Portable Library Support for Irregular Applications

by

Chih-Po Wen

B.A. (National Chiao-Tung University, Taiwan) 1988
M.S. (University of California at Berkeley) 1992

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Katherine A. Yelick, Chair
Professor Susan L. Graham
Professor Stuart Dreyfus

1995

Portable Library Support for Irregular Applications

Abstract

Portable Library Support for Irregular Applications

by

Chih-Po Wen

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Katherine A. Yelick, Chair

Building portable parallel programs on distributed memory multiprocessors and workstation networks is a complex task that is greatly facilitated by powerful infrastructure. In this dissertation, we develop important components of that infrastructure, focusing on irregular applications such as unstructured mesh computations, search problems, and discrete event simulation. We use a library-based approach to building such applications. The library provides a uniform programming interface on multiple platforms and has highly tuned implementations developed by the library programmer. Therefore, applications built on the library can be portable both in functionality and in performance.

We describe the major components of our parallel data structure library called *Multipol*, including two of the more irregular data structures and one application. The two data structures are a *task stealer* for dynamic load balancing and an *event graph* for discrete event simulation. The application is a timing-level circuit simulator for combinational circuits. We analyze the workloads of several applications built by the Multipol group and quantitatively characterize their irregularities.

The Multipol library is built on a runtime layer consisting of threads as well as communication mechanisms. The thread layer supports a basic computational abstraction called *fibers*, which are code sequences that appear to execute atomically. The fiber abstraction enables a portable multithreading execution environment for latency hiding. The thread layer also allows the programmer to supply *customized schedulers* to enforce application-specific scheduling policies. The communication layer provides portable primitives for expressing irregular communication. It uses a technique called *message aggregation*

to trade the excess parallelism in the application for better communication bandwidth.

We provide a new performance profiling toolkit called *Mprof* to help tune the performance of irregular parallel programs. Mprof identifies two major sources of performance inefficiency: overhead and insufficient parallelism. It uses statistical modeling to extract reusable cost models from benchmark executions. The cost models are combined with high-level statistics collected from an actual execution to provide low-overhead profiling information. Mprof also provides a *performance interface* for the library programmer to customize the profiling information and thereby preserve the library abstraction. Using information from Mprof, we optimize the performance of several irregular applications and demonstrate the performance portability of the Multipol library and runtime layer.

Professor Katherine A. Yelick
Dissertation Committee Chair

To my parents,

Chih-Huang Wen and Shiao-Lien Tsai,

and my brother,

Chung-Yao Wen

Contents

List of Figures

Acknowledgements

I would like to thank my research advisor, Professor Katherine Yelick, for her constant guidance and support since I joined her group. Kathy is the nicest professor I have ever met. She helped improve my presentation skills tremendously. She also played an important role in shaping the direction of my research.

I gratefully acknowledge contributions from other members of the Multipol group. They are Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones, and Arvind Krishnamurthy. Etienne and Soumen participated in the initial design of the Multipol runtime layer. I thank Professor James Demmel, Professor Stuart Deryfus, and Professor Susan Graham for serving on my qualifying exam and for reading this dissertation. I also thank Professor C. V. Ramamoorthy for his support during my first year at Berkeley.

My life at Berkeley would have been dull without the friendship of other EECS graduate students. They are Claudia Chandra, Wan-Teh "Wonton" Chang, Szu-Tzong Cheng, Sean Huang, Han-Gyo Kim, Joe King, Luis Miguel, Yung-Chul Shim, Huey-Yih Wang, and Robert Wang. Hanging out with them is a lot of fun. Special thanks go to Yochai Konig for being my pal and my partner in numerous projects.

Finally, I would like to dedicate this work to my parents, Chih-Huang Wen and Shiao-Lien Tsai, and my brother, Chung-Yao Wen. Without their encouragement and sacrifice this dissertation would not have been possible.

This work was supported in part by the Advanced Research Projects Agency (DOD) under contract no. DABT63-92-C-0026, by the Department of Energy grant no. DE-FG03-94ER25206, by the National Science Foundation grant no. CCR-9210260 and NSF Infrastructure grant nos. 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.

Chapter 1

Introduction

Recent research and commercial development efforts have produced a multitude of distributed memory parallel machines in the form of dedicated multiprocessors and networks of workstations. However, there has been a lack of consensus on their programming interface, and the parallel programming primitives often vary with the machine platform. Due to the rapid advances in hardware technology, the users of non-portable applications are often faced with the dilemma of staying with a stale platform or re-developing the applications for more powerful platforms.

One approach to parallel programming that addresses the portability problem is to use so-called "heroic" compilers such as parallelizing Fortran compilers [HKT91, ABC+88, PGH+90] or optimizing compilers for data parallel languages [CMF92, Hig92]. The programmer relies on the compiler to discover parallelism and to schedule communication and computation. This approach has had some success on problems such as dense matrix algorithms and regular grid computations, but there are many irregular applications that do not lend themselves to automatic parallelization or to array-based data parallelism. Examples of these irregular applications include unstructured mesh computations, divide and conquer algorithms, and discrete event simulation.

Another approach to parallel programming involves explicit message passing with message passing layers such as PVM [BDG+91] and MPI [For94]. Such message passing layers are often used when performance is the most important concern, because they allow the programmer to have complete control over parallelism and communication. However, the programming is fairly low-level, and the primitives are designed for cooperative message passing which makes irregular communication patterns awkward. In addition, although message passing provides functional portability, they do not provide performance portability, because the application programs contain implementation details that should vary with the machine for performance.

Parallel libraries offer a promising alternative by providing some of the programmability of high-level languages with the performance of message passing. A library may consist of portable communication primitives and computational abstractions, standalone algorithms, or encapsulated data structures. Applications built on the libraries can be portable without sacrificing performance, because the library has a uniform programming interface on multiple platforms, and it has highly tuned implementations developed by the library programmer. The library is also extensible and can provide domain-specific abstractions. This approach has been used successfully in several areas including dense matrix algorithms [Dem89, ABB+92, CDPW92], unstructured mesh computation [DHU+93, DUSH94, MSH+95], and adaptive mesh computation [KB95]. While their applications include some degree of irregularity, these libraries do not handle task parallelism or asynchronous communication.

Multipol [YCD+95] is a parallel data structure library designed for irregular applications with pointer-based data structures, asynchronous communication, and unpredictable task dependences or computation times. Multipol applications include unstructured mesh computations, search problems, discrete event simulation, and symbolic algebra problems. In this dissertation, we address several components in the design and implementation of Multipol. First, we give an overview of Multipol and its applications, including two data structures developed by the author, a *task stealer* for dynamic load balancing and an *event graph* for discrete event simulation. Second, we present the design and implementation of a portable runtime layer on which the library is built. The runtime layer runs on several distributed memory multiprocessors and networks of workstations, and it provides functional portability as well as performance portability across platforms. Third, we present *Mprof*, a performance profiling toolkit for Multipol which detects both overhead and insufficient parallelism in Multipol applications. Finally, we evaluate the performance of Multipol and identify several performance tuning techniques for Multipol applications, using the profiling information from Mprof.

1.1 Overview of Results

This dissertation addresses the following three questions:

- What programming interface should be used for building irregular applications?
- How should irregular computation and communication patterns be supported on distributed memory platforms?
- What profiling information is useful for detecting performance inefficiencies in irregular parallel programs and how can such information be collected efficiently?

We begin in Chapter 2 by examining the programming abstractions and execution behaviors of irregular applications. Several non-trivial applications built by members of the Multipol group are used to illustrate irregularities in parallel programs and the primary data structures within them. The applications include examples from bulk-synchronous simulation, divide and conquer algorithms, search problems, and discrete event simulation; their workload gives a quantitative characterization of the irregularities in parallel programs. The data structures perform functions such as data distribution, load balancing, scheduling, communication and synchronization; they suggest the common programming abstractions used in irregular applications.

We then present the Multipol runtime layer, which provides infrastructure for building distributed data structures and irregular applications. The runtime layer consists of threads as well as communication mechanisms. The thread layer supports a basic computational abstraction called *fibers*, which are code sequences that appear to execute to completion without preemption or suspension. The fiber abstraction enables a portable multithreading execution environment for latency hiding. The thread layer also provides mechanisms for enforcing application-specific scheduling policies using *customized schedulers*. The communication layer provides a machine-independent interface for expressing irregular communication. It also optimizes irregular communication workloads for efficient execution on distributed memory architectures, using a technique called *message aggregation*, in which several small messages are packed into one larger message. In asynchronous applications, message aggregation trades excess parallelism for communication bandwidth, a tradeoff that we show is essential for performance on platforms with high communication start-up overhead. The design and the implementation of the Multipol runtime layer is presented in Chapter 3.

Multithreaded programs with unpredictable communication and synchronization patterns are difficult to performance tune. We provide a performance profiling toolkit called *Mprof* to help the programmer tune the parallel program. We identify two common sources of performance inefficiency: overhead and insufficient parallelism. Mprof reports on the execution costs of data structures and the runtime layer to expose overhead, and it measures the latency of synchronization events to expose insufficient parallelism. Mprof uses statistical modeling to extract reusable cost models from benchmark executions. The cost models are combined with high-level statistics collected from an actual execution to provide low-overhead profiling information. The use of high-level statistics is particularly important for profiling multithreaded programs, where direct measurement methods are difficult to implement and incur high runtime overhead. To profile programs built from Multipol and other data structure libraries, Mprof provides a performance interface which the library programmer can use to customize the profiling information for a particular data structure. The design and implementation of Mprof is described in Chapter 4 using a simple running example.

In Chapter 5, we evaluate the effectiveness of our performance tools and the Multipol library using several applications. We show how profiling information provided by Mprof can be used to identify performance inefficiencies and optimize the applications from Chapter 2.

Finally, in Chapter 6, we summarize the results in this dissertation and highlight our contributions. We also draw conclusion about the generality of our work and discuss future research directions.

Chapter 2

Building Irregular Applications with Distributed Data Structures

There is no silver bullet to the parallel programming problem. Different types of applications require different programming primitives and optimization techniques. Instead of providing a fixed set of abstractions and optimizations in the form of a language, we adopt a library-based approach to building irregular applications, in which an extensible library of data structures is provided along with some basic communication and synchronization primitives. The use of a library allows for portability, because the high-level programming interface provided by the library shields the application programmers from the implementation details. The library approach has been adopted by PARTI [BSS91], CHAOS [DHU+93, DUSH94, MSH+95] and LPARX [KB95], which are specialized libraries for unstructured and adaptive mesh applications. They handle certain irregular data structures, but are limited to bulk synchronous computation patterns.

In this chapter, we present an overview of the Multipol data structure library and some of the applications that use the library. The data structures and applications were developed by members of the Multipol group, including the author. The purpose of the chapter is two-fold. First, we give a quantitative characterization of irregular applications to justify the design of the runtime layer, which is discussed in detail in Chapter 3. The example applications are also used to evaluate the performance of our runtime library in Chapter 5. Second, we describe two of our own data structures that are interesting in their own right: the task stealer and the event graph. For each of the data structures, we describe its programming interface and the implementation techniques used to address the irregularities arising in its use.

The rest of the chapter is organized as follows. Section 2.1 describes the characteristics of irregular parallel programs. Section 2.2 presents four Multipol data structures and their applications. Two of the data structures and four of the applications are written by other members of the Multipol group. Section 2.3 summarizes the application characteristics and runtime requirements, and Section 2.4 describes related work.

2.1 Irregularities in Parallel Programs

We call an application "irregular" if it exhibits at least one of the following five characteristics:

- irregular data layout
- an unpredictable communication schedule
- dynamic computation granularities
- unpredictable synchronization patterns
- speculative parallelism

Irregular data layout refers to the use of pointer-based data structures. For example, irregular meshes and sparse matrices are often stored in compact pointer-based representation instead of arrays with simple layout parameters. Irregular data layout creates problems for compile-time analysis of access dependencies and data distribution, because the layout is dependent on the input data. Therefore, optimizing applications with irregular data layout is often performed manually by the programmer. If the communication schedule of the application is static or changes infrequently, runtime preprocessing tools such as PARTI or CHAOS can be used to automate data distribution and optimize communication performance. These tools use an *inspector* primitive to analyze the data layout and produce an optimal communication schedule, which is then used by the *executor* primitive to perform the actual communication. The inspector/executor model depends on having long running computations to amortize the cost of running the inspector.

An application has an *unpredictable communication schedule* if the occurrences of communication events change frequently during its execution. For example, a program that performs dynamic updates on a distributed hash table may generate an unpredictable number of hash table accesses that require communication. In contrast, a program that simulates an irregular mesh generates a fixed communication schedule while the mesh structure is unchanged. Unpredictable communication schedules are expensive to implement because they require hand-shaking between the sender and the receiver for each communication event. Furthermore, on distributed memory architectures with large per-message overheads, an irregular communication schedule makes it impossible to apply compile-time loop transformations such as *message vectorization* [HKT91] or runtime preprocessing techniques such as the inspector primitive to reduce communication overhead.

Dynamic computation granularities arise when the control structure of the computation is highly dependent on the data. For example, in heuristic search problems, a node in the search space may generate an unpredictable number of child nodes, depending on the scheduling and pruning policies. In discrete event simulation, the amount of computation required to simulate an entity may depend on the state of simulation. Dynamic computation granularities create load imbalance, and therefore often require dynamic load balancing to sustain processor efficiency.

An application has a *unpredictable synchronization pattern* if the processors are synchronized dynamically, depending on the state of computation. For example, in discrete event simulation, the simulation of an entity may wait for external conditions such as the arrival of new events and the availability of message buffers. In contrast, in uniform time-step simulation, all entities can proceed independently between barrier synchronizations. Unlike bulk-synchronous applications, asynchronous applications cannot be clearly divided into global synchronization phases, which can be synchronized and load balanced more efficiently. Programming and optimizing such applications is also considerably more difficult.

Finally, applications that exploit *speculative parallelism* may perform redundant computation. For example, a search application using imperfect heuristics may explore parts of the search space that do not contribute to the solution [JY95], and an optimistic discrete event simulator [Jef85, WY93] may waste time on computations that use stale data. Redundant computation presents a tradeoff between parallelism and overhead, because an optimization that increases parallelism may not necessarily improve performance due to a proportional increase in redundant work. Understanding and tuning such applications requires extensive knowledge of the problem.

2.2 The Multipol Data Structure Library

In this section, we present an overview of four data structures in the Multipol library. For each data structure, we describe the programming interface, the implementation techniques, and the driving applications. For each example application, we describe its irregularities and execution behaviors using measurements from parallel executions. Several of the applications are described more completely elsewhere [CDG+93, DDR94, JY95, WY95]. The data structures and applications are then summarized and compared in Section 2.3.

We examine four data structures in increasing "degree of irregularity" of their driving applications: the bipartite graph data structure for bulk-synchronous computation over an irregular mesh, the hash table data structure for distributed set accesses, the task stealer data structure for dynamic load balancing, and the event graph data structure for asynchronous computation over irregular graphs. Five applications are examined: a 3D electro-magnetic field solver (EM3D), a search algorithm solving the tripuzzle game (Tripuzzle), an eigenvalue solver for symmetric tridiagonal matrices (Eigenvalue), a program for building phylogeny trees (Phylogeny), and an event-driven timing simulator (CSWEC). These applications range from the straightforward, bulk-synchronous EM3D program to the completely asynchronous CSWEC program. We implemented the task stealer data structure, the event graph data structure, and the CSWEC application, while other data structures and applications are adapted from the code written by other members of the Multipol group. These data structures and applications provide a fairly complete picture of the target applications of the Multipol library, since their workload covers all the irregularities described in the previous section.

We now introduce some concepts in the Multipol execution model that are used to characterize the behavior of the application. The Multipol runtime layer supports a MIMD programming model. The computation on each processor is decomposed into a collection of independently scheduled threads. The threads can initiate a variety of communication events such as transferring blocks of memory between processors or starting a thread on a remote processor. Each thread is constructed from a collection of *fibers*, each of which appears to execute to completion without preemption or suspension. Put simply, a fiber is the segment of code executed by a thread between two synchronization events. When a thread suspends to wait for a synchronization event, the runtime layer can schedule another thread to hide the synchronization latency. Thread suspension is implemented as a fiber continuation which is triggered by the completion of a synchronization event. To allow latency hiding, many data structure operations have a *split-phase* interface to separate the issue and completion of remote operations; each of the two or more phases executes as a fiber.

Threads, fibers, and communication events are described in detail in Chapter 3. For this chapter, it is sufficient to observe the pattern of their occurrences such as the distribution of the fiber running times, which is a measure of the time between synchronization events, and the frequency and quantity of data communication. In the sections that follow, we use these statistics to characterize the application workload.

2.2.1 Bipartite Graph

The bipartite graph data structure was implemented by Etienne Deprit [YCD+95] based on an implementation by the Split-C group [CDG+93]. It is used to perform bulk-synchronous computation over an unstructured mesh. Each node in the graph has a color, which is either red or black and, by construction, edges only connect nodes of different colors. The state of a node depends on its own state and the states of its neighboring nodes that have a different color. The application consists of bulk-synchronous iterations that compute the states of the red nodes and the black nodes in alternate steps. The data structure can be used in red-black simulation algorithms such as the simulation of electro-magnetic fields described in Section 2.2.1.3.

2.2.1.1 Interface

The bipartite graph data structure allows the programmer to build arbitrary subgraphs on each processor and then connect them to form a distributed graph. Each node in the graph has a user-defined state. Each directed edge contains a weight and a pointer to a copy of the state of its source node. The state of a node can be computed as a function of its own state, the states of its neighboring nodes, and the weights on the edges.

The data structure provides iterators over the nodes and edges for updating of the node states. After an update, the states modified by one processor may not be visible to the other processors. To force a consistent view of the data structure, the programmer must explicitly "validate" the edges by calling a split-phase data structure primitive. When the validation operation completes, all processor views of the graph are up to date.

2.2.1.2 Implementation Techniques

The bipartite graph uses replication to reduce the amount of communication in accessing the node states. A "ghost node" is allocated on each processor for each node that has cross-processor edges. The ghost node has a replica of the node state, and all reads are handled locally using the replica. After a node's state has been updated, the program must call the validate primitive to ensure consistency of the ghost nodes. The data structure assumes a bulk-synchronous computation model, so all processors call the validate operation only after all updates in a phase have been performed. Because of the presence of ghost nodes, the number of messages for validating a node is bounded by the number of processors instead of the number of fanout nodes.

The data structure also takes advantage of the structure of the graph to optimize communication performance. Instead of validating the node states individually, the data structure packages all the validations from a processor to another processor in one physical message. The aggregation overhead is justified by the reduction in communication start-up overhead and message dispatching overhead. Although the Multipol runtime layer is capable of aggregating messages in an application transparent manner, aggregating at the application level may lead to better performance, because the aggregated messages tend to be more compact. The graph structure is assumed to be static, and the implementation techniques are similar to those used in an inspector/executor model.

