore avoid communication overhead and synchronization using 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 used 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.

2.7 Summary of Application Workload

Figure 1 compares the application workload in the following three aspects: computation, communication, and synchronization.

The computation in an application can be statically balanced if their granularities can be predicted (such as in the EM3D program) or reasonably estimated using application-specific heuristics (such as in the Tripuzzle program and the CSWEC program). Otherwise, a dynamic load balancer should be used. Choosing the load balancing strategy involves a tradeoff between load balance and locality. For example, locality has little impact on the Eigenvalue program and the Grobner basis program, so randomized task pushing can be used to improve load balance. However, the Phylogeny program performs better with task stealing, because it preserves locality and thereby reduces the amount of communication and redundant work.

Communication performance is affected by the the total volume and the nature of the communication events. Communication overhead is less pronounced if the volume of communication per event is large, as in the EM3D program. The perfor-

Application	Workload		
	Computation	Communication	Synchronization
EM3D	Statically balanced using input structure	Some large predictable events	Predictable global events
CSWEC	Statically balanced using heuristics	Many small unpredictable events	Independent events
Eigenvalue	Dynamically balanced, locality not important	Some small unpredictable events	Independent events
Gröbner Basis	Dynamically balanced, locality not important	Some small & large unpredictable events	Independent events
Phylogeny	Dynamically balanced, locality is important	Some small & large unpredictable events	Independent & global events
Tripuzzle	Statically balanced by randomization	Mostly small unpredictable events	Predictable global events

Figure 1: Comparison of application work loads.

mance of such applications are limited by the network bandwidth. In comparison, small, unpredictable communication events incur more overhead, because they require additional hand-shaking between the sender and the receiver to set up the communication. For applications with many small unpredictable communication events, such as the Tripuzzle program, the performance is likely to be limited by the communication start-up overhead.

Synchronization events can take place globally for a collection of accesses, such as the global barriers in the EM3D, Phylogeny, and Tripuzzle programs, or independently for the individual accesses, such as retrieving a task in the Eigenvalue program or acquiring the basis lock in the Gröbner Basis program. Global synchronization events has less performance impact if their overhead can be amortized over a large number of accesses. Bulk synchronous applications such as the EM3D program and the Tripuzzle program are more likely to take advantage of global synchronization events. Independent synchronization events may become the main source of processor idle time, unless their latencies can be overlapped with useful computation.

3 Data Structure Design Issues

The applications demonstrate varying degrees of irregularity that challenge many existing systems. We used a library-based approach by providing a set of distributed data structures that are designed for high performance, clean interfaces, and portability. Because the architecture targets are large scale multiprocessors, performance is the priority, and some compromises are made to the interfaces to satisfy performance demands. In this section, we discuss some of the programming techniques that are used to obtain high performance across architectures.

When parallel programs show suboptimal performance, the problem can generally be identified as 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 using a combination of the techniques outlined below.

3.1 Latency Masking

The latency of remote operations may 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.

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 on workstation networks with longer hardware latencies and expensive remote message handlers, the savings may 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 finite-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. An 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], n-body solvers, and the tripuzzle problem, data structure updates are delayed until the end of a computation phase, at which point all processors wait for all updates to complete. For these applications, the latency can be amortized by packing small messages into larger ones. Moreover, because all processors reach a communication phase at roughly the same time, very efficient communication techniques including synchronous communication or asynchronous messaging with polling can be used.

In our experience, a more significant source of latency is the time one processor spends waiting for synchronization events from remote processors. These events are often the completion of remote function invocations, which may be delayed by scheduling constraints, insufficient polling, or high interrupt overhead. Of the six applications, latency problems are worst in the application with many small messages, unpredictable synchronization events, and no separate synchronization phase, namely the Gröbner basis application and CSWEC. In both programs, having multiple remote operations outstanding is not sufficient, so user level multi-threading is used to overlap the delays. In CSWEC, multi-threading is fundamental to the asynchronous simulation semantics, so it is difficult to quantify a pure performance benefit, but in Gröbner basis, the overlap of high level operations saves about 10%. The Eigenvalue problem has many of the same characteristics, but there are few synchronization events between tasks; it uses a non-blocking operation to insert tasks

into a distributed queue, but does not require the more complex multi-threading.

3.2 Locality

Locality is crucial when the state used by a given task is large and cannot be replicated for reasons of space or expense of keeping the copies consistent. One way to improve locality is to reduce the volume of communication by carefully placing tasks on the processors that own most of the relevant data. Static techniques for locality optimizations include partitioning, which attempts to divide up the data set into loosely dependent partitions, and replication, which keeps a copy of infrequently written data on each processor. The EM3D application is typical of static unstructured grid problems in that the grid is irregular, but once the program starts the grid remains fixed, so the application is optimized for load balance and locality using a static partition of the grid. In some problems where the grid changes slowly over time, the same technique can be used with some repartitioning steps after a set of timesteps. Dynamic locality techniques include mobile objects in which a single copy moves around the system, and caching, which maintains multiple copies, depending on the runtime usage. For replication and caching, relaxed consistency may be used to further reduce communication by allowing multiple copies to be inconsistent.