2.2.1.3 Example Application – EM3D

The EM3D program computes the flow of electro-magnetic waves through a three-dimensional object. 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. The faces form nodes in a bipartite graph, because the computation of a face uses data only from the faces of a different grid. The structure

```
Create a thread on each processor:

    For all time steps:
        Iterate over all local red nodes:
            Compute new state of node.
        [ Validate red nodes ]
        [ Barrier synchronization of all processors ]
        Iterate over all local black nodes:
            Compute new state of node.
        [ Validate black nodes ]
        [ Barrier synchronization of all processors ]
```

Figure 2.1: Pseudo code of the EM3D program. The statements enclosed in square brackets are long latency operations that require synchronization.

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. It is calculated during the first step of an iteration. Similarly, the change in the magnetic field of a node is a linear combination of the electric field of its neighboring nodes, and it is calculated in the second step of an iteration. Since the electric field and the magnetic field are calculated in different steps, dependencies exist only between steps and never within a step.

The Multipol implementation of the EM3D program is simply a sequence of bulk-synchronous phases that alternately compute the states of the two parts of the bipartite graph. Each phase iterates over all nodes in one part in parallel to update their electric or magnetic field. The update of a node uses the local replica of its neighboring nodes. The communication and synchronization required for validating the replica are encapsulated in the bipartite graph. Figure 2.1 shows the pseudo code for the EM3D program.

We ran the Multipol implementation of the EM3D program on a 32 processor CM5 to collect the statistics shown in Figures 2.2 and 2.3 The computation is performed on a random bipartite graph with the percentage of cross-processor edges specified by the user. We tried two settings: 10% and 20% remote edges among all edges in the graph.

Remote Edge	Number of Threads	Number of Fibers	Number of Sync Events	Number of Comm. Events	Total Communication Volume
10%	1401	1801	601	1584	2.75 MB
20%	1357	1730	582	1539	4.59 MB

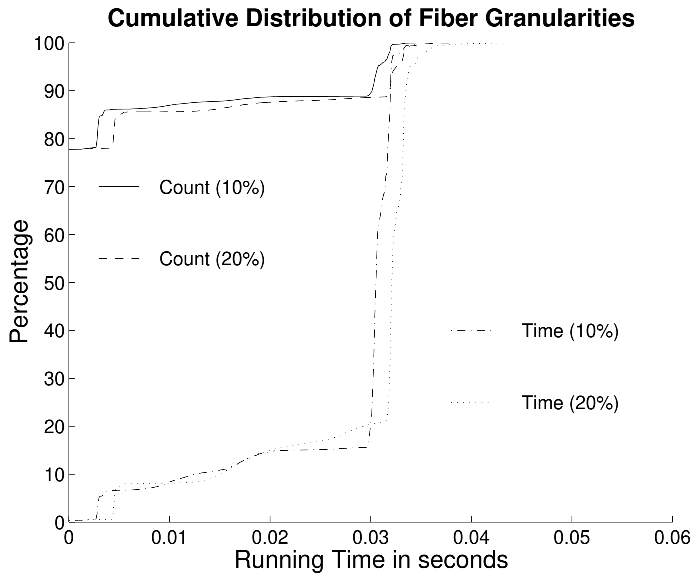
Figure 2.2: Characteristics of two EM3D executions on the CM5. The numbers shown are the averages over 32 processors.

Each processor has 1000 graph nodes with 20 outgoing edges per node. The processors are logically arranged as a linear array, and each edge may adjoin nodes that are at most 3 processors away. The computation is performed for 100 iterations. The statistics collected exclude the construction and preprocessing times of the bipartite graph.

The program features relatively large computations between synchronization events, which are common in bulk-synchronous applications. Although 80% of the fibers execute in 100 microseconds or less, they consume only a small fraction of the total running time. 80% of the execution time is spent in fibers that execute for more than 30 milliseconds. These large fibers compute the new states of the bipartite graph. The rest of the time is spent in fibers running for several milliseconds which are used to validate the node replicas.

The communication traffic consists mainly of large messages. When 10% of the edges are remote, more than 70% of the messages have size greater 2K bytes, and they contribute to almost all the communication traffic. Similarly, when 20% of the edges are remote, the communication traffic is dominated by messages larger than 3.5K bytes. There are relatively few synchronization events in the program.

In summary, the EM3D program is a bulk-synchronous program with irregular data layout. It has a predictable synchronization pattern consisting of bulk-synchronous phases. The communication schedule is also predictable, because it depends only on the structure of the graph which remains static throughout the execution. The application is the least irregular of the five applications we examine. Because all accesses to the graph can be statically predicted, the bipartite graph data structure is able to apply runtime preprocessing to optimize the communication schedule.



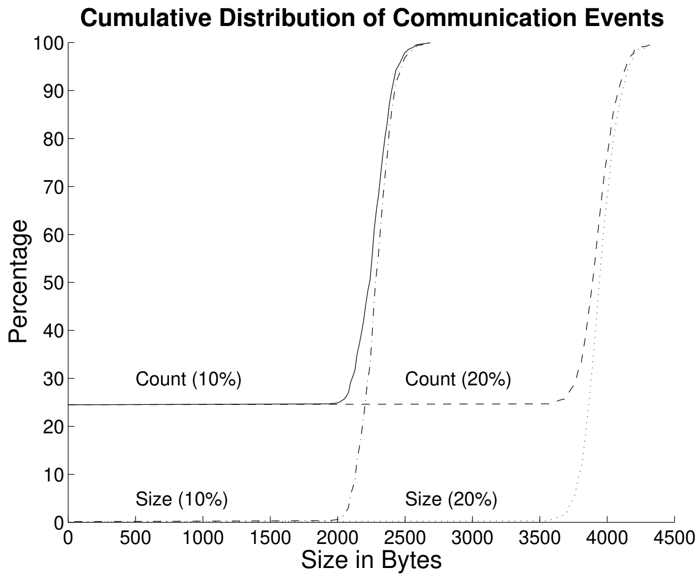


Figure 2.3: Distribution of fibers and communication events for EM3D. The resolution used for profiling the fiber granularity is 100 microseconds, and the resolution for the communication event size is 16 bytes.

2.2.2 Hash Table

The hash table data structure was implemented by Etienne Deprit [YCD+95]. The data structure is used to implement a distributed set of key/value pairs. A set can be used to cache the result of computation or to guarantee the uniqueness of data elements. Its applications include logic verification algorithms [BRB90] and path counting problems such as the Tripuzzle application described in Section 2.2.2.3.

2.2.2.1 Interface

A hash table contains a collection of bins for storing distinct data entries. The assignment of entries to bins and the distribution of bins among the processors are based on a hash function specified by the programmer.

The data structure supports the standard insert, delete, and lookup operations for a dictionary-like object. The programmer may also specify a duplicate handler function, which is invoked when the program attempts to insert an existing item. For example, the program can keep track of the number of duplicates using a duplicate handler which increments a integer counter in the hash table entry.

There are two versions of the insert and delete operations: acknowledged and unacknowledged. An acknowledged access suspends the caller until the access has taken effect, while an unacknowledged one returns control to the caller immediately after the operation is issued. The data structure guarantees that unacknowledged accesses take effect eventually, but not immediately upon their return. For example, a lookup operation following an unacknowledged insert operation may not see the inserted item. Unacknowledged accesses can be used if data races between the mutator and observer accesses cannot happen, or if they do not affect the correctness of the program. The programmer may use a *sync* operation provided by the data structure to force unacknowledged accesses to take effect. The implementation of the sync operation uses the snapshot mechanism provided by the Multipol runtime layer, which is described in Section 3.4.2.

The data structure also provides iterators over the hash table entries for scanning the set. Because the hash table is physically partitioned among all processors, different processors may iterate over different partitions independently to exploit data parallelism over the set elements.

2.2.2.2 Implementation Techniques

The weak semantics of the unacknowledged accesses is designed to reduce synchronization overheads. It eliminates the hand-shaking for the individual insert or delete operations between the calling processor and the processor containing the bin. A group of such operations can be acknowledged efficiently using the sync operation.

2.2.2.3 Example Application – Tripuzzle

The Tripuzzle program computes the number of distinct solutions to the tripuzzle game. A tripuzzle of size N is a triangular board with N rows. The number of columns in each row ranges from 1 in the top row to N in the bottom row. Initially, the board is filled with pegs in all but one position. A legal move may move a peg 2 positions away in a row, column, or diagonal direction, as long as the new position is empty and there is a peg between the old and the new positions. After the move, the peg crossed over by the moved peg is taken out of the board. A solution is a sequence of legal moves that leads to a board configuration with only one peg.

The Multipol implementation of the Tripuzzle program was written by Etienne Deprit [YCD+95] based on a CM-5 implementation by Kirk Johnson [Joh93]. The program performs an exhaustive, breadth-first search to count all possible solutions. The search space is a graph of board configurations where each directed edge represents a legal move. The graph is acyclic because all moves are irreversible (pegs taken out of the board are never put back). It is not necessarily a tree because the same board may be obtained from different boards with more pegs.

The program consists of a series of bulk-synchronous phases. Each phase enumerates all the board configurations that can be reached via a legal move from the boards obtained in the previous phase. That is, each phase traverses one level of the search graph. The program takes advantage of the symmetry of the board to prune the search space by a constant factor. To further prune the search space, the program uses a hash table to eliminate duplicate board configurations and thus avoid searching from the same board twice. Consequently, the number of paths leading to each board configuration needs to be recorded to keep track of the number of solutions. This is achieved by the custom duplicate handler function which accumulates the number of paths for all duplicates in the hash table entry that represents them.

```
Create a thread on each processor:

    Create two empty hash tables called A and B, respectively.
    Insert the initial board in A.
    Set the variable "previous" to A, and the variable "current" to B.
    Set N to the number of pegs in the initial board.

    While N is greater than one:
        Iterate over all boards in the "previous" hash table:
            For all possible moves:
                Insert the resulting board in the "current" hash table.
        Clear all local entries in the "previous" hash table.
        [ Sync all insert operations on the "current" hash table ]
        Swap the values of the variables "previous" and "current".
        Decrement N.

    Iterate over all boards in the "previous" hash table to count solutions.
```

Figure 2.4: Pseudo code of the Tripuzzle program. The statements enclosed in square brackets are long latency operations that may require synchronization.

Figure 2.4 shows the pseudo code of the Tripuzzle program. Two hash tables are used to hold the board configurations enumerated in the previous phase and the current phase, respectively. Only two hash tables are required for the computation, because there is no need to keep track of the old states except for those generated in the previous phase. Unacknowledged insert operations can be used for performance, because all insert operations commute and the same hash table is obtained regardless of the order of insertions. The program uses the sync operation to force all insertions to take effect before entering the next iteration.

We ran the Multipol implementation of the Tripuzzle program on a 32 processor CM5 to collect the statistics shown in Figures 2.5 and 2.6. We use a initial board containing 7 rows of pegs with an empty position in the middle of the 5th row. The results show that the program is dominated by small fibers for handling the hash table insert operations. Over 80% of the fibers executes for less than 200 microseconds, and over 80% of the total fiber running time is contributed by fibers smaller than 600 microseconds. The communication traffic in the program is also dominated by messages carrying less than 32 bytes

Number of				Total
Threads	Fibers	Sync Events	Comm. Events	Communication Volume
22034	27494	1424	19279	0.52MB

Figure 2.5: Characteristics of the Tripuzzle execution. The numbers shown are the averages over 32 processors.

of data. Finally, the program has a clear phase structure which is delimited by the global synchronization events for draining the insert accesses using the sync operation. The sync operations account for almost all of the synchronization events.

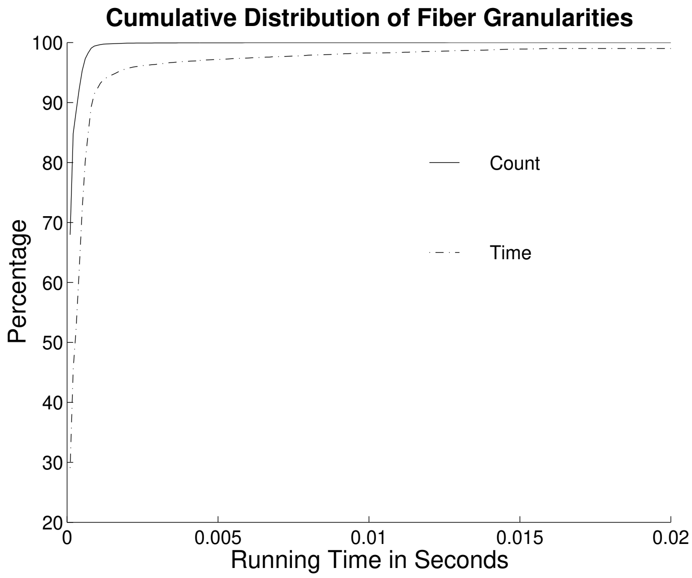
In summary, the Tripuzzle program is similar to the EM3D program in that they are both bulk-synchronous. However, the Tripuzzle program is more irregular because the communication schedule for accessing the hash tables is unpredictable.

2.2.3 Task Stealer

The task stealer data structure was implemented by the author. It performs dynamic load balancing, scheduling, and termination detection. It serves as a common repository for parallel tasks which are represented by a user-defined data structures. The program starts with some initial tasks, which may generate more tasks when processed. The data structure hands off tasks to processors until all scheduled tasks have finished execution and there are no tasks left in the data structure. The data structure also attempts to preserve locality by keeping the tasks on the processors that created them. This is important for applications where migrating a task incurs significant computation or communication overhead. This data structure can be used in applications with dynamic computation granularities, such as divide and conquer algorithms or heuristic search problems.

2.2.3.1 Interface

The task stealer data structure is a collection of task pools, one for each processor in the system. Tasks may be migrated between different task pools to keep the processors busy. Although the data structure represents a single logical pool of tasks, in the implementation the logical task pool is distributed over the processors. This distribution shows through the interface because scheduling is performed separately by the processors, and a processor can



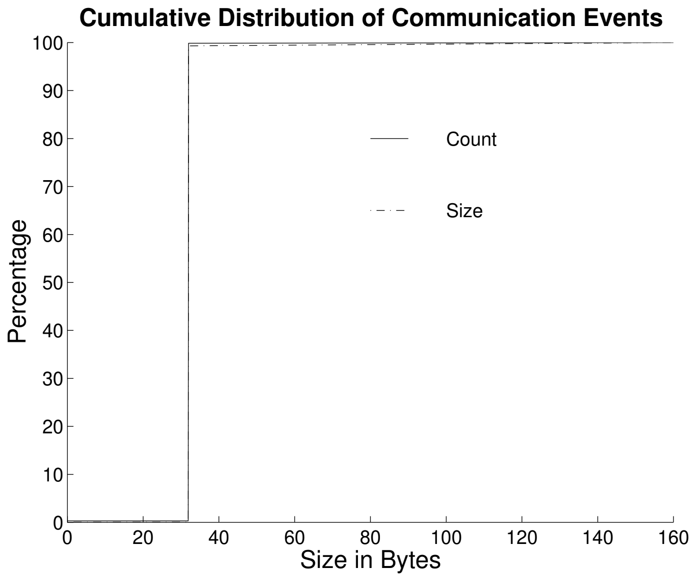


Figure 2.6: Distribution of fibers and communication events for Tripuzzle. The resolution used for profiling the fiber granularity is 100 microseconds, and the resolution for the communication event size is 16 bytes.

be out of work when there is still task in the system. The interface is different from the sequential interface proposed by Ho for task parallelism [Ho94].

An application can add and remove tasks in the task pools. Along with the content of the task, the user may also provide hints for scheduling and load balancing. The hints include the local scheduling priority of the task, the estimated amount of computation performed by the task, and the estimated amount of penalty incurred when the task is migrated. The load balancer uses the latter two estimates to determine what tasks to migrate. Addition of a task is by default to the local task pool, but the programmer may also explicitly specify the destination task pool. Tasks are always removed from the local task pool. If the local task pool is empty, the data structure suspends the calling thread until more tasks are generated by the local processor or migrated from other processors. This suspension, like all suspensions in the Multipol runtime layer, is implemented as a fiber continuation.

The data structure also provides primitives for detecting termination in a multi-threaded execution environment. Termination is established when all task pools are empty and no tasks are in progress. The former condition is automatically determined by the data structure, but the programmer must explicitly establish the second condition by registering the completion of every task obtained from a given task stealer object. The registration is performed by calling a notification primitive on the object. By appropriate use of the notification primitive, termination detection can be performed over multiple task stealer objects.

2.2.3.2 Implementation Techniques

Although the task stealer data structure uses a priority based scheduler for each task pool, the semantics of priorities is weakened to increase parallelism. Priorities are used as a scheduling hint instead of a synchronization mechanism for tasks, since strict enforcement of priorities would imply centralization and thus limits scalability.

The load balancer attempts to preserve locality by migrating tasks on demand, that is, when requested by an idle processor. When there is sufficient parallelism in the system, the processor can retrieve tasks from its local task pool without requiring synchronization. When the load of a processor drops below a certain threshold, the data structure attempts to "steal" tasks from its neighboring processors. To enhance locality, tasks with low migrating penalty or tasks that have been migrated at least once are the preferred targets to steal. The load balancing protocol is completely distributed and thus avoids hot spots.

The programmer can specify a logical network between the task pools for task migration purposes. For example, the task pools in the system can form a ring, a hypercube, or a complete crossbar. Tasks migrate only between neighboring processors. Network topologies with high connectivities may lead to better load balance, but they introduce more overhead in the load balancing protocol.

If locality is not essential, the programmer may also select a randomized load balancer that randomly "pushes" tasks to task pools as they are added to the data structure. Task pushing has been found to achieve better load balance for certain applications we encountered (such as the Eigenvalue program described below). However, the theoretical optimality of both types of load balancing protocols have been reported [BL94, CRY94].

Task migration creates problems for termination detection, because some migrated tasks may be in transit in the network layer when all local task pools are empty. To perform termination detection, the data structure first performs a snapshot on the collection of task pools by temporarily suspending task migration and forcing the delivery of all migrated tasks. The state of the individual task pools can then be probed and combined to determine global termination. If tasks become available when termination detection is taking place, they can be processed immediately without delay. Although our termination detection protocol is not fully incremental [CL85] in that it blocks task migration, it does not impact performance in practice because the protocol is rarely invoked. The snapshot mechanism used for suspending task migration is described in Section 3.4.2.

2.2.3.3 Example Application – Eigenvalue

The Eigenvalue program computes the eigenvalues of an N by N symmetric tridiagonal matrix, which is known to have N real eigenvalues. The program uses the bisection algorithm [DDR94] to approximate the eigenvalues to an arbitrary precision. Given an input matrix, it first computes an initial interval of real numbers that contains all possible eigenvalues for the matrix. The interval is then divided into two half-intervals (hence the name bisection), and the number of eigenvalues in each half is computed. The half-intervals that do not contain any eigenvalue are discarded. The program then performs bisection recursively on the useful half-intervals until the desired precision is achieved.

The bisection of different intervals can proceed in parallel. The processing of an interval completes when it produces two useful sub-intervals or when it becomes a leaf interval. Because the amount of computation required to process an interval is unknown, the program uses the task stealer data structure to dynamically assign the intervals to processors. Pointers to the intervals are placed in the task stealer data structure. The processors repeatedly retrieve pointers from the data structure, fetch the interval, and perform bisection. Each task repeatedly performs bisection on its interval until two useful sub-intervals are obtained. The input matrix is replicated on all processors, so the bisection computation is local once the interval is obtained. The computation kernel of the Eigenvalue program was written by Soumen Chakrabarti [YCD+95]. Inderjit Dhillion implemented a CM5 version of the bisection algorithm prior to the Multipol implementation [DDR94].

Figure 2.7 shows the pseudo code of the Eigenvalue program. A main thread is used to set up the initial interval and perform post-processing after the computation. A computation thread is created on each processor to process the intervals. Either task pushing or task stealing can be used to perform dynamic load balancing. For task pushing, new tasks are added to a random task pool. For task stealing, new tasks are added to the local task pool.

We ran the program using a 1000 by 1000 random matrix on the CM5 to obtain the statistics shown in Figures 2.8 and 2.9. We compare two different load balancing strategies: task pushing and task stealing. The statistics expose the non-uniformity in task granularities. Although fewer than 5% of the fibers run for more than 50 milliseconds, they contribute to more than 90% of the running time. The larger fibers are for intervals where multiple bisections need to be performed to obtain two useful sub-intervals. Their granularities vary between 50 milliseconds and 160 milliseconds.

Most messages generated by the program are small, and their sizes range from 16 to 144 bytes. For task pushing, the program generates a couple communication events per task for randomly distributing the task and fetching the intervals. For task stealing, the program generates slightly more communication because of the task migration protocol. Detailed performance comparison of the two load balancing strategies is given in Chapter 4.

In summary, the Eigenvalue program consists of a collection of parallel tasks that can execute independently without communication. However, the tasks have dynamic computation granularities, and the program requires a dynamic load balancer to sustain pro-

```
Create a "main" thread on processor 0:

    Compute initial interval
    Add pointer to initial interval to local task pool.
    [ Wait until termination ]
    Collect and print solutions ...

Create a "compute" thread on each processor:

    Repeat
        [ Remove a task from local task pool ]
        If task is not generated locally:
            [ Fetch the interval ]
        Repeat
            Perform bisection on interval.
        Until two half intervals are obtained or interval is a leaf.
        If size of interval is within tolerance
            Record solution ...
        Else
            Add pointers to half intervals to some task pool.
```

Figure 2.7: Pseudo code of the Eigenvalue program. The statements enclosed in square brackets are long latency operations that may require synchronization.

Strategy	Tasks	Number of				Total
		Threads	Fibers	Sync Events	Comm. Events	Communication Volume
Stealing	33	421	663	212	163	4.5KB
Pushing	33	321	531	187	135	4.1KB

Figure 2.8: Characteristics of the Eigenvalue execution on the CM5. The numbers shown are the averages over 32 processors.

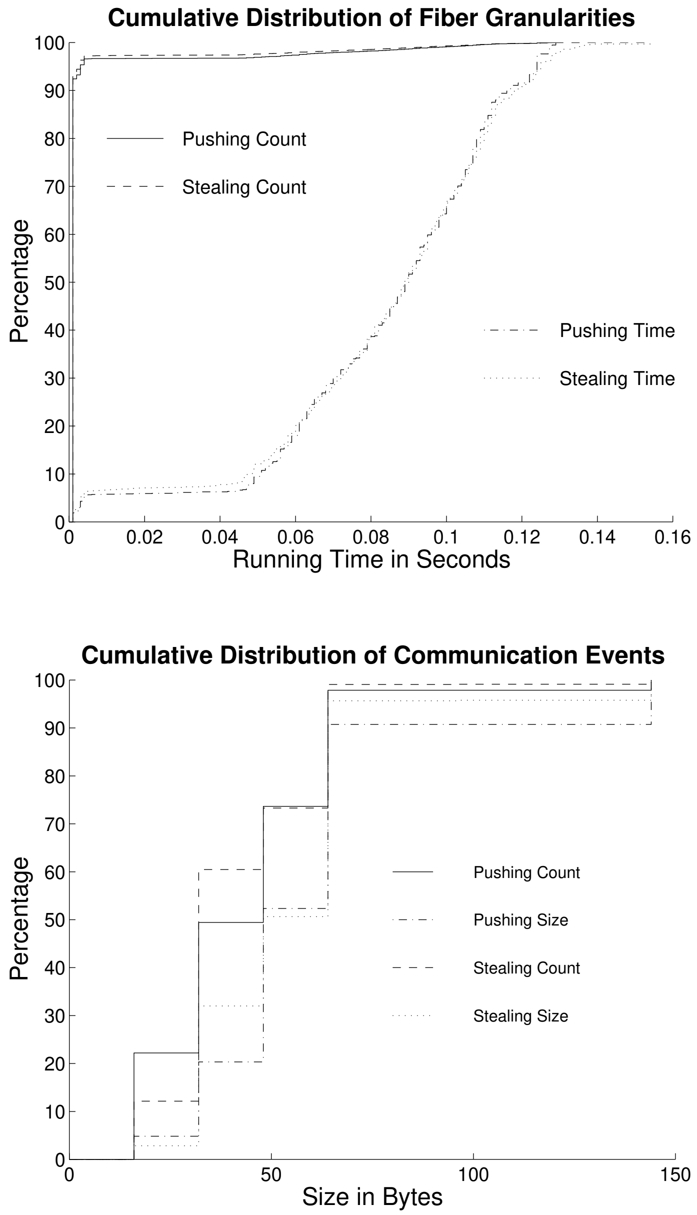


Figure 2.9: Distribution of fibers and communication events for Eigenvalue. The resolution used for profiling the fiber granularity is 1 milliseconds, and the resolution for the communication event size is 16 bytes.

cessor efficiency. The program generates a moderate number of messages which are small in size. The program contains independent synchronization events and does not have a clear phase structure. There are only a few global synchronization events which are used for termination detection.

2.2.3.4 Example Application – Phylogeny

A phylogeny tree for a set of species describes the evolutionary history of the species. Each species is described by a vector of character values called traits. Each character describes an attribute of the species such as its skeletal structure or coloring or a DNA sequence. A perfect phylogeny tree is a special phylogeny tree where no traits in a path arise more than once. That is, once a species loses a certain trait, its descendent species can never regain it. The Phylogeny program finds the maximal character subsets that have a perfect phylogeny tree for a given set of species. The problem is important to a branch of biology known as systematics [JY95].

The Multipol implementation of the Phylogeny program is based on the CM-5 implementation written by Jeff Jones [JY95]. The program performs a search on the space of character subsets to find the subsets that have a perfect phylogeny tree. The program takes advantage of two properties of perfect phylogeny trees in pruning the search space. First, if a set of characters does not have a perfect phylogeny tree (called a failure), none of its supersets have a perfect phylogeny tree. Second, if a set of characters has a perfect phylogeny tree (called a success), all of its subsets have a perfect phylogeny tree. The program maintains a "success store" and a "failure store" of explored character subsets to prune the search space using the above properties. Each store forms a trie, which is built using bit strings representing the subsets.

The root of the search tree is the null character subset. Child nodes are formed by adding one character to their parent node. When exploring a character subset, the stores are first checked to determine if the search subtree rooted at the node can be pruned. If the node survives pruning, the program attempts to construct a perfect phylogeny tree for it. If the attempt is successful, the node is added to the success store, and all its child nodes are generated and scheduled for exploration, since we are interested in the maximal subsets. Otherwise, the node is added to the failure store and its subtree is pruned from the search tree. The maximal subsets can be found in the success store when the search completes.

The program is similar to the Eigenvalue program except that tasks are not completely independent. Each character subset is a parallel task. Because the number and granularities of the tasks cannot be predicted, the program uses the task stealer data structure to dynamically assign the tasks to the processors. Pointers to the character subsets are placed in the data structure, and the processor may need to fetch the subset from a remote processor to process a task. Each processor maintains its own success and failure stores to prune the assigned tasks.

The salient feature of the program is the impact of locality on the amount of redundant computation. Task locality is very important for effective pruning. For example, a failure in the first child of a node may help eliminate the nodes in the second subtree that are supersets of the first child. By keeping the tasks local, nodes in the same subtree stay on the same processor and thus share the same failure store.

The speculative nature of the search algorithm also suggests that combining the stores on different processors may make pruning more effective and thereby reduce redundant computation. The program periodically combines the failure stores by performing a global reduction of the stores on all processors.

Figure 2.10 shows the pseudo code of the Phylogeny program. A main thread is used to set up the initial tasks and perform post-processing after the computation. A compute thread is created on each processor to process the tasks. Either task pushing or task stealing can be used to perform dynamic load balancing. For task pushing, new tasks are added to a random task pool. For task stealing, the tasks are added to the local task pool.

The compute thread periodically creates a combining thread to combine the failure stores. The combine threads first perform a barrier among themselves to ensure all processors are ready to combine the failure stores. The barrier is used to mimic a "consensus" protocol, and its latency can be overlapped with useful work. When the barrier completes, all processors suspend their compute threads to perform the combination synchronously. The combination consists of $log(P)$ iterations, where P is the number of processors in the system. Each iteration progresses one level up the reduction tree.

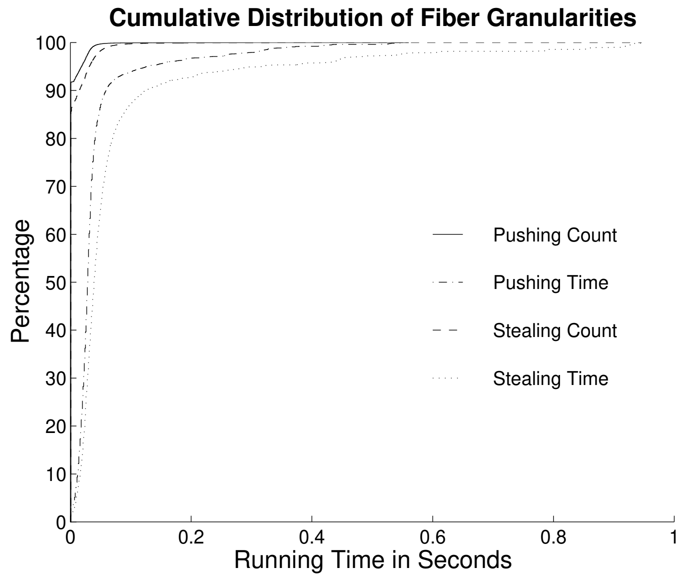
We ran the Phylogeny program on a 32 processor CM-5 using an input with 50 characters and 14 species. Figures 2.11 and 2.12 shows the results of two executions which used task stealing and task pushing, respectively. For task stealing, the computation time is dominated by fibers running for 10 to 50 milliseconds. For task pushing, almost all fibers

```
Create a "main" thread on processor 0:

    Add all single-character subsets to some task pools.
    [ Wait until termination ]
    Collect and print solutions.

Create a "compute" thread on each processor:

    Repeat
        Create a combine thread if necessary.
        If combination is in progress
            [ Wait until combination completes ]

        [ Remove a task from the local task pool ]
        If task is not generated locally:
            [ Fetch the character subset ]

        Process the subset (using the success and failure stores).
        If the subset is a success:
            Form supersets of current subset by adding one character.
            Add supersets to some task pool.

Each "combine" thread does:

    [ Perform global barrier on all "combine" threads ]
    /* Combination in progress */
    [ Perform global reduction on all failure stores ]
    /* Combination completes */
```

Figure 2.10: Pseudo code of the PHYLOGENY program. The statements enclosed in square brackets are long latency operations that may require synchronization.

Strategy	Tasks	Number of				Total Communication Volume
		Threads	Fibers	Sync Events	Comm. Events	
Stealing	2838	6636	11694	3288	2609	2.0MB
Pushing	2862	24856	38319	12795	9139	1.9MB

Figure 2.11: Characteristics of the PHYLOGENY program on the CM5. The numbers shown are the averages over 32 processors.



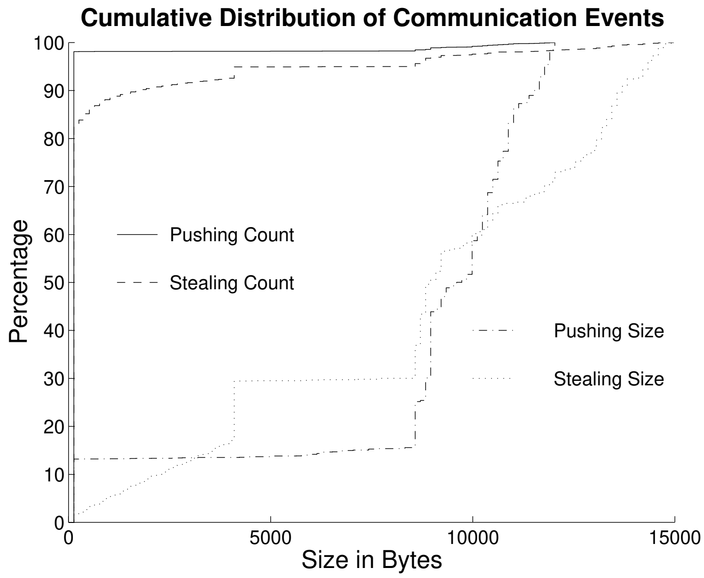


Figure 2.12: Distribution of fibers and communication events Phylogeny. The resolution used for profiling the fiber granularity is 1 millisecond, and the resolution for the communication event size is 128 bytes.

run for less than 1 millisecond. The task pushing version generates more fibers with smaller granularities primarily because most tasks are remote, and more synchronization events are required for fetching the character subsets for the tasks. The task pushing version execute more tasks than the task stealing version because of the lack of locality in task pushing.

The majority of the communication events are small, but the large messages contribute to most of the communication cost. Over 90% of the messages are less than 128 bytes for task pushing, and over 80% of the messages are less than 128 bytes for task stealing. These messages include those generated by the task stealer for dynamic load balancing and termination detection, those generated by the application for fetching remote character subsets, and those used for coordinating the combination of the failure stores. The task stealing version generates some 4K byte messages for migrating the tasks in batches. Messages larger than 8.5K bytes are used to fetch the failure stores, and they account for more than 85% and 70% of the total communication volume for task pushing and task stealing, respectively. Overall, the tasking stealing version generates fewer small messages because most tasks can be processed locally.

The program contains independent synchronization events for accessing the task pool and fetching the character subsets. The program also contains bulk synchronization events for coordinating the combination of failure stores.

In summary, the Phylogeny program consists of a collection of parallel tasks that can be scheduled independently, but are most efficient if they share information. The tasks have unpredictable quantity and granularity, and the program requires a dynamic load balancer to sustain processor efficiency. Some of the tasks are speculative, and the load balancer must attempt to preserve locality to reduce the amount of communication and redundant work. The program has a irregular communication schedule which contains both small and large messages; the majority of the communication events are small, but large messages contribute to most of the communication cost. The program contains both independent synchronization events and global synchronization events.

2.2.4 Event Graph

The event graph data structure was implemented by the author. It is used to perform asynchronous parallel computation on a irregular graph. Unlike the bipartite graph data structure where the nodes synchronize in a bulk-synchronous manner, nodes in the event graph synchronize independently. Each directed edge in the event graph is essentially a communication channel for passing *event messages* from the source node to the sink node. We use the term event messages to refer to the application-level messages in the event graph, which are different from the physical messages in the network layer. 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*). It also enforces data dependencies and resource dependencies by suspending threads that attempt to send an event message that overflows the capacity of an edge or threads that attempt to retrieve an event message from an empty edge. The data structure can be used in applications that have pipelined or producer/consumer parallelism such as discrete event simulation.

2.2.4.1 Interface

The edges of the event graph can be thought of as fixed-sized FIFO buffers with which the nodes send and receive event messages. The size of the buffers is called the capacity of the graph, and is specified by the programmer upon creation of the graph. The setting of capacity determines the tradeoff between parallelism and memory overhead. The higher the capacity, the fewer the synchronization events for flow control and the higher the parallelism between the sending and receiving nodes. However, if the capacity is too high, thrashing in the memory hierarchy may occur when the amount of allocated memory exceeds the physical memory of the system. See Section 5.1.5 for an example.

The nodes in the graph are the end points of communication in the program. They also represent the units of computation required to process the communication events. The nodes are statically distributed among the processors by a partitioner function supplied by the programmer. The partitioner encapsulates the programmer's knowledge of the application on the tradeoff between load imbalance and communication overhead.

After the graph is created on all processors, the program can issue four types of operations: send an event message, examine and/or remove a event message, wait for a new event message, and take a snapshot of the state of the graph.

When a node sends an event message, the message is multicast along all outgoing edges of the node. The data structure suspends the sender if any of the outgoing edges runs out of capacity. When the call returns, the data structure guarantees that the event message will eventually be delivered, and the order of arrival will match the sending order. The completion of the call does not necessarily imply that the event message has been received.

The receiving node sees a FIFO queue of event messages for each incoming edge. The received messages can be examined out of order, but they must be removed in the order they are sent. The receiving node may suspend itself until some event message is available. The programmer has the option of resuming the receiver when a message arrives at an empty edge, or when a message arrives at any edge. The former generates fewer synchronization events, and it can be used in applications where the event messages must be processed in order, such as in the CSWEC program described in Section 2.2.4.3.

Because the event graph is distributed, there is no way for a single processor to determine when the graph is in a globally consistent state, that is, when all event messages have been received. Certain applications require a consistent snapshot of the graph state to compute global properties, such as the presence of deadlock or the progress of computation. To support such applications, the data structure provides a pair of primitives called *freeze* and *unfreeze*, which are used to create a window of time when the state of the graph is globally consistent. After invoking the freeze operation, the program can examine the individual node states in an arbitrary order and obtain the same snapshot. The snapshot mechanism is a component of the runtime layer and is described in detail in Section 3.4.2.

2.2.4.2 Implementation Techniques

The event graph data structure uses the following techniques to improve performance. First, like the unacknowledged accesses in the hash table data structure, the semantics of a send operation is weakened to reduce synchronization overhead. The weak semantics eliminates the hand-shaking between the sending and the receiving nodes for acknowledging event messages. The application can call the freeze primitive to force the delivery of all event messages, for example, when taking a snapshot of the graph.

The data structure 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, such as in the CSWEC program described below.

2.2.4.3 Example Application – CSWEC

The CSWEC program was implemented by the author, based on the sequential SWEC program written by Lin [LMSK91]. 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. Longer time steps are used for subcircuits that have infrequent activities, while shorter time steps are used for highly active subcircuits to maintain the desired precision. 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 computation because digital signals change infrequently. The simulation algorithm has been shown to be both accurate and efficient in the work by Lin et al. [LMSK91].

The parallel implementation of the CSWEC program is a classical example of parallel discrete event simulation. We adopt the *conservative* approach to parallelizing asynchronous simulation [CM81]. A graph node is allocated for each input signal port and each voltage point of a subcircuit. A thread is created for each subcircuit to simulate its time-varying state. The threads communicate and synchronize via the event graph data structure, which encapsulates the connectivity structure of the circuit.

A thread suspends until all the required inputs are present, that is, when the minimum time of the input events is no less than the next time step of the subcircuit. The thread then simulates the subcircuit for a time step, and if the new state needs to be propagated, it sends the new state via the event graph. The event graph ensures all events are properly ordered at the receiving end and the memory used by the simulation is bounded.