Many applications can take advantage of these relaxed data structures because there is no strict ordering on updates and because an old value of the data structure can be used profitably. In two of the applications, the solution data structure has a kind of monotonicity that can be exploited. 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 the set. This yields a correct, albeit different, execution than the sequential program [CY93b, JY95]. These out-of-date copies can lead to parallelism overhead in the form of useless computation in both applications, so there is a crucial trade-off between the frequency of communication to update the shared state and the amount of redundancy on the computation.

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 cost, the former is very important. Many small messages are aggregated into one large physical message to amortize the overhead. Several other systems, including Chaos [DUSH94] and LPARX [Bad91], also use message aggregation. Even for machines such as the 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 compiler or dynamically by the runtime system. Chaos and LPARX uses a static or semi-static approach, in which small messages are packed by the library and shipped at a user specified time. We use a more dynamic approach, in which messages accumulate in a buffer and are sent when the buffer fills or when a certain amount of time has passed since the last packet was sent. Our approach is more complicated to implement and

involves subtle issues of liveness and flow control, but is essential in applications like CSWEC and Gröbner basis, in which there is no advanced knowledge of communication events and no global synchronization points at which all communication may be performed.

There is a down-side to message aggregation in that holding messages until a sufficiently large amount of data has collected may increase the observed latency to the higher level software. This must be weighed against the reduction in communication overhead, but our decision to use aggregation is motivated by a desire to run on machines with high as well as low communication overhead. The down-side of aggregation is mitigated by our use of multi-threading since the increased delays may be hidden by other work. This set of design decisions relies on the availability of excess parallelism in the application, which is certainly available in our highly asynchronous applications.

A 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. A Split-C version of the EM3D problem based on fine-grained communication also showed a noticeable improvement [CDG+93].

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 take an intermediate position, in 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 structure. 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 the local data. In the tripuzzle program, this means iterating over the elements of the local hash table to compute the next set of board values, while insertions are performed across processors. In Gröbner basis and Phylogeny, search operations are performed on the local view of a list or tree, but modifications are logically applied to the global view. The data structures are therefore said to be *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, while load balance of tasks requires that work be spread evenly across processors. In applications with high locality, the two kinds of load balancing are intimately linked. There is typically a trade-off between locality and load balance, since increased load balancing moves more tasks away from their creating processor, which is likely to be the processor holding their data.

For applications with predictable computation costs and fixed data structures, like EM3D, static load balancing techniques apply. In problems with highly unpredictable computational costs, dynamic decisions are often the right choice. The Gröbner basis application, phylogeny problem, and eigenvalue problem all use a kind of task queue to balance work. The loss of locality is acceptable, because the shared state is replicated or cached for other reasons. However, we find a significant difference between aggressively pushing tasks to remote processors and stealing them only when idle: the former gives better load balance and performs better in the eigenvalue problem in which the shared state is replicated read-only; the latter gives better performance in the phylogeny problem, because the local copy of the solution set is much more likely to avoid redundant work than another processor's copy. The tripuzzle uses an unusual form of dynamic load balancing that is similar to task pushing: the hash table holds both the current approximation to the solution and the set of tasks for the next step, and the load balance comes from the hash function's distribution across processors.

The CSWEC application is unusual, because it uses static load balancing in spite of high task time variability. Subcircuits are randomly assigned to processors at program startup time, using some heuristics to balance the workload. This works reasonably well because even at runtime there is very little information that could be used in advance to schedule tasks. More importantly, the state associated with each subcircuit is large and should not be duplicated, so the avoidance of high communication costs from moving subcircuits outweighs the advantage of improved load balancing.

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 to enhance portability of the data structures. 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 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

We implemented all five applications except Gröbner Basis on top of the Multipol runtime layer and data structure library. The programs are analyzed and optimized using *Mprof*, a performance profiling toolkit for programs that use the Multipol library [Wen95]. The Multipol programs are portable across several distributed

memory platforms, including the Thinking Machines CM5, the Intel Paragon, and the IBM SP1/SP2. Figure 2 gives the speedups of the SWEC, Eigenvalue, Phylogeny, and Tripuzzle programs on these three machines. The Gröbner basis implementation was originally developed before Multipol, and although it was converted to use the library and runtime support, it contains some machine-specific functions that limit its execution to the CM5.

The performance of these applications, like many irregular problems, is highly dependent on the input and the machine architecture, and there is no obvious notion of problem scaling. The generally good results from the Multipol implementations show that our approach achieves performance portability on a variety of architectures.