Since we target combinational circuits, the data dependence graph of the subcircuits is acyclic, and deadlock due to the lack of global information cannot occur. However, because the event graph places a upper bound on the number of outstanding event messages, subcircuits sending new events may wait for its fanout subcircuits to release the required buffer space, creating resource dependencies. The overall dependence graph of the subcircuits may have cycles and therefore deadlock due to both data dependencies and resource dependencies. To ensure the progress of simulation, we use both *null messages* and deadlock recovery. A null message is an event that carries only the simulation time. A subcircuit sends a null message if it progresses for several time steps without producing any event. Deadlock recovery is used to ensure correctness, but in practice our heuristics generates enough null messages to eliminate the need for deadlock recovery.

Figure 2.13 shows the pseudo code of the CSWEC program. The event graph data structure propagates both event and null messages. The compute thread may suspend to wait for new event messages, for buffer space to send event messages, or for buffer space to send null messages. The latency of these operations can be overlapped with the simulation of other subcircuits. The recovery thread remains inactive until the processor runs out of compute threads to execute. The thread then performs a global barrier operation to synchronize with all deadlock recovery threads. If some subcircuit threads become available for execution, they can be scheduled to overlapped the latency of the barrier, and thus temporarily suspend the deadlock recovery process which has become unnecessary. If all processors are out of work, deadlock recovery takes place eventually to unblock the simulation.

We ran the CSWEC program on a 32 processor CM5 with two input circuits: a 32-bit register file (called REGFILE) and an unknown circuit from the ISCAS benchmark suite (called C2670). The REGFILE circuit has 325 subcircuits and 4832 transistors, and the C2670 circuit has 2033 subcircuits and 5364 transistors. The results are shown in Figures 2.14 and 2.15.

The computation granularities for the REGFILE circuit follow a bimodal distribution. 70% of the fibers ran for less than 100 microseconds, while the total running time is dominated by the remaining 30% of the fibers which ran for more than 5 milliseconds. The non-uniformity of computation granularity comes from the irregularities in subcircuit size – the circuit contains 32 large subcircuits that take up most of the transistors. The C2670 circuit also has non-uniform computation granularities. 70% of the fibers run for 100 microseconds or less, and they contribute to only 10% of the total time. The rest of the time is contributed by fibers running between 0.1 to 10 milliseconds.

Although the fiber granularities are non-uniform for both circuits, they are roughly proportional to the sizes of the subcircuits they simulate. Therefore, we statically assign

```
1. Create a "compute" thread for each subcircuit to simulate its state:

   Repeat
     [ Wait until a new event message arrives at an empty edge ]
     While the minimum time event message has time less than next time step:
         Incorporate the fanin subcircuit state carried by the message.
         Decrease next time step to maintain accuracy.
         Advance edge time to message time.
         Remove message.

     While minimum edge time is no less than next time step:
         Compute new state of subcircuit at next time step.
         Advance next time step.
         If new state cannot be linearly extrapolated from old state:
             [ Send new state to all fanout subcircuits ]

     If subcircuit has advanced without generating events:
         [ Send null messages to all fanout subcircuits ]
   Until simulation is done.

2. Create a "recovery" thread on each processor for deadlock recovery:

   Repeat
     [ Wait until no subcircuit on the processor can progress ]
     [ Perform global barrier among all deadlock recovery threads ]
     Find minimum time of all local subcircuits
     [ Perform reduction to find globally minimum time ]
     For all local subcircuits:
         Advance edge time to globally minimum time if applicable.
   Until simulation is done.
```

Figure 2.13: Pseudo code of the CSWEC program. The statements enclosed in square brackets are long latency operations that may require synchronization.

Circuit	Number of				Total
	Threads	Fibers	Sync Events	Comm. Events	Communication Volume
REGFILE	2376	5656	463	1720	40KB
C2670	73111	113148	10363	59009	1.38MB

Figure 2.14: Characteristics of the CSWEC program on the CM5. The numbers shown are the averages over 32 processors.

the subcircuits to the processors based on their sizes. Although the actual amount of computation required by the subcircuits is dependent on the input signals, we chose not to use a dynamic load balancer because each subcircuit has a substantial amount of state, and the communication penalty for migrating subcircuits is high.

Almost all communication events are between 16 to 32 bytes in size, and they are generated by the event graph data structure. By eliminating redundant event messages, the data structure successfully reduces the number of event messages by more than 95% for both circuits.

The number of synchronization events depends on the amount of memory allocated for the simulation. Figure 2.16 shows the characteristics of the executions when the amount of memory allocated is only 1/4 of that in Figure 2.14. The results show a significant increase in synchronization events due to the decrease in the parallelism between the sender and receiver subcircuits.

In summary, the CSWEC program is highly irregular because it exhibits irregularities in data layout, communication schedule, computation granularities, and synchronization pattern. The communication traffic is dominated by small messages, and the program contains independent synchronization events whose number decreases as the amount of memory allocated increases. The asynchronous structure of the program also requires a multithreaded execution model for scheduling and latency hiding. The CSWEC program is the most challenging application we have examined.

2.3 Summary and Comparison

In this section, we summarize our experiences with the Multipol library and point out the runtime support required to build irregular applications. We first give an reca-

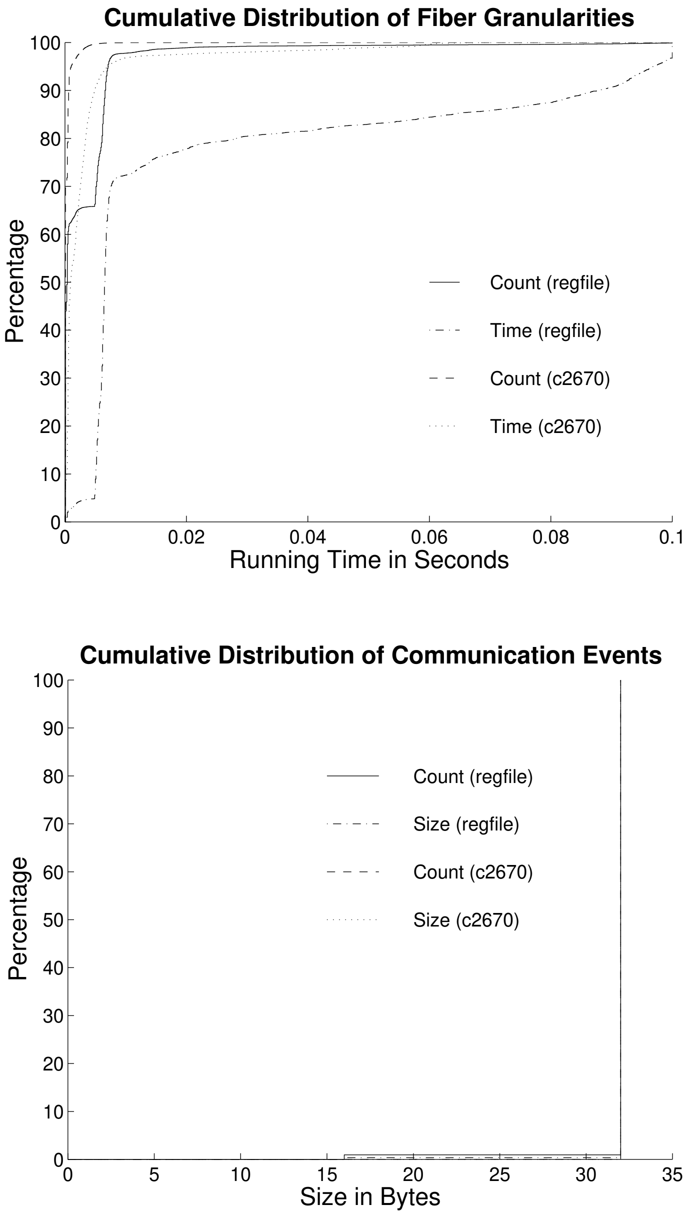


Figure 2.15: Distribution of fibers and communication events for CSWEC. The resolution used for profiling the fiber granularity is 100 microseconds, and the resolution for the communication event size is 16 bytes.

Circuit	Number of				Total Communication Volume
	Threads	Fibers	Sync Events	Comm. Events	
REGFILE (small)	3815	7920	1405	1959	45KB
C2670 (small)	109937	169567	32200	68719	1.59MB

Figure 2.16: Effect of reducing the memory allocation on simulation. The amount of memory is reduced to 1/4 of the original size.

Irregularities	Applications				
	EM3D	Tripuzzle	Eigenvalue	Phylogeny	CSWEC
Data Layout	Yes	Yes	–	–	Yes
Communication schedule	–	Yes	Yes	Yes	Yes
Computation granularities	–	Some	Yes	Yes	Yes
Synchronization pattern	–	–	Some	Some	Yes
Speculative parallelism	–	–	–	Yes	–

Figure 2.17: Irregularities in the examp le applications.

pitulation of the irregularities arising in the five example applications. We then compare their work load and discuss the performance implication. Finally, we suggest the runtime infrastructure required to build such applications.

2.3.1 Application Characteristics

Figure 2.17 gives a summary of the irregularities arising in the five example applications. It shows that the data structures in the Multipol library cover a wide range of irregular applications. Figure 2.18 compares the application workload in the following three aspects: computation, communication, and synchronization.

The computations 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 and 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 with its replicated matrix, so randomized task pushing can be used to improve load

Applications	Workload		
	Computation	Communication	Synchronization
EM3D	Statically balanced using knowledge of input	Some large predictable events	Predictable global events
Tripuzzle	Statically balanced by randomization	Mostly small unpredictable events	Predictable global events
Eigenvalue	Dynamically balanced, locality not important	Some small unpredictable events	Independent events
Phylogeny	Dynamically balanced, locality is important	Some small and large unpredictable events	Independent and global events
CSWEC	Statically balanced using heuristics	Many small unpredictable events	Independent events

Figure 2.18: Comparison of application work loads.

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 of communication 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 performance of such applications are limited only 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 and the CSWEC program, the performance is likely to be limited by the communication start-up overhead.

Synchronization events can take place independently for the individual accesses, or globally for a collection of accesses. 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.

2.3.2 Required Runtime Support

Our experiences with the Multipol library and applications suggest the following runtime infrastructure for building irregular parallel programs.

First, a multithreading execution environment is required for hiding the communication and synchronization latency. In addition to simple reads and writes, distributed data structures may perform arbitrary computations on a remote processor, such as a dictionary lookup. Remote accesses may also wait for synchronization events that have indefinite latency, such as waiting for event messages or remote buffer space in the CSWEC program. Multithreading can be used to hide the latency of such operations.

Second, the overhead of small unpredictable communication events needs to be addressed. In addition to the hand-shaking overhead involved in setting up an unpredictable message, most distributed memory architectures have poor bandwidth characteristics for small messages. To improve performance, the communication workload of applications such as Tripuzzle and CSWEC must be transformed into a workload that can be efficiently supported by the machine platform, using techniques such as automatic message aggregation, which is discussed in the next Chapter.

Finally, all distributed data structures share the same *split-phase interface*, which separates the issue and completion of remote accesses to hide latency. The implementation of the data structures can also share the infrastructure for naming, data distribution, concurrency control, and synchronization.

Although none of the applications described in this work make use of dynamic caching, the Multipol library contains a data structure called the weakly consistent object layer, which has successfully applied caching to a Gröbner basis program. The program uses a kind of speculative parallelism that is similar to a search problem, but with more subtle forms of pruning [CY93].

2.4 Related Work

Fox [Fox92] classified parallel applications based on their temporal structures. He put all applications in three categories: synchronous, loosely synchronous, and asynchronous. He described example applications in each category and discussed their degree of irregularity. Instead of directly classifying applications, we enumerate the common sources of irregularities and use the application workload to quantitatively characterize them. Our approach provides more insight into the techniques for optimizing irregular applications.

There has been a great deal of work on specialized libraries for scientific applications. Examples include LAPACK [Dem89, ABB+92] and ScaLAPACK [CDPW92], which are libraries of SPMD algorithms for solving linear algebra problems such as matrix multiplication, matrix factorization, and eigenvalue computations. Unlike the Multipol library, they contain standalone procedures rather than data structures and they solve mostly regular problems.

PARTI [BSS91] and CHAOS [DHU+93, DUSH94, MSH+95] (an enhanced version of PARTI) are runtime layers designed to support simulations on spatially irregular data structures. They generate optimized *communication schedules* to reduce the overhead of accessing such data structures. LPARX [KB95] is a library for adaptive mesh computation. It provides primitives for building adaptive meshes, partitioning the computation, and accessing the mesh state. These libraries use various optimization techniques to resolve the performance inefficiencies due to irregular data layout and uneven computation granularities, but they are limited to bulk-synchronous algorithms and numerical applications. In contrast, Multipol is designed for a more general class of irregular problems which includes asynchronous algorithms. Out of the five applications we examined, only the EM3D application can be optimized using libraries such as CHAOS or LPARX.

Ho [Ho94] developed a toolbox for parallelizing symbolic applications in Lisp. His toolbox provides two types of abstractions: parallelism and data sharing abstractions. The parallelism abstraction is similar to our task stealer data structure. The data sharing abstractions include automatically locked objects and speculative read-modify-write operations. The toolbox is similar to the Multipol data structure library in that they both support irregular, asynchronous applications. However, Ho's toolbox is designed for shared-memory platforms and emphasizes the importance of a single-threaded, sequential interface for programmability. Some of the data structures in the Multipol library relax the consistency requirements and sacrifice the sequential interface to allow for more efficient implementations. For example, the task stealer data structure has a termination detection interface that exposes the multithreaded execution model.

Many load balancing protocols for distributed memory platforms have been proposed. Their major differences are not in the algorithms, but in the surrounding software support. Languages and runtime layers such as Charm [SK91, KK93] and Cilk [BJK+95] have built-in language mechanisms to allocate parallel tasks that are automatically migrated by a embedded load balancer. These systems have a fixed set of load balancing and scheduling policies that cannot be easily customized for a particular application. Our experiences with irregular applications indicate that it is essential to use different load balancing and scheduling policies for different applications. For example, the Eigenvalue program performs better with task pushing, while the Phylogeny program performs better with task stealing. Static load balancing is sometimes better, even for problems with unpredictable task times, as in the CSWEC program where migrating the subcircuits incurs high penalty. In Multipol, we use the library data structures to perform load balancing. The programmer can choose from a variety of load balancing policies provided by the library or implement a customized load balancing policy for a particular application.

Chapter 3

Multipol Runtime Layer

Distributed memory multiprocessors have been the dominant architecture for medium to large scale parallel computing. Workstations connected by high bandwidth networks are also gaining popularity. However, such architectures tend to encourage programs with bulk, predictable communication patterns. They do not match the characteristics of irregular applications with fine-grained, asynchronous communication. Direct mapping of irregular communication patterns on distributed memory machines often leads to poor performance.

Portability is also a concern in developing irregular applications. Irregular parallel programs are mostly written with explicit communication primitives from the vendor's communication library, which varies with the machine. The differences in programming environments create problems for code sharing across machines. MPI [For94] is a message passing standard which provides a uniform communication interface on distributed memory platforms. But the MPI primitives are designed for cooperative message passing and therefore do not support irregular communication patterns well.

In this chapter, we describe the Multipol runtime layer for building irregular applications on distributed memory architectures. The runtime layer underlies the data structures described in Chapter 2, but it can also be used directly by the application programmer. It has two main components: a thread layer and a communication layer. The thread layer addresses the performance issue by allowing the programmer to overlap the latency of remote accesses with local computation. It also simplifies programming by providing a remote invocation mechanism. The communication layer addresses the portability issue by providing the programmer with a uniform programming interface on machines with different native communication mechanisms. It also optimizes irregular communication schedules for efficient execution on distributed memory platforms with different computation to communication cost ratios.

The thread layer and the communication layer are tightly integrated to facilitate concurrency control and scheduling. The thread system provides a basic programming abstraction called fibers which have guaranteed atomicity. Communication events take place asynchronously in the form of remote fiber invocation without disrupting concurrency control. The fibers can be scheduled using the default schedulers provided by the runtime layer or the customized schedulers written by the programmer. The runtime layer takes advantage of semantics of the schedulers to optimize performance.

The runtime layer also provides some system data structures for managing the distribution of data structures and performing distributed snapshots. They are the basic build blocks for other higher-level data structures.

The rest of the chapter is organized as follows. Section 3.1 gives an overview of our approach. Section 3.2 presents the thread layer, covering issues in concurrency control, scheduling, and programming. Section 3.3 describes the interface and implementation of the communication layer. Section 3.4 describes the data structures provided by the runtime layer. Section 3.5 describes the implementation of the runtime layer on different types of machines. Section 3.6 describes related work, and Section 3.7 summarizes the chapter.

3.1 Overview

In this section, we point out the mismatch between the characteristics of distributed memory architectures and the workload of irregular applications. We then describe our approach to resolving this mismatch.

3.1.1 Characteristics of Distributed Memory Architectures.

A distributed memory machine consists of a collection of processors, each of which has its own memory module. Without a shared address space, accessing remote memory requires executing code on the processor owning the memory. The accesses can be performed by cooperative message passing or active messages [vECGS92]. To support messages of an arbitrary size, the buffer space used for each message must be explicitly allocated before the communication takes place.

Machine	Network	Nodes	Comm. Layer	Overhead	Latency	Bandwidth
CM5	Fat-tree	32	CMAML	4.3 us	14.5 us	10 MB
Paragon	Mesh	8	NX	91 us	164 us	39 MB
SP1	Multi-stage	8	MPLp	44.8 us	68.3 us	8.5 MB
Sparc cluster	Crossbar	8	TCP/IP	$\simeq$ 0.8 ms	$\simeq$ 1.6 ms	9 MB

Figure 3.1: Communication characteristics of 4 distributed memory machines. The overhead is the processor time for sending and receiving a 0-byte message. The latency is the round-trip time (including overhead) between sending a request message and receiving a reply message. The bandwidth is the peak point-to-point bandwidth attainable by the communication library.

The processors can be connected by a variety of networks such as the fat-tree interconnect of the CM5 [LAD+92], the mesh interconnect of the Paragon [LC95], and the switched LAN (e.g., Myrinet [BCF+95]) of the Sparc cluster built by the NOW [CLMY96] group at Berkeley. The native communication libraries on these machines typically have high start-up overhead, and therefore bandwidth increases with the size of the physical message. Communication latency varies with the physical distance between the nodes, but the difference is usually negligible compared to the communication overhead.

Figure 3.1 shows the communication performance of the machines used in this work. For portability, we use the communication libraries provided by the vendors instead of the research prototypes such as the generic active messages (GAM) [CKK+94] developed by the NOW group. The measurements for the CM5, the Paragon, and the SP1 are from Luna [Lun94], and the measurements for the Sparc cluster are from Keeton et al [KAP95]. On every machine, the communication time for small messages is entirely dominated by the send and receive overheads on each end.

3.1.2 Our approach

Although distributed memory architectures provide high performance and scalability, they do not provide an adequate programming model for irregular applications. Two issues arise from the distributed memory model, namely the latency of remote operations and the overhead of communication. As described in Chapter 2, accesses to distributed data structures are often performed by invoking computation on a remote processor, and the latency observed by the processors for these accesses can be high. Many irregular applications also generate unpredictable communication events that are small in size, a communication pattern that cannot be supported efficiently by most message passing libraries. To address these two issues, the Multipol runtime layer uses the following two techniques: multithreading and message aggregation. Both techniques reply on the availability of excess parallelism in the application.

The runtime layer provides a simple user-level thread layer for hiding the latency of remote operations. Instead of waiting for a remote operation to complete, the processor simply switches to another available thread to continue execution. The threads are built on an abstraction called *fibers* which appear to execute without interruption until completion. Fibers can be scheduled freely to fill latency gaps.

The runtime layer also performs automatic *message aggregation* to accumulating small, asynchronous messages into large physical messages. Message aggregation improves communication efficiency by amortizing the communication start-up overhead over a larger amount of payload. The optimization trades the excess parallelism from the multithreaded execution model for higher communication bandwidth.

Figure 3.2 illustrates our approach. The left side of the figure represents the workload of a conventional single-threaded program which features many small messages and synchronization events. The right side of the figure represents the desirable workload for distributed memory architectures, which favors bulk communication and synchronization. The mismatch between the two workloads causes the program to run with poor efficiency. The goal of the Multipol runtime layer is to provide the mechanisms for dynamically transforming the program workload into a workload that can be executed efficiently on conventional distributed memory machines.

The Multipol runtime layer also enables the same program to execute on a variety of distributed memory machines. Figure 3.3 illustrate the portability layers in the runtime layer. The applications and data structures are built on a portable programming interface provided by the runtime layer. The runtime layer itself is also built on a common machine interface to facilitate porting. The multithreaded execution model and the optimized communication layer enable Multipol programs to have *performance portability*, that is, a single application implementation may be used without changes to achieve high performance across platforms.

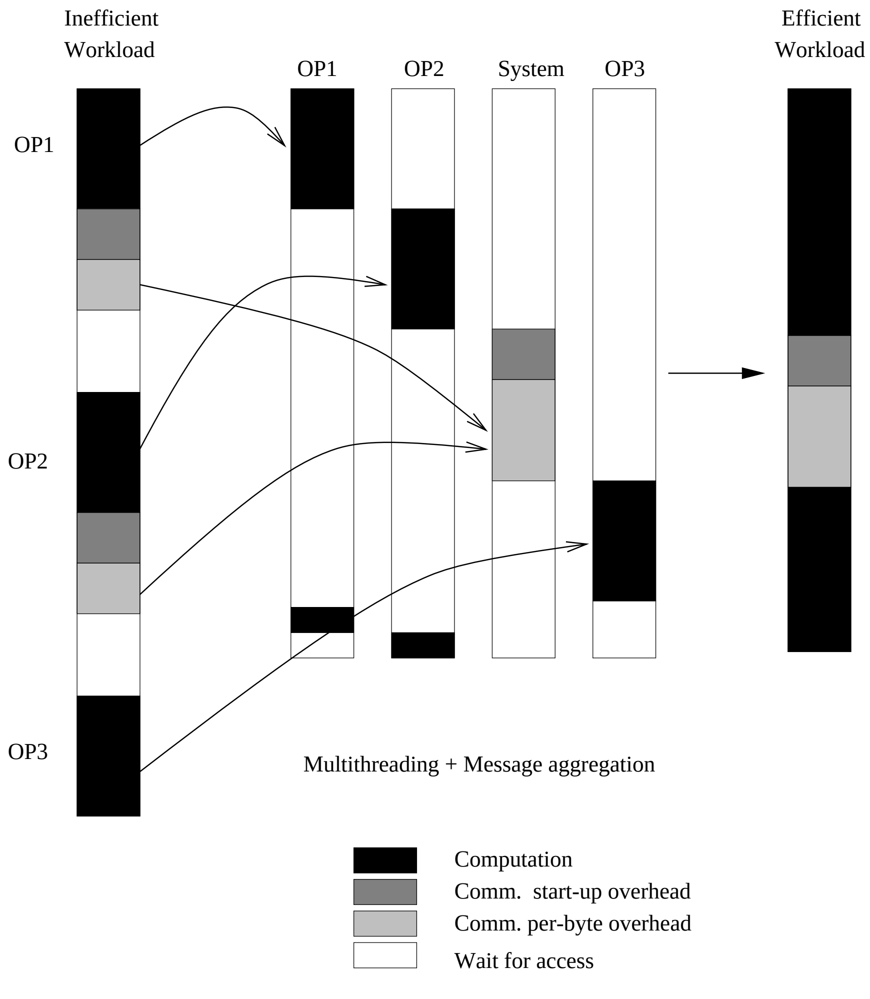


Figure 3.2: Improving efficiency with multithreading and message aggregation. Multithreading hides the latency of synchronization events (the white area in the left figure), and message aggregation reduces the communication start-up overhead (the dark gray area). The result is the more efficient workload shown in the right figure.

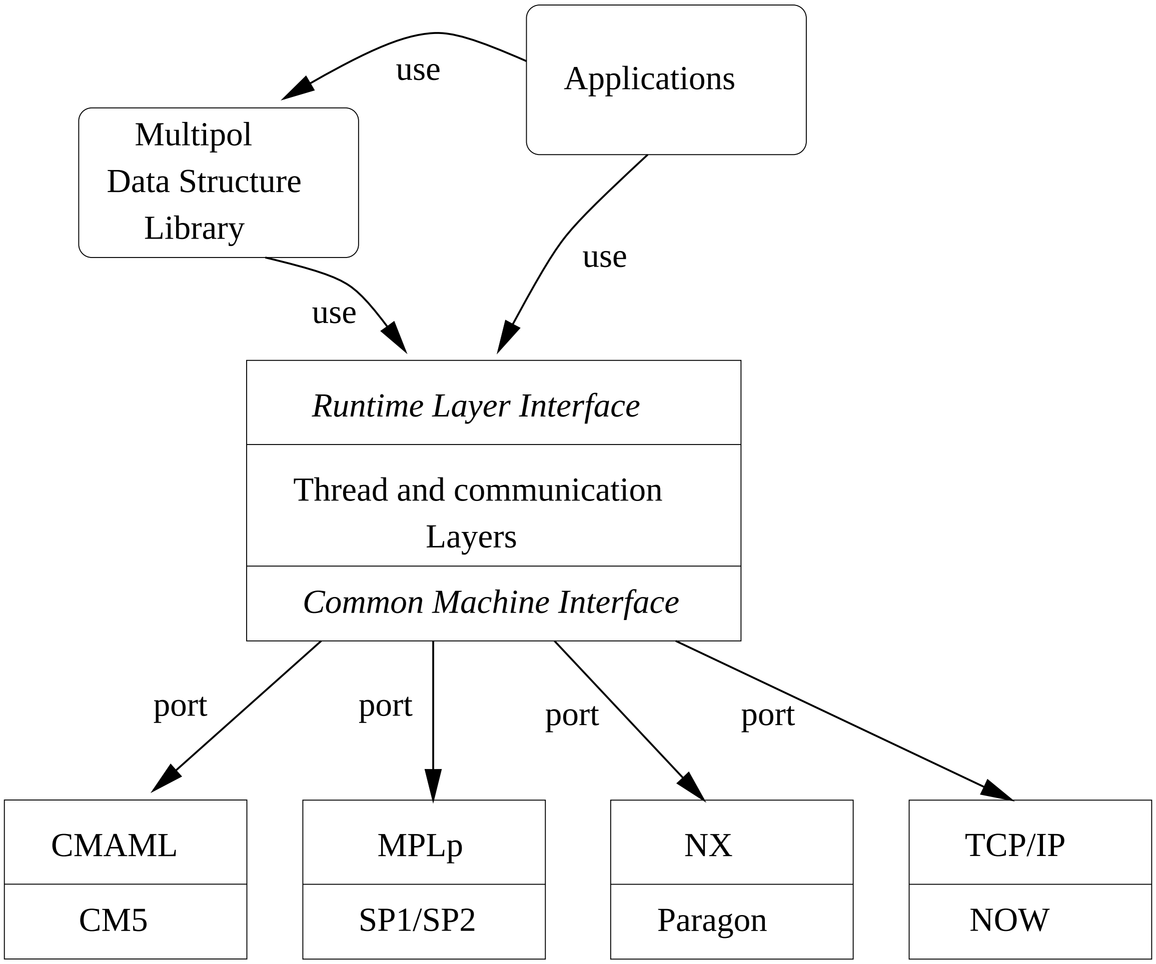


Figure 3.3: Portability layers in the Multipol runtime layer.

3.2 The Thread Layer

In this section, we present the Multipol thread layer and illustrate its use in implementing distributed data structures.

3.2.1 Threads and Fibers

The Multipol runtime layer supports a MIMD programming model. The computation on each processor proceeds by spawning threads that are used for simulated parallelism on that processor. When a thread suspends to wait for a synchronization event, the runtime layer schedules another available thread for execution to keep the processor busy.

Threads are not directly supported by the runtime system, but are built from fibers. Fibers provide the programmer with a computational abstraction that appears to execute atomically. Each thread then consists of a sequence of fibers separated by synchronization events, with the invocation of a fiber in the sequence acting as a continuation of the thread. The atomicity of fibers helps concurrency control for accessing data structures that are distributed over the processors; different fibers can update different partitions of the data structure concurrently without requiring locking, because each fiber has exclusive access to its partition when it executes. The non-blocking semantics of fibers also facilitate scheduling and performance profiling. Although the runtime layer does not provide internal support for thread context management, we provide a set of macros to help the construction of threads from the fibers. A prototype compiler was developed by Jones and Papavassiliou [JP95] to automatically converts programs with blocking accesses into programs with only split-phase accesses and fibers, but in this thesis, all of our programs use hand-coded fibers or fibers generated by the macros.

To create a fiber, the programmer specifies a function and its actual arguments. Once scheduled, the fiber executes the function with the arguments. Besides a few words of data for book-keeping purposes,[1] the function and the arguments comprise the entire state of a fiber. Stack frames needed by a fiber are allocated as normal function calls on the program stack; because fibers run to completion, the registers and stack frames associated with a fiber never need to be saved. Therefore, the fiber abstraction provides a light-weight thread layer without requiring machine-dependent code for managing processor states.

Creating a fiber involves allocating its state and copying the input arguments.

[1]These include the total size of the arguments and a few integers for profiling purposes.

Since many fibers are used to implement thread continuations, the runtime layer allows a executing fiber to directly pass on its state to a new fiber upon termination. Creating a continuation fiber incurs lower overhead because the new fiber simply reuses the space of the terminated fiber without requiring buffer allocation or copying.

During the life-time of a fiber it can be in one of three stages – *created*, *enabled*, and *executing*. Typically, a fiber in the created stage is waiting for a synchronization event. It has its associated state allocated, but is not eligible for execution until it is explicitly enabled. A fiber in the enabled stage may be scheduled for execution at any time.

Threads and fibers are used only for latency hiding. They are not meant to be used as units of true parallelism or load balance and they do not migrate across processors. Unlike other compilers or runtime systems with built-in load balancers [SK91, BJK+95, GSC95, MKH91, CR95], load balancing mechanisms are left out of the Multipol runtime layer so that the programmer can have more control over locality and scheduling. Load balancing can be performed by the task stealer data structure or other data structures implemented by the user. The programmer can reduce the runtime overhead for managing parallelism by creating parallel tasks only when necessary.

3.2.2 Synchronizing with Split-phase Operations

Fibers have names called *handles* that are generated by the runtime layer at creation time. A fiber handle can be stored in user data structures to implement any synchronization mechanism. The runtime layer provides one such synchronization data structure called a *counter*.

A counter has a current value that can be set or incremented by the fibers. A fiber can "wait" for a counter to reach a certain value by registering its handle and the desired value with the counter. The fiber is immediately enabled if the counter's value reaches the desired value. Whenever the counter value is updated, the system checks and enables the fibers that are eligible for execution. The runtime layer does not allow a fiber to wait on multiple counters, so the checking can be implemented inexpensively.

Counters are used to implement split-phase operations. Every split-phase operation takes a counter argument, which is incremented upon the completion of the operation. To wait for a split-phase access to complete, a thread creates a fiber as its continuation and registers the fiber handle with the counter used by the operation. A thread can also wait for the completion of multiple accesses by using an appropriate wait value.

The interface of a split-phase operation provides a "shortcut" synchronization mechanism when the operation is local. Every split-phase operation returns a status code which is OK, WAIT, or FAIL. When a split-phase operation can be processed locally, it simply executes to completion and returns the OK status. If communication is required, it starts the communication and returns the WAIT status. Otherwise, it returns the FAIL status indicating an exception. If the return status is WAIT, the calling thread creates a continuation fiber to perform any remaining computation and synchronization.

3.2.3 Concurrency Control

The fiber abstraction simplifies concurrency control by reducing locking and avoiding certain deadlock scenarios. The atomicity of fiber executions allows many operations to be implemented without explicit locks. A remote operation on a distributed data structure is often a simple read-modify-write operation on one part of the data structure. Such an operation can be implemented in a single fiber with no locking, because the fiber is the only observer and mutator of the part when it executes.

The use of fibers also eliminates, by fiat, deadlock due to spin-waiting, because no fiber is allowed to spin on external conditions. For multithreaded programs comprising multiple data structures, it is generally unsafe to spin-wait. For example, an event graph send operation cannot spin-wait on the availability of buffer space, because it may depend on the execution of another event graph receive operation on the same processor. Programs with spin-waiting also have lower performance portability, because spin-waiting is a scheduling policy whose benefits vary with the machine and the workload. For these reasons, we require that the fibers complete once they start executing.

The fiber executions need only "appear" atomic, meaning that the result of a concurrent execution is equivalent to some sequential execution. Preemption can in fact occur as long as atomicity is preserved at the abstract level. For example, a task stealer operation may preempt an event graph operation, since they update different data sets and the result is equivalent to some sequential ordering of their executions. Preemption can be used to reduce the number of fibers and their associated overheads. Section 3.2.4 discusses fiber preemption in detail.

3.2.4 Scheduling

The best scheduling policy for the fibers depends on the workload. For example, in speculative computations, fibers that are more likely to lead to useful work should be scheduled in preference to other fibers to avoid redundant work. In dynamically load balanced applications, fibers requesting work for idle processors should be given higher scheduling priority than local computation fibers. Instead of providing the programmer with a fixed set of scheduling policies, the runtime layer provides the mechanisms that allow the programmer to write *customized schedulers*. For example, in the CSWEC program, the customized scheduler can examine the state of simulation to determine when deadlock detection should take place.

Figure 3.4 depicts the scheduling hierarchy of the runtime layer. The top-level scheduler is responsible for dispatching the fibers for executions. It selects the next fiber by making queries to the customized schedulers. The programmer specifies the customized scheduler to use for a fiber upon its creation. When a fiber is enabled, the runtime layer simply hands off the fiber handle to its designated scheduler. The customized scheduler enforces its own scheduling policy by giving back the fiber handles to the top-level scheduler in the appropriate order. Multiple customized schedulers can be present at the same time to schedule different types of fibers.

A customized scheduler is an arbitrary data structure with two required operations: *deposit* and *select*. The deposit operation takes a fiber handle and stores it in the scheduler's internal data structure such as a FIFO or a priority queue. The select operation chooses a fiber handle and deletes it from the data structure. The programmer specifies the deposit operation for use when creating a fiber. The programmer also registers the select operation with the runtime layer so that it is invoked periodically by the top-level scheduler.

In addition to the user-defined schedulers, the runtime layer provides three default schedulers: *urgent*, *FIFO*, and *preemptive*. The top-level scheduler always pick fibers from the urgent scheduler if available. The urgent scheduler is used for system threads such as the message handlers and other high-priority application threads such as those that migrate work to idle processors. The FIFO scheduler is a simple first-in-first-out scheduler for use as the default when the fiber does not require sophisticated scheduling. The preemptive scheduler is used for fibers that can preempt the executing fiber.

Preemption is used primarily as a performance optimization. When a preemptive

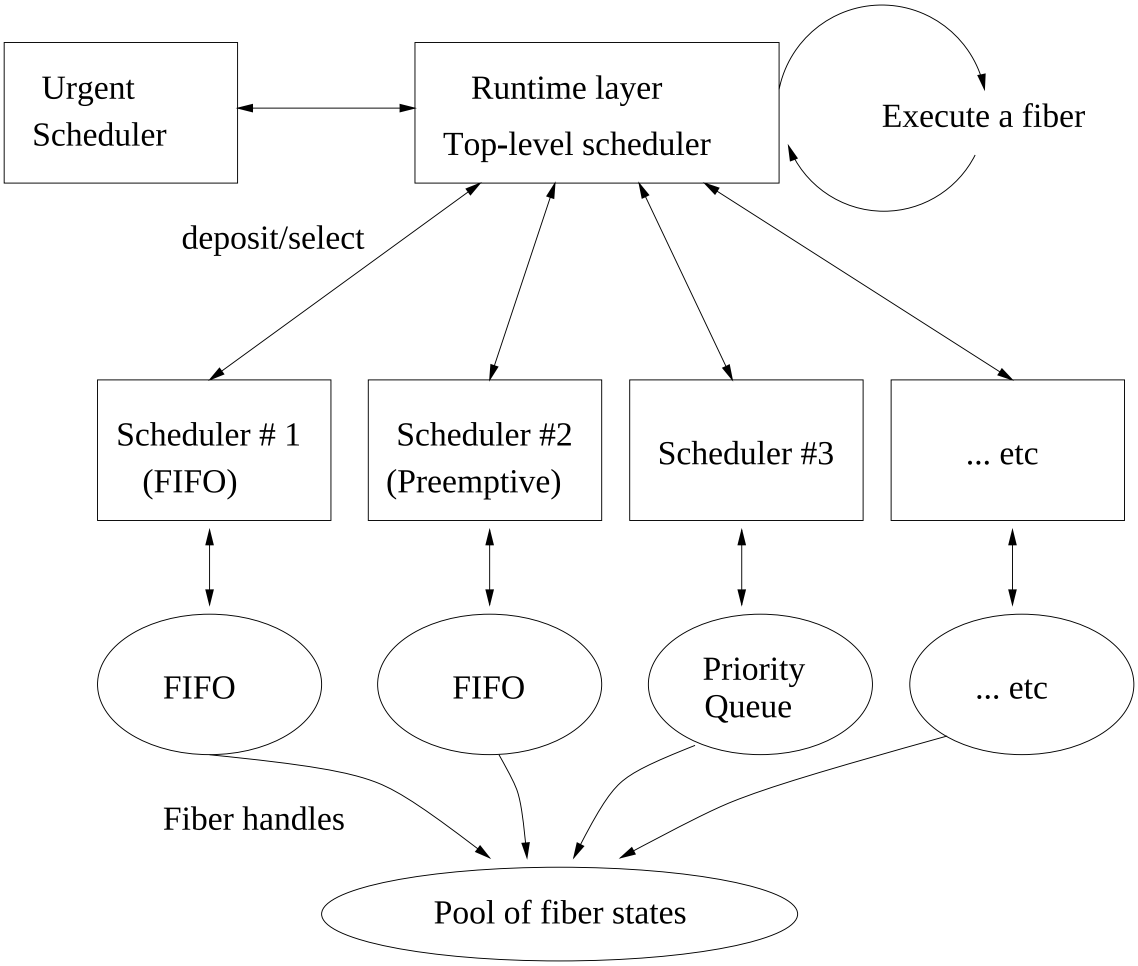


Figure 3.4: Scheduler hierarchy of the runtime layer.

fiber is enabled, it simply starts executing in the enabling fiber's context without requiring state allocation and dispatching. A few restrictions are made to make this optimization possible while preserving the fiber semantics: the preemptive fiber cannot modify its arguments, cannot have continuations, cannot cause starvation of other fibers, and cannot disrupt the atomicity of the executing fiber. The first two restructions are easily enforced in programming the fiber. The third restriction states that number of preemptive fibers cannot grow indefinitely and requires the programmer to check for recursive fiber invocations. The last restriction is more subtle. It requires the programmer to know when a preemptive fiber may execute[2] and carefully code the related fibers to preserve atomicity. Because of the sophistication of use, preemptive fibers are mostly used by the library programmer in highly optimized data structures. Sections 5.1.2 and 5.1.5 demonstrate the use of the preemptive scheduler.