6 Conclusions

We have described several irregular parallel 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.

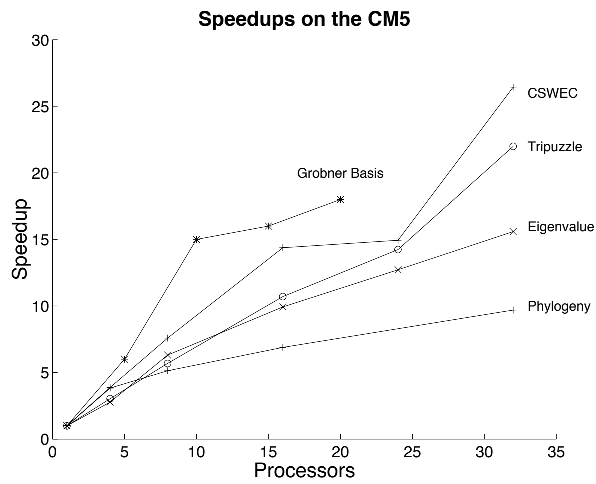
References

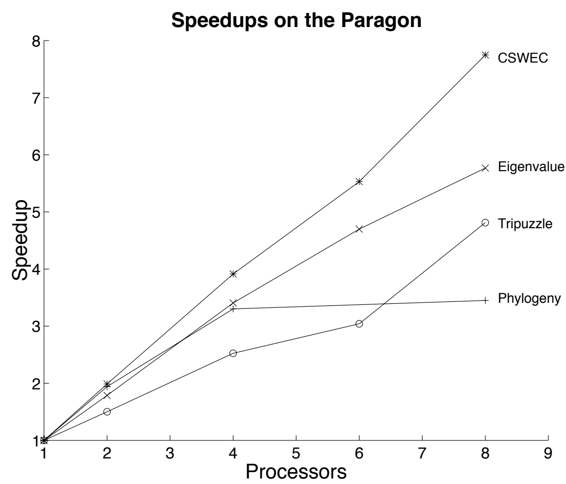
[And93] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.

[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.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Thirty-Fifth Annual Symposium on*





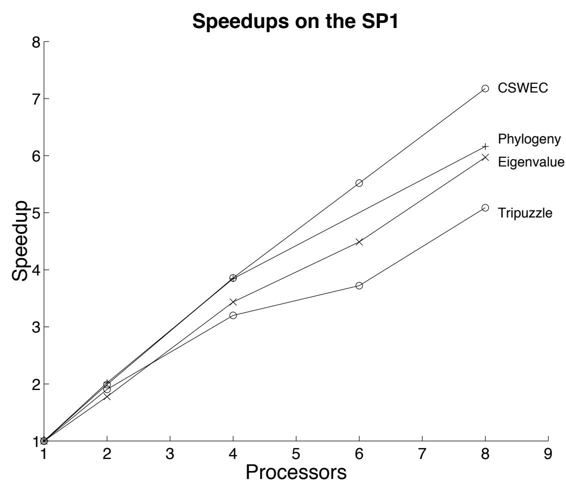


Figure 2: Speedup curves of the applications.

Foundations of Computer Science (FOCS '94), pages 356–368, November 1994.

[CCK92] Peter Carlin, Mani Chandy, and Carl Kesselman. The Compositional C++ Language Definition. Technical Report CS-TR-92-02, California Institute of Technology, 1992.

[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.

[CDPW92] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[CM81] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11), April 1981.

[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).

[CY93a] Soumen Chakrabarti and Katherine Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Proceedings of the 1993 Conference on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[CY93b] 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.

[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.

[DUSH94] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, September 1994.

[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.

[Jon94] Jeff Jones. Exploiting Parallelism in the Perfect Phylogeny Computation. Master's thesis, University of California, Berkeley, Computer Science Division, December 1994.

[JY95] J. Jones and K. Yelick. Parallelizing the Phylogeny Problem. In *Supercomputing '95*, December 1995.

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

[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, Computer Science Division, December 1994.

[WCD+95] Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Katherine Yelick. Runtime Support for Portable Distributed Data Structures. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*, May 1995. Boleslaw K. Szymanski and Balaram Sinharoy (Editors), Kluwer Academic Publishers, Boston, MA, pp. 111–120.

[Wen92] Chih-Po Wen. Parallel Timing Simulation on a Distributed Memory Multiprocessor. Master's thesis, University of California, Berkeley, CA, 1992.

[Wen95] Chih-Po Wen. *Portable Library Support for Irregular Applications.* PhD thesis, University of California, Berkeley, CA, 1995.

[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.

[WY95] Chih-Po Wen and Katherine Yelick. Portable Runtime Support for Asynchronous Simulation. In *International Conference on Parallel Processing*, Oconomowoc, Wisconsin, August 1995.