When the urgent scheduler has no fibers, the top-level scheduler picks fibers from the other schedulers in a round-robin manner. If a fiber is scheduled by a fair scheduler, the runtime layer guarantees that it executes eventually unless the program terminates, as long as the number of urgent and preemptive fibers does not grow indefinitely,

Summarizing the above, the Multipol runtime layer decouples scheduling policies from the main application code by localizing scheduling in the implementation of the customized schedulers. The programmer can tune the scheduling policies for performance without restructuring the main application code.

3.3 The Communication Layer

The runtime layer provides a variety of portable communication primitives for expressing irregular communication patterns. We classify the primitives into asynchronous communication primitives, which are used in asynchronous applications such as the CSWEC timing simulator and the Phylogeny program, and bulk communication primitives, which are used in bulk synchronous applications such as the EM3D program.

3.3.1 Asynchronous Communication Primitives

The runtime layer allows the program to invoke an arbitrary fiber on a remote processor. The primitive is similar to creating a local fiber, except that no fiber handle is

[2]To be precise, a preemptive fiber may execute whenever it is enabled or when the network is serviced.

returned and the fiber is enabled immediately upon its arrival at the processor. The fiber invocation does not require any acknowledgement from the remote processor. It returns immediately after the fiber state is copied into an internal message buffer so that the caller can proceed with other computation. The runtime layer guarantees that the fiber invocation takes place eventually unless the program terminates.

A split-phase operation can be implemented by a sequence of local and remote fiber invocations as follows. First, the interface function of the operation takes the user arguments and computes the processor partition with the requested data. If the access can be satisfied locally, the function synchronizes with the caller as describe in Section 3.2.2. Otherwise, it creates a continuation fiber and forwards the request and the continuation to the remote processor by invoking a fiber. The remote fiber accesses the data and replies with the result by invoking another fiber on the requesting processor, which synchronizes with the continuation fiber to resume the remainder of the operation.

Remote fiber invocation is similar to the notion of an active message, in that both allow the programmer to start computation on a remote processor. Other than that basic similarity, the two mechanisms are quite different. Active messages are designed as an efficient network layer, not as a computational abstraction [vECGS92], and are insufficient for implementing remote operations in the follow respects. First, active messages take a fixed number of arguments (typically 4 words) for efficiency, and leave the problems of buffer allocation and fragmentation to the upper-level software. A later version of active messages called generic active messages (GAM) [CKK+94] allows the programmer to invoke the message handler with an arbitrary number of arguments, but the upper-level software is still required to allocate the storage prior to the communication. Second, the handler code must follow a request/reply protocol to avoid network level deadlock [vECGS92]. For example, a handler invoked by a request message cannot send another request message. The protocol limits the types of accesses that can execute in a handler. Finally, the scheduling of the handler code is coupled with the delivery of the message, which makes it difficult to implement other scheduling policies. Because active message handlers may execute whenever the network is serviced, it is also difficult to enforce concurrency control. For example, local computation may be preempted by an arbitrary active message handler when it polls the network. Buffering of messages is often required to enforce atomicity without causing network congestion, due to disabled interrupts or insufficient polling.

The runtime layer separates the use of remote fibers and network handlers such as active messages. Remote fibers are used to perform computation, while network handlers are used to efficiently transfer data between processors and post fibers. Interrupts or polling need be disabled only in a small set of system routines such as memory allocation and fiber manipulation. The fibers can send or receive arbitrary messages without compromising atomicity or causing network deadlock.

3.3.2 Bulk Communication Primitives

Bulk communication primitives such as bulk read, bulk write, and bulk store transfer blocks of memory between the processors. They provide a global address space on a distributed memory machine, where a memory location is uniquely identified by its local address and its processor identifier.

The read and write accesses are split-phase, and they increment a counter when the transfer is completed. The store access returns immediately without waiting for the transfer to take place, and it increments a counter when the source memory can be safely reused. Upon completion of the transfer, a fiber is invoked on the receiving processor with the destination address. The fiber may reply to the sender with an acknowledgement message, although a more typical scenario is to acknowledge multiple store operations at the end of a phase, such as in the EM3D program. The bulk accesses in our runtime layer are modeled after the Split-C [CDG+93] bulk accesses and the GAM [CKK+94] store operation, except that our primitives have a split-phase interface that is integrated with the Multipol thread layer.

3.3.3 Implementation and Optimization

The communication primitives are translated into the machine primitives by the communication layer, which handles buffer allocation and flow control. The communication layer also attempts to optimize the handling of the messages based on their sizes and scheduling semantics.

All messages generated by the communication primitives are put into four categories depending on their sizes and the underlying machine characteristics. They are handled differently by the communication layer as shown in Figure 3.5.

Large messages are those that transfer more data than the *aggregation threshold*, the desired physical message size for achieving good communication bandwidth. Because

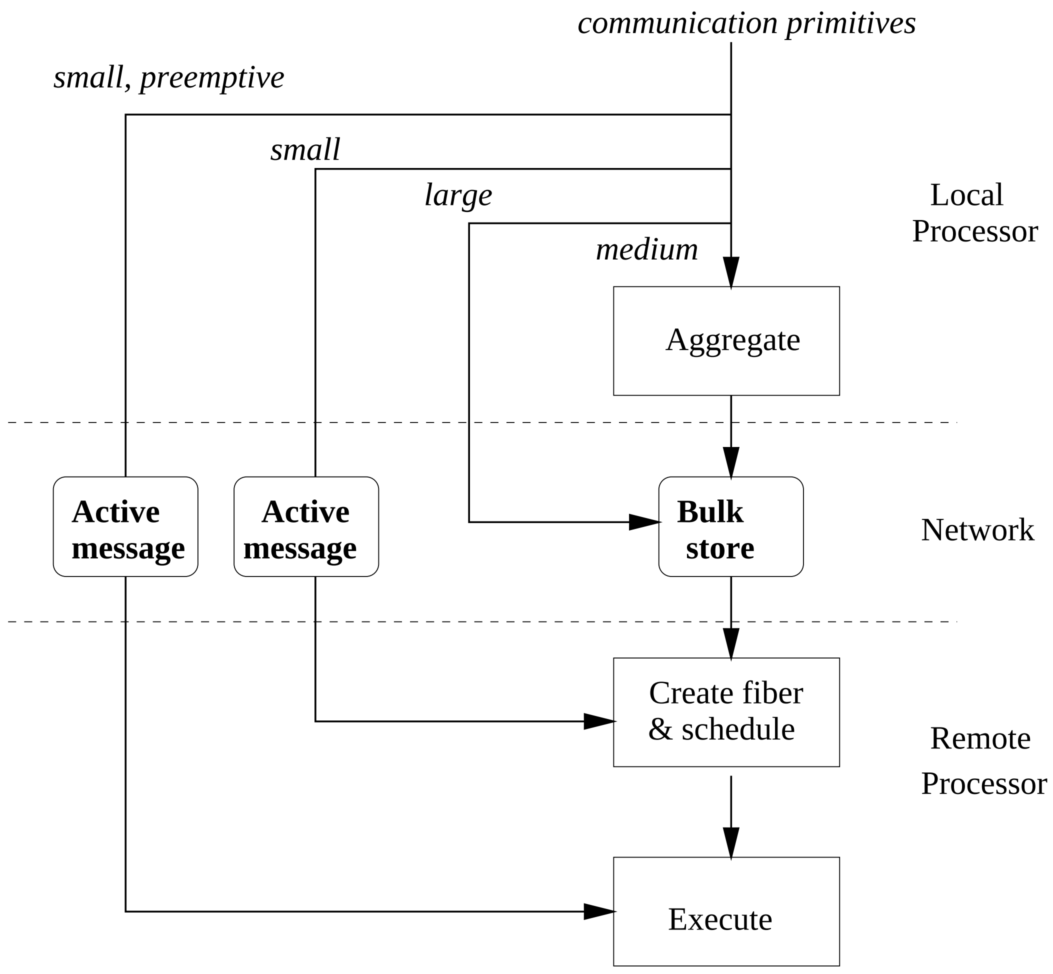


Figure 3.5: The structure of the message layer.

large messages have good overhead-to-payload ratios, they are sent immediate without further aggregation. For example, the EM3D program contains mostly large messages and it would achieve the same performance with or without message aggregation.

Medium messages are those smaller than the aggregation threshold but larger than an active message (e.g., 4 words). Such messages have poor overhead-to-payload ratio if they are sent separately. Therefore, the communication layer accumulates medium messages in a internal buffer and sends the aggregated message in bulk when the size of the buffer exceeds the aggregation threshold. The runtime layer guarantees the messages are delivered eventually by flushing the buffer to the network when the processor runs out of fibers to execute, or when a certain number of fibers have executed since the last communication event. The programmer may also flush the message buffer explicitly.

Message aggregation uses excess parallelism to reduce the communication start-up overheads. Figure 3.6 shows the effect of message aggregation on the performance of the CSWEC program running on the Sparc cluster and the CM5. The results shows that message aggregation can have a significant impact on machines with large communication start-up overhead, such as the Sparc cluster, but it has a much smaller effect on machines with low communication start-up overhead like the CM5. Aggregating messages may increase the latency of synchronization events and even increase total running time. This tradeoff between overhead and latency will be discussed further in Chapter 5.

A *small* message fits in an active message. On machines that support efficient active messages, small messages are sent directly as active messages to save the buffer allocation and flow-control overhead. For remote fiber invocations, the active message handler simply creates the fiber and deposits the arguments into the fiber's state. If the fiber is scheduled by the preemptive scheduler (called a *small preemptive* message), the active message handler may directly execute the fiber code to avoid the costs of creating a new fiber. This optimization is similar to the idea of optimistic active messages [WHJ$^+$95].

3.4 System Data Structures

The runtime layer provides two system data structures: the distributed object manager for naming distributed objects and the snapshot data structure for taking snapshots of distributed objects.

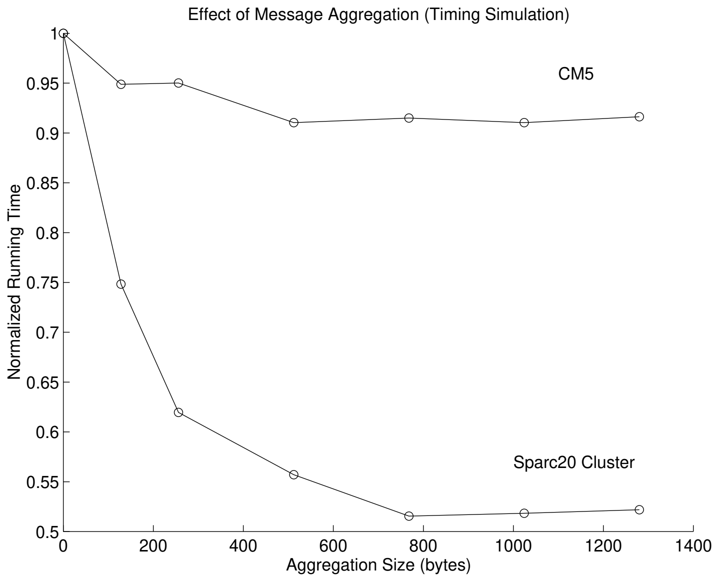


Figure 3.6: Effect of message aggregation on the Sparc cluster and the CM5. The graph shows the reduction of running time (percentage) due to message aggregation.

3.4.1 Distributed Object Manager

The distributed object manager provides primitives for naming, creating, and accessing data structures that are distributed over multiple processors.

An instance of a distributed data structure is called an *object*. The storage for an object is partitioned over all the processors in the system. The partitions are logically strung together by the object's unique name, which is allocated by the distributed data manager.[3] The data structure also provides primitives for mapping an object name to its partition address on the local processor.

The data structure also provides global operations on the partitions of an object, such as split-phase barrier and reduction. The global operations are used to synchronize the threads that access different parts of the the same object. Multiple global operations on different objects can take place at the same time, which makes it easy to compose data structures. For example, in the Phylogeny program, a barrier operation can be performed on the task stealer object to detect termination when another barrier operation for combining the failure stores is taking place.

[3]The distributed object manager reserves a collection of names for direct allocation by the application programmer.

3.4.2 The Snapshot Data Structure

The snapshot data structure provides a uniform interface for taking "consistent" snapshots on the states of distributed objects. The snapshot of a distributed object is consistent if it corresponds to some state of the object when no mutating accesses are in progress: all mutators have either completed or have been suspended before causing any side-effect.

Because the processors are independently scheduled, taking a consistent snapshot requires coordinating the accesses issued by different processors. The snapshot data structure provides a general coordination mechanism that creates a window of time when the object state is guaranteed to be consistent. The mechanism can be applied to any type of object. It can also be used for the snapshot of the collective state of multiple objects.

The definition of a mutator access is dependent on the data structure and the property of interest. For example, in detecting the termination of a task stealer object, the only mutator access is the task migration operation. The task add operation, task remove operation, and other local computation can proceed concurrently with the snapshot operation without disrupting termination detection as described in Section 2.2.3.

The interface of the snapshot data structure allows the programmer to define mutator accesses as follows. First, one snapshot object is associated with each property to be detected, such as the termination of a task stealer object or the completion of all insert accesses of a hash table object. The name of the snapshot object is then used by its corresponding mutator accesses to register their state of execution. Those accesses that do not use the snapshot object are classified as non-mutators, and they can executely concurrently with an on-going snapshot operation. For example, to detect termination, the task migration operations must register their status with their associated snapshot object, while the task add and remove operations can proceed as usual. The same snapshot object can be used by different objects to glue together their states. For example, two hash tables can share the same snapshot object, so that their sync operations return when the insert accesses on both hash tables have completed.

To guarantee the progress of a snapshot operation, each mutator access must obey the following protocol. Before an access causes any side-effect on the property of interest, it must call the *query* operation on the snapshot object to determine if it can proceed with the mutation. If the return value is OK, the access calls the *execute* operation on the snapshot object and resumes execution. The *execute* operation informs the snapshot object that a mutator access is in progress. Once the access starts to mutate the object, it must guarantee to complete in finite time. When an access completes, it calls the *complete* operation on the snapshot object to declare its completion. The execute and complete operations may be called by different processors if the access involves remote computation.

If the query operation returns WAIT, a snapshot operation is already in progress and the access must suspend itself by creating a continuation fiber. The access then calls the *suspend* operation on the snapshot object with the fiber handle. The snapshot object automatically resumes the suspended access when the snapshot operation completes by enabling the continuation fiber. When an access is resumed in this manner, it must repeat the protocol by issuing the query operation again.

When the client application needs to perform a snapshot, it does so in three stages. First, the client invokes the *freeze* operation on the snapshot object to temporarily "freeze" the objects in a consistent state. The freeze operation has a split-phase interface, which completes when all mutators that have called the execute operation on the snapshot object have also called the complete operation. The client then reads the states of the objects to obtain the snapshot. The same snapshot is obtained regardless of the order in which the partitions are read, because no mutator access can modify the object states when a snapshot operation is in progress. Finally, the program invokes the *unfreeze* operation on the snapshot object to allow resumption of all suspended mutators. Figure 3.7 illustrates the three stages of a snapshot operation. The snapshot data structure handles all but the second stage, which requires client-specific code to interpret the object state.

The implementation of the freeze operation uses the global operations provided by the distributed object manager. The operation first performs a global reduction on the snapshot object to compute the number of mutator accesses that have started executing. Each processor then uses a separate counter to wait for the completion of all its mutator accesses. Because the freeze operation completes as soon as the calling processor completes its share of mutator accesses, the programmer may need to perform a barrier operation after the freeze operation to ensure all processors have left the freeze stage.

The snapshot operation can also be used to perform group acknowledgement in bulk synchronous programs such as the Tripuzzle program. The snapshot data structure provides a *sync* operation, which waits for the completion of all previously issued accesses. The sync operation is simply a sequence of the three operations: freeze, barrier, and un-

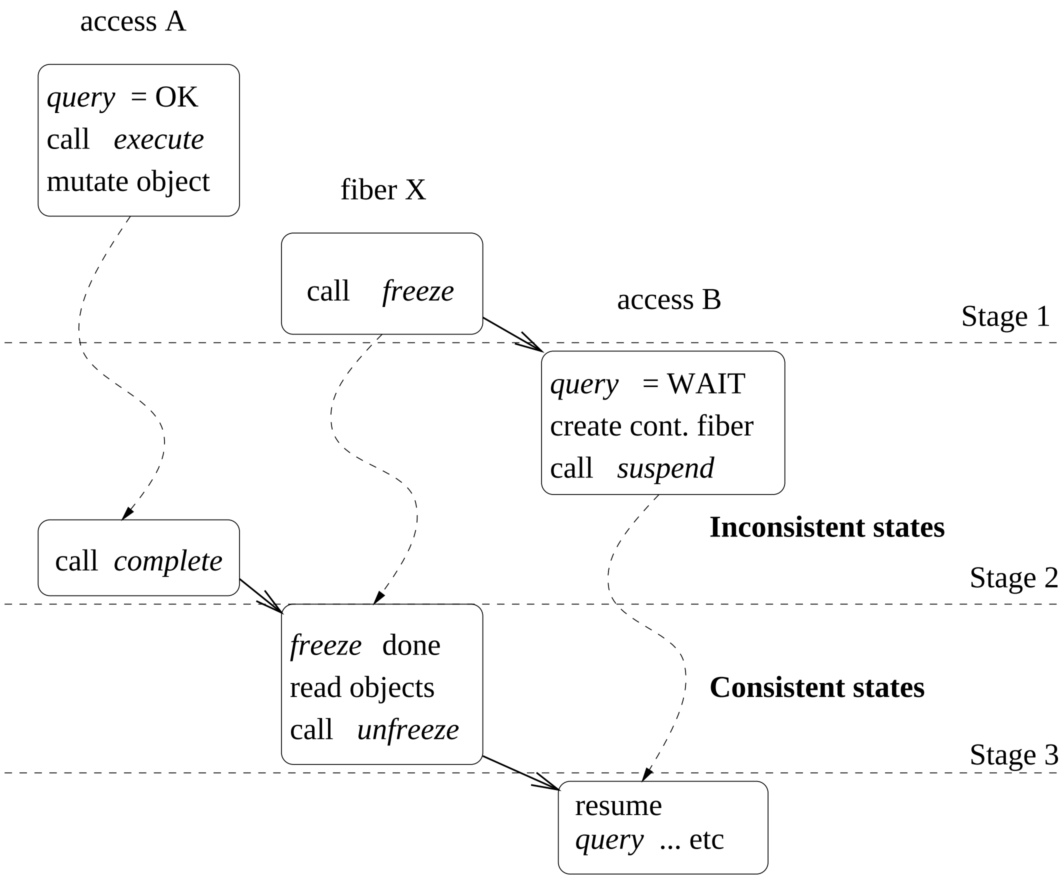


Figure 3.7: The snapshot operation. The dashed arrows string together the continuations of the same thread. The solid arrows denote the dependencies between threads. The snapshot obtained is the object state after access A completes but before access B takes place.

freeze.

Many distributed snapshot algorithms have been proposed, notably the algorithm by Chandy and Lamport [CL85] for distributed FIFO channels. The Multipol snapshot data structure is different from these algorithms in that it has stronger semantics. The data structure does not simply return an arbitrary snapshot; it returns a unique snapshot which corresponds to the state of the distributed object when the freeze operation is invoked plus the updates performed by the mutators that have started execution before the freeze operation. The data structure is also more versatile in that it allows the programmer to take a snapshot of the high-level state of the object by defining the appropriate mutator accesses. However, the stronger semantics prevents the snapshot algorithm from being incremental – all new mutator accesses are suspended when a snapshot operation is taking place. The data structure also imposes more restrictions on its use to ensure progress.

3.5 Implementation of the Communication Layer

The runtime layer is highly portable because it is built on a common machine interface. Porting the runtime layer to a new machine involves implementing a *main* procedure for starting up the processor programs, defining a few runtime constants, and implementing a small set of bulk communication primitives such as barrier, read, write, and store. Most of the communication primitives are themselves built on the bulk store primitive.

The machine interface also defines some "short-cut" primitives for implementing the optimizations described in Section 3.3.3. The communication layer calls these primitive to determine if a message can be sent efficiently as a small or preemptive message. The short-cut primitives encapsulate knowledge of the underlying machine, so the communication layer is free of any machine-dependent code.

The runtime layer has been ported on three types of communication libraries: active messages (CMAML), cooperative message passing (MPLp, NX), and BSD sockets using the TCP/IP protocol stack. We reuse code from GAM [LC95, CKK+94] and libsplit-c [Lun94], the communication library for the Split-C language [CDG+93]. Most modifications to their code are for integrating the the communication primitives with the Multipol thread layer. In the paragraphs that follow, we sketch the implementation of these ports and describe the source of their overheads. We only present the store primitive, because read and write are implemented in terms of store.

3.5.1 Active Messages

We implemented the store primitive on top of the CMAML active messages. Performing a store operation requires the following steps. First, the sending processor sets up the local state for the transfer and sends an active message to open a "segment" on the receiver processor. The segment is used to check the completion of the transfer because the network may reorder active messages. The store operation then returns immediately so that the program can proceed with other computation. The segment implementation is modeled after the implementation of active messages by Thorsten von Eicken [vECGS92].

Upon receiving a send request, the receiver sets up a segment and replies the sender with the address of the segment using another active message. When the sender receives the reply, it fragments the message into 4-word packets and injects them into the network. After the network has received the entire message, the sender synchronizes with the caller so that the send buffer can be reused. When the receiving processor receives the entire message, it creates the specified fiber and frees the segment.

The communication start-up overhead of the store operation is from the active messages used to set up the segments.

3.5.2 Cooperative Message Passing

In the cooperative message passing model, both the sender and the receiver are required to initiate the communication, because the receiver determines the destination address of the message. The model creates difficulties for implementing one-way communication such as the store operation.

We reuse most of the libsplit-c code for NX and MPLp. The implementation is similar to the CM5 implementation, except that a receive message must be explicitly posted when setting up a segment. Because of the multithreaded execution model, a Multipol programs may have multiple store operations outstanding between the same pair of processors. To match the send operation with the appropriate receive operation, we use the *message tags* provided by the native communication libraries. A receive operation always removes the message posted by a send operation with the matching tag.

The communication start-up overheads includes the active messages for setting up the segment and the inherent start-up overhead in the underlying libraries.

3.5.3 Socket

The socket implementation is based on the GAM implementation by Lok Tin Liu, where communication takes place via TCP/IP streams. The port can be used for almost any UNIX workstation. However, sockets are rather heavy-weight, and sending a message incurs at least 800 microseconds of overhead on the Sparc cluster.

3.6 Related Work

The fiber abstraction in the Multipol runtime layer is similar to the thread abstraction in TAM [CSS+91], Cilk [BJK+95], and Nexus [FKOT91], and the chare abstraction in Charm++ [SK91, KK93]. TAM is a threaded abstraction machine for data-flow computation. TAM threads are nonblocking units of computation for latency hiding which are generated by a compiler (each thread corresponds to a basic block in the data-flow graph of a function). TAM threads also have fixed scheduling and synchronization mechanisms that are tailored to data-flow computation. In comparison, Multipol fibers are higher-level abstractions used by the programmer. They have more flexibility in scheduling and use.

Cilk threads are also nonblocking and are generated explicitly by the programmer, but they are mainly used for load balancing instead of latency hiding. The Cilk runtime layer uses a fixed load balancing and scheduling policy based on task stealing, which is similar to the works on lazy task creation [MKH91] and stacklet [GSC95]. In contrast, Multipol fibers are used for latency hiding and do not migrate across processors. Furthermore, the scheduling policies for Multipol fibers can be customized for a particular application. For example, the CSWEC program uses a customized scheduler that inspects the state of simulation to determine if a deadlock detection thread should be scheduled.

Nexus threads are POSIX threads that can block on synchronization events or voluntarily yield control of the processor. An arbitrary Nexus thread can be invoked on a remote processor (called remote service request), which is similar to remote fiber invocation. Nexus targets heterogeneous computing environments and has additional overhead in the implementation, such as the packing and unpacking operations for placing data in message buffers. Nexus also has little provision for customized scheduling policies.

The chare abstraction in Charm has entry functions that can be invoked by other chares on a remote processor. Like Multipol fibers, the entry functions must contain non-blocking code. Like Cilk threads, Charm chares are also units of load balance and they may migrate across processors. The Charm system provides little support for using customized scheduling policies for the entry function invocations.

There exist many portable communication layers such as MPI (Message Passing Interface [For94] and active messages [vECGS92]. MPI provides cooperative message passing primitives for point-to-point and group communication. However, it does not provide irregular communication primitives such as remote fiber invocation. Active messages are similar to remote fiber invocation, but impose more restrictions (see section 3.3.1 for detailed comparison). Neither MPI nor active messages have an integrated thread layer.

Optimistic active messages [WHJ+95] are similar to the small preemptive messages in Multipol in that they both reduce the thread creation overhead. An optimistic active message executes as an active message until it blocks on a synchronization event, in which case a continuation thread is created to execute the remainder of the message handler. Although optimistic active messages permit the message handlers to execute arbitrary code, they do not permit the programmer to use application-specific scheduling policies.

3.7 Summary

The Multipol runtime layer provides infrastructure for building irregular applications on distributed memory platforms. It addresses the following issues: programmability, portability, and performance.

For programmability, the runtime layer provides the fiber abstraction for implementing distributed data structures. The atomicity of fibers facilitates concurrency control, scheduling, and performance profiling. The programmer can also supply customized schedulers to enforce application-specific scheduling policies for the fibers. In addition to the basic computational abstractions and communication primitives, the runtime layer also provides infrastructure for naming, accessing, and taking snapshots of distributed objects.

For portability, the runtime layer provides a uniform programming interface on distributed memory platforms. Applications built from the runtime layer can run without change on a variety of machines including the CM5, SP1, Paragon, and workstation clusters. The implementation of the runtime layer is also highly portable because it is built on a common machine interface.

For performance, the runtime layer uses the excess parallelism in the application to hide latency and reduce overhead. It provides a multithreaded execution model in which the latency of remote operations can be overlapped with useful computation. It also takes advantage of the scheduling semantics of the communication events and the machine characteristics to optimize communication performance. For platforms with high communication start-up overhead, the runtime layer performs automatic message aggregation to improve communication bandwidth.

Chapter 4

Mprof: a Performance Profiling Toolkit for Multipol Programs

After the pain-staking process of debugging and testing a parallel program, the programmer's task is only half-done. To justify the additional investment in processors and networks, the parallel program must also run efficiently. Therefore, a significant amount of time spent in developing parallel programs is in performance tuning. At this stage of program development, the programmer performs experiments, analyzes the outcomes, and modifies the program to improve performance. The tuning process is especially burdensome for irregular parallel programs, because they often have unpredictable communication schedules and synchronization patterns that are difficult to characterize. To increase productivity, it is important to provide sufficient profiling information to help the programmer detect and analyze performance inefficiencies.

In this chapter, we describe a toolkit called *Mprof* for profiling the performance of Multipol applications. Our toolkit identifies the two major sources of performance inefficiency: overhead and insufficient parallelism. Overhead is measured as busy processor cycles that are unnecessary or redundant. Insufficient parallelism manifests itself in the amount of idle time caused by dependencies between the program components. Mprof provides information on the cost of data structure accesses to detect overheads. It also identifies long latency synchronization events to detect the lack of parallelism.

Mprof uses profiling data from benchmark executions in modeling the performance of data structures. Specifically, we estimate the performance of a data structure using its runtime access pattern and the unit cost of each access. The access pattern is characterized by a small set of *high-level statistics* collected during the execution. The cost of each access is obtained by statistical modeling based on a *parameterized cost model* specified by the library programmer. Although the values of the statistics vary with the executions, the cost model remains the same for a given machine. By combining runtime statistics and reusable cost models, we obtain accurate profiling information with low runtime overhead.

To help tuning applications that use abstract data structures from a library, we provide a *performance interface* for the library programmer to customize the profiling information for each data structure. In the performance interface, the library programmer specifies the statistics to collect, their parameterized cost models, and how the costs are to be summarized for presentation. The information is used by the toolkit to automatically instantiate the cost models and to produce the profiling information after each execution.

The rest of the chapter is organized as follows. Section 4.1 describes the difficult issues in profiling Multipol applications. Section 4.2 gives an overview of the Multipol profiling toolkit. Section 4.3 introduces an example parallel program called *PIPE*, which is used to illustrate our methods throughout the remainder of the chapter. Sections 4.4 and 4.5 describe the techniques we use in profiling overhead and insufficient parallelism, respectively. Section 4.6 gives an overview of related work in performance profiling, and Section 4.7 summarizes the results in the chapter.

4.1 Issues in Performance Profiling

Multipol applications raise new issues in performance profiling. In this section, we discuss these issues and describe how they are addressed by the techniques used in the Multipol profiling toolkit.

4.1.1 The Problems

We categorize the difficult issues into those arising from the parallel execution model, the irregularity of applications, and the library abstraction.

First, parallel executions require more extensive profiling information than sequential executions, such as the costs of the individual operations and the synchronization delays due to their dependencies. The multithreaded execution model in Multipol creates problems for obtaining such information, since in Multipol, a long latency operation is decomposed into a sequence of fibers. The fibers are scheduled individually and therefore must be profiled separately, which increases the profiling overheads. The decomposition of operations into fibers also obscures the profiling information on the operations themselves.

Second, irregular parallel programs often use concurrent data structures with unpredictable access patterns. The cost of each type of accesses needs to be determined in order to characterize the cost of the data structure. Unlike timing standalone procedures, it is difficult to accurately time concurrent data structure accesses.

Finally, the library imposes an abstraction barrier between the application and the data structures. Most available profiling tools are faced with the dilemma of providing too much information and thus breaking the abstraction barrier, or providing insufficient information and thus preventing performance tuning.

4.1.2 Our Approach

The three features of our approach are:

- Profiling both the costs and the dependencies of data structure accesses.
- Measuring high-level statistics instead of absolute running time.
- Preserving the library abstraction with a customizable performance interface.

First, we provide mechanisms for profiling the time spent in each data structures and the amount of idle time caused by dependencies between data structures. The combination of both types of profiling information provides more insights into the performance inefficiencies in the execution and the optimization techniques for resolving them.

Second, we characterize parallel executions using a set of high-level statistics instead of the absolute running time. The statistics are combined with their cost models to produce the final profiling output after each execution. High-level statistics are easier to insert than direct timer calls, which have to be carefully placed to ensure well-formedness and accuracy in profiling multithreaded applications. They also incur lower runtime overhead, because one statistic can be used to represent the costs of multiple fibers or functions. Although the statistics must be combined with their cost models to produce useful profiling information, the cost models can be built off-line using benchmark executions and reused by many applications.

Finally, instead of providing a fixed set of profiling information, we provide a performance interface for the library programmer to customize the profiling information for each data structure. The profiling information can be made independently of the implementation of the data structures, so the library can be developed and optimized separately from the applications. The use of a performance interface allows the library programmer to provide useful profiling information while preserving the library abstraction.

Many of the key features in the profiling toolkit are enabled by the design of the Multipol runtime layer, especially the non-blocking semantics of Multipol fibers. The non-blocking semantics exposes all idle processor cycles to the runtime layer, so that it can accurately profile the amount of idle time and its distribution among the synchronization events. Isolating idle cycles is also important for the reusability of the cost models, because the amount of idle time spent in an operation may be workload dependent, while its other costs are predictable.

4.2 Overview of Mprof

Mprof is used to manage the following activities: program instrumentation, benchmarking, and modeling. The Multipol library lies in the center of all activities. In addition to the data structure source code, the library contains the following information: a performance interface for each data structure consisting of its parameterized cost models and statistics to measure, a database of performance profiles from benchmark executions, and a set of instantiated cost models. Figure 4.1 depicts the organization of the library.

The flow of the activities is illustrated in Figure 4.2. First, the library programmer specifies the performance interface for each data structure using a simple language provided by the toolkit. The interface is transformed into a set of C functions for manipulating the statistics. The library programmer then instruments the source code using these functions. Mprof also uses these functions to instantiate the parameterized cost models and produce the profiling output.

Next, the library programmer designs benchmark programs to exercise the data structures. A program can be used as a benchmark if all its program components have a performance interface. The benchmark programs are executed, and their measurements are recorded in the profile database for use in instantiating the cost models. The measurements include the values of the statistics for each data structure and the total execution time of

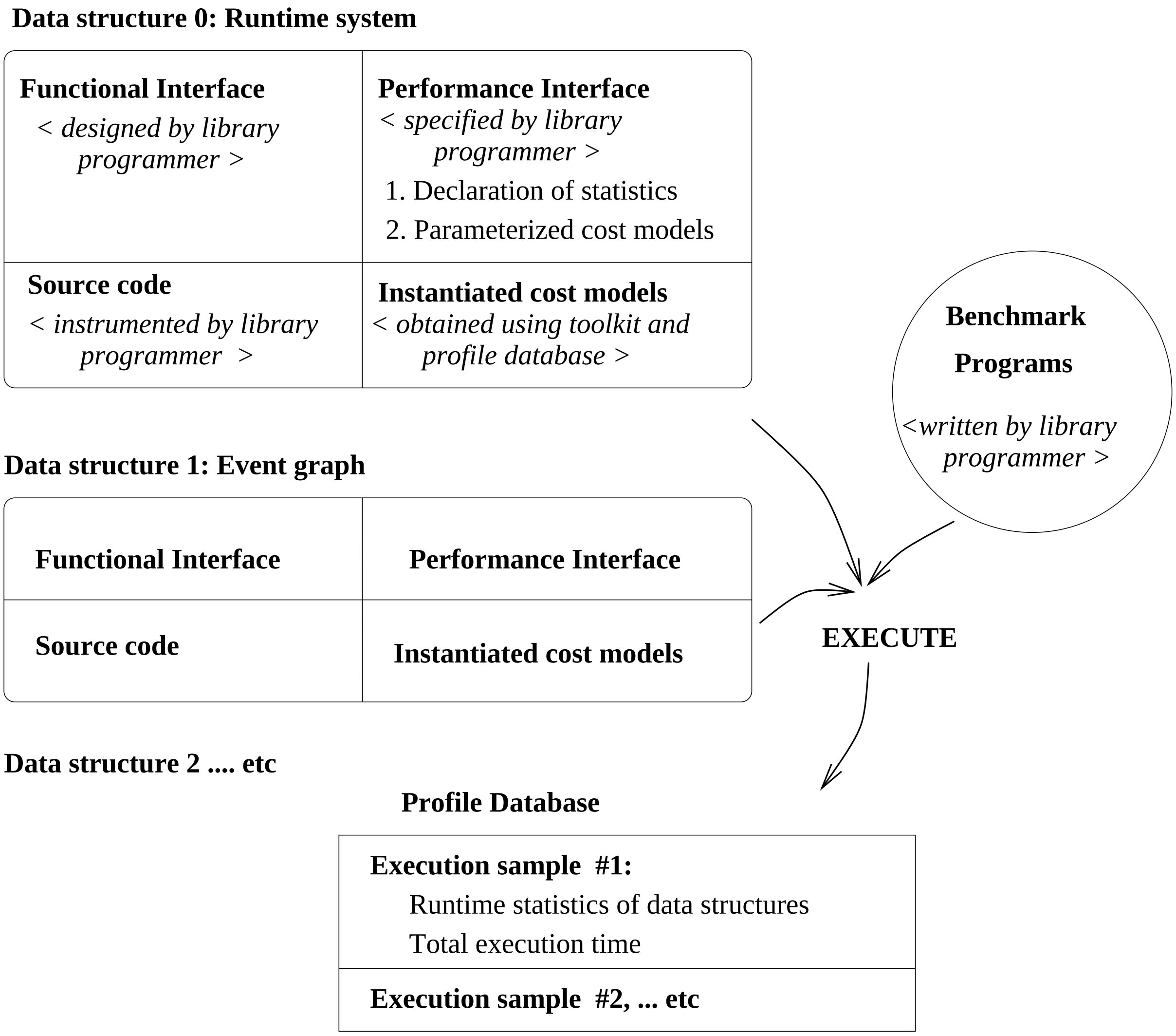


Figure 4.1: Organization of the Multipol library.

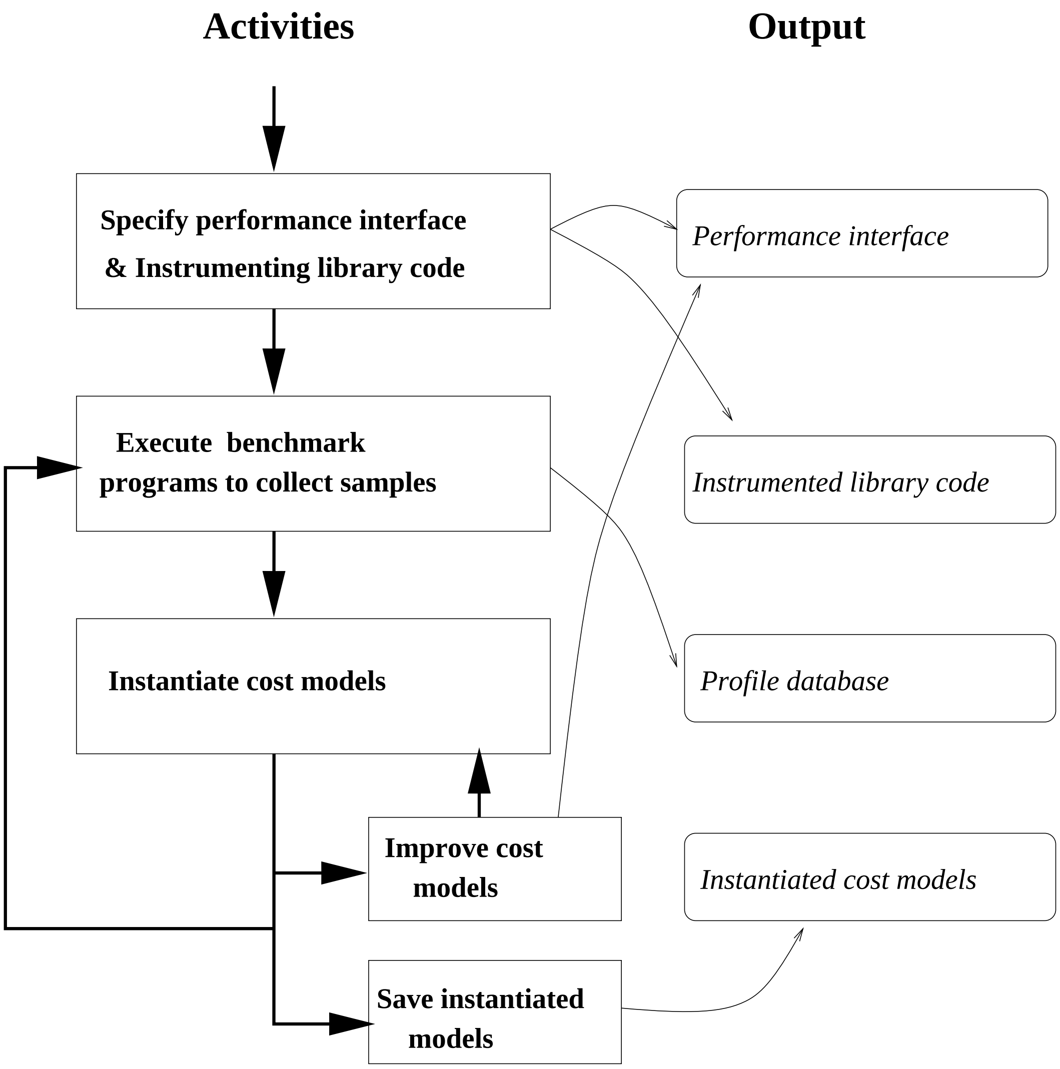


Figure 4.2: Performance profiling activities.

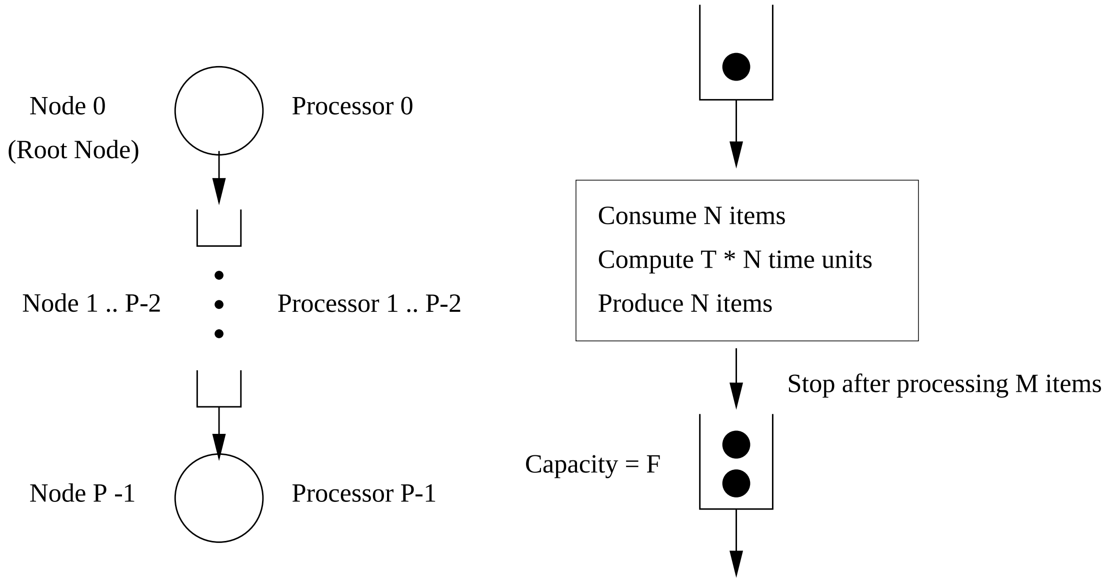


Figure 4.3: Structure of the PIPE program.

the program.

After sufficient samples are obtained, the library programmer starts the modeling step of the toolkit to automatically instantiate the cost models. The cost models in the performance interface are parameterized, and the parameters are set by the modeling tool to match the samples in the profile database. The modeling tool provides feedback on about the quality of the calibrated cost models. If the library programmer is sufficiently confident of the models, they are saved for use in actual profiling. Otherwise, the library programmer improves the cost models and repeat the modeling session or obtain more samples by executing more benchmarks. The toolkit provides a simple scripting language for managing the modeling activities.

4.3 An Example Parallel Program – PIPE

We use a small parallel program called PIPE to illustrate our performance profiling methods. The PIPE program uses the event graph data structure to exploit pipeline parallelism. It resembles the kernel of the CSWEC program described in Chapter 2. In this example, the graph forms a simple path of P nodes, each of which is assigned to a unique processor as shown in Figure 4.3. All nodes except the root node repeat the following three

```
consume:

    repeat
        while (fewer than N items removed by current iteration) {
            [ wait for new items ]
            remove items (up to N items) from graph
        }
        perform (N * T) units of computation
        issue produce
        [ wait for produce to complete ]
    until (M items processed)

produce:

    repeat
        [ enqueue a new item to graph ]
    until N items enqueued
```

Figure 4.4: Pseudo code of the PIPE program. The long latency operations are enclosed in square brackets.

steps: remove N incoming items, compute for $T \times N$ time units, and then enqueue N items. The root node simply fills the pipeline by enqueueing N items after every $T \times N$ units of computation. The capacity of the event graph is set to F. The program stops after M items exit the pipeline.

Figure 4.4 shows the pseudo code executed for each graph node. A *consume* thread is created for each node. The thread awaits and removes new items from the incoming edge, performs the computation, and then calls the split-phase operation *produce* to enqueue new items. The consume thread blocks until the produce operation has successfully enqueued all items.

The value N determines the burstiness of the pipeline traffic, which has to do with the best setting of F. There is also a tradeoff between the latency and the overhead of the communication layer. A high degree of message aggregation reduces overhead, but may increase the latency of propagating items and thus reduce parallelism.

4.4 Measuring the Costs of Data Structures

In this section, we describe the techniques we use in measuring the costs of the data structures. We discuss how high-level statistics can be used to characterize costs, how to obtain their cost models, and how to summarize the profiling output in an implementation independent manner to preserve the library abstraction.

The techniques described in this chapter are not limited to the Multipol data structures; they can be used in profiling any code module constructed from the Multipol fibers, such as the Multipol runtime layer. For convenience of discussion, we use the term data structure to refer to a generic code module.

4.4.1 Characterizing Costs with High-level Statistics.

The behavior of irregular parallel programs can often be characterized by a small set of high-level statistics. For example, the total computation per node in the PIPE program can be characterized by the statistics T, N, and M described in Section 4.3; the time spent in the runtime layer can be characterized by statistics on the thread and the communication layers; the time spent in the event graph data structure can be characterized by the size of the data item, the connectivity structure of the graph, and the pattern of accesses such as the number of operations. Such high-level statistics can be collected inexpensively because they occur relatively infrequently in the execution. They also provide more insights for tuning, because they summarize the costs of high-level objects or operations. The programmer determines the set of statistics to collect, which is a tradeoff between accuracy and overhead – the more complete the statistics are, the more accurately they characterize the behavior of the program and the higher the cost of instrumentation and measurement.

Mprof provides a simple language for specifying the statistics to collect for a data structure. Figure 4.5 shows the part of the performance interface that declares the statistics for the PIPE program.[1] Each statistic has a symbolic name and a one-line comment on its meaning. The interface is transformed by Mprof into C functions, which the programmer manually inserts in the source code.[2] There are functions to create, set, and increment the values of the statistics.

The high-level statistics are not detailed enough for making performance tradeoffs.

[1] In reality, the performance interface for each data structure is stored in a different file.
[2] The transformation is done with Tcl scripts [Ous94].

```
# Declare statistics for "pipe"
 P                            -- Partitions per processor (= 1)
 T                            -- Computation
 N                            -- Size of a enqueue or dequeue batch
 M                            -- Total number of items produced

# Declare statistics for "rts"
 P                            -- Partitions per processor (= 1)
 Sched                        -- Number of customized schedulers
 ThreadAlloc                  -- Number of threads allocated
 ThreadEnable                 -- Number of threads enabled
 ThreadDisp                   -- Number of threads dispatched
 ThreadContext                -- Total size of thread contexts
 IdleCycle                    -- Number of idle scheduling cycles
 MsgSent                      -- Number of data messages.
 FlowMsgSent                  -- Number of flow control messages.
 ByteSent                     -- Total number of bytes in the messages.
 FastMsgSent                  -- Number of fast messages.
 RemoteOp                     -- Number of remote communication events.
 ByteCopy                     -- Number of bytes in local communication.

# Declare statistics for "graph"
 P                            -- Partitions per processor (= 1)
 EltSize                      -- Size of each element
 EdgeCap                      -- Capacity
 Node                         -- Number of nodes on this processor
 Edge                         -- Number of edges on this processor
 NOp                          -- Number of accesses and queries
 NBlock                       -- Number of suspended accesses
 NMsg                         -- Number of internal messages
 EnqSize                      -- Bytes of data enqueued
 DeqSize                      -- Bytes of data dequeued
```

Figure 4.5: High-level statistics for the PIPE program. The figure shows parts of the performance interfaces of the program's computational kernel (pipe), the runtime layer (rts), and the event graph data structure (graph).

For example, two messages of 4 bytes are not necessarily cheaper than 1 message of 1000 bytes. Therefore, these statistics must be converted into the absolute running time of their corresponding operations. The conversion is performed using the parameterized cost models supplied by the programmer, which are to be instantiated against measurements from benchmark executions. Although the statistics must be dynamically collected, their cost models remain largely static on the same platform, due to the non-blocking semantics of Multipol fibers. For example, the number of messages generated by an application varies with the workload, but the cost of sending a fixed size message (excluding the network latency) is roughly the same on the same machine.[3]

The library programmer's task is to find the parameterized cost models for each data structure on a given machine. For example, the cost model of sending a n-byte message may be defined as $c_1 + c_2 \times n$, where c_1 and c_2 are the parameters to be instantiated. Unlike standalone procedures, for data structure access functions, it is usually impossible to obtain the cost model by devising a benchmark that exercises only a particular type of access. For example, the PIPE program, although very simple in construction, consists of three interacting modules: the Multipol runtime layer, the event graph data structure, and the computational kernel of the program. Therefore, we use an equation solving method to instantiate the parameters in the cost models based on profiling data from multiple executions. In the sections that follow, we describe the meaning of the cost models and how Mprof instantiates them.

4.4.2 Specifying the Cost Models

Assuming that the running time of a benchmark program is completely characterized by the time spent in its data structures, we can set up the equations for instantiating the cost models as follows. Let E be the set of all samples in the profile database. Let D^e be the set of data structures used in the sample e, which has a total running time of t^e. Let s_d^e be the set of statistics collected for the data structure d in the sample e. Let $f_d(s_d)$ be the parameterized cost model that computes the time spent in the data structure d, given the set of measured statistics s_d. We can then derive the following equations from the profile database:

[3] The cost of sending a fixed size message may vary with the workload if the network interface is blocking.

$$\forall e \in E, \left[t^e = \sum_{\forall d \in D^e} f_d(s_d^e) \right] \tag{4.1}$$

The cost model can be further refined to provide more details on the costs of the data structure accesses. We use the term *access category* to denote a collection of accesses, functions, or any user-defined entities in a data structure. The cost model can be refined to give the cost of each access category as follows. Let O_d be the set of access categories in the data structure d. Let $f_{d,o}$ compute the time spent in the access category o, given the measured statistics s_d. We have

$$f_d(s_d) = \sum_{\forall o \in O_d} f_{d,o}(s_d) \tag{4.2}$$

Equations 4.1 and 4.2 provide a method for obtaining the cost models by executing benchmarks and solving equations. Ideally, if the total number of free parameters in the model is N, they can be solved perfectly using N samples from the profile database. However, the models are merely approximations, and there may be errors in measuring the running time, so the equations may be inconsistent. Therefore, we seek a "best-effort" cost model that best fits the samples in the profile database.

Let x be some setting of the model parameters, and let t_x^e be the running time of the sample e indicated by the cost model with setting x. A feasible criterion for assessing the quality of the cost model is its sum of square errors (SSE) on the samples:

$$SSE = \sum_{e \in E} (t_x^e - t^e)^2 \tag{4.3}$$

Direct use of SSE gives more weights to samples with long running times. To give equal weights to all samples, the samples can be normalized with respect to their running time before the model instantiation process begins.[4]

The "best" model is then the one with the minimum SSE. If we assume the measurement error is a normally distributed random variable with zero mean, the model that minimizes SSE is also the maximum likelihood estimator of the true model. The normal distribution assumption is intuitively attractive because, according to the central limit theorem, the sum of many small, independent random variables (e.g., system effects) roughly

[4]To be precise, each term in the cost model is divided by the total running time of the sample.

```
# Declare terms for "pipe"
 Comp                        -- Units of computation per node.
    { Comp = M * T; }

# Declare access categories and their terms for "pipe"
 STARTUP                     -- Start-up cost
    { P }
 COMP                        -- Computation cost
    { Comp }
```

Figure 4.6: Cost models for the PIPE computational kernel

follows a normal distribution. We adopt this statistical interpretation of the samples because it gives us more detailed information about the quality of the cost models.

The method of finding the minimum SSE model has much to do with the characteristics of the functions comprising the model. When the model is a linear function of its free parameters, the best parameter settings can found by many well-known numerical algorithms. We use the family of linear functions for f_d and $f_{d,o}$, not only because they are easier to solve, but also because we intend to interpret each free parameter as the unit cost of some operation, which is usually multiplied by a scalar indicating its frequency and then summed with the costs of other operations, and both of these are linear operators on the parameters.

Figures 4.6, 4.7 and 4.8 show the part of the performance interface that specifies the parameterized cost models for the PIPE program and its data structures. The first section of each figure contains the declaration and definition of *terms*. The declaration of each term consists of its symbolic name and a one-line comment describing its meaning. The definition of each term can be an arbitrary C function of the data structure's declared statistics, such as those defined in Figure 4.5. The terms are linear components of the costs of the access categories, which are defined in the second section. Each access category has a symbolic name, a one-line comment describing its meaning, and a list of terms that comprise its cost. The costs represented by the access categories must be mutually exclusive, and no term can appear in more than one access category.

Each term is associated with a distinct free parameter that denotes its unit cost. The cost of each access category is the sum of the products of its terms and parameters, and

```
# Declare terms for "rts"
 Thread                        -- Thread costs
    { Thread = ThreadAlloc + ThreadEnable + ThreadDisp; }
 Idle                          -- Idle time
    { Idle = IdleCycle * Sched; }
 Msg                           -- Messages
    { Msg = MsgSent + FlowMsgSent; }

# Declare access categories and their terms for "rts"
 STARTUP                       -- Start up cost
    { P }
 THREAD                        -- Thread costs
    { Thread ThreadContext }
 IDLE                          -- Idle time
    { Idle }
 COMM_ALPHA                    -- Communication startup costs
    { RemoteOp Msg FastMsgSent }
 COMM_BETA                     -- Inverse communication bandwidth
    { ByteSent }
 COMM_LOCAL                    -- Local communication
    { ByteCopy }
```

Figure 4.7: Cost models for the runtime layer

```
# Declare terms for "graph"
 Op                            -- Costs of control
    { Op = NOp + NBlock + NMsg; }
 Data                          -- Costs of data movement
    { Data = EnqSize + DeqSize; }

# Declare access categories and their terms for "graph"
 STARTUP                       -- Start up cost
    { P }
 OP                            -- Control cost
    { Op }
 DATA                          -- Data movement cost
    { Data }
```

Figure 4.8: Cost models for the event graph data structure

the cost of the data structure is the sum of the costs of all its access categories. Formally, let $R_{d,o}$ be the set of terms for the access category o of the data structure d, c_r be the free parameter associated with the term r in $R_{d,o}$, and f_r be the function represented by the term r, we have

$$f_{d,o}(s_d) = \sum_{r \in R_{d,o}} c_r * f_r(s_d) \tag{4.4}$$

Only the costs of the access categories are presented to the application programmer. The access categories, which are customized by the library programmer, provide high-level profiling information that is independent of the implementation of the data structure. For example, in Figure 4.7, the `THREAD` category summarizes the costs of the thread system, while the `COMM_ALPHA` and `COMM_BETA` categories summarize the start-up overheads and the per-byte cost of communication, respectively. The cost models can be improved during the instantiation step without invalidating the samples in the profile database, as long as the set of high-level statistics remain the same.

In the next section, we describe our choice of algorithm for solving the system of equations, and how feedback from the solutions can be used to improve the quality of the parameterized cost models.

4.4.3 Instantiating the Cost Models

We use the Singular Value Decomposition method (SVD) to find the best c_r [PTVF92]. The method is robust in the presence of linearly dependent terms and nearly identical samples. Linearly dependent terms do not contribute to the expressiveness of the model, and should be merged into fewer terms. For example, the thread related statistics are merged into the same term `Thread` to simplify the model, because they are usually linearly dependent. Nearly identical samples occur quite often because it is difficult to design experiments that always produces new combinations of statistics. Such samples lead to numerical problems when solving the model equations. Fortunately, the SVD method has internal mechanisms for filtering out such samples.

Developing the right models and experiments is not an easy task. It is often a repetitive process, using the programmer's knowledge of the source code as well as feedback from the model instantiation tool. Using the statistical meaning of SSE, Mprof provides the following feedback about the quality of fit:

- The sum of square errors of the fit. Assuming unit variance of the measurement errors, the SSE follows a χ^2 distribution with $n - m$ degrees of freedom, where n is the number of samples and m the number of free parameters. Although direct use of the χ^2 probability is difficult, we can use it to compare the quality of different instantiations of the cost models.

- The mean relative error (MRE) of the fit [Bre94]. MRE is defined as the geometric mean of the error ratios $(1 + |t^e_x - t^e|/t^e)$ over all samples. It gives the expected error of the fitted model on new samples, assuming the samples in the profile database represent the "common" workload.

- The singular values from the SVD algorithm. The number of non-zero singular values represents the rank of the model equations. If it is less than the number of free parameters, more experiments must be performed to fully instantiate the model.

- The confidence intervals of the fitted parameters. The 90% confidence interval for the fitted parameter c with mean u_c and variance v_c (estimated from the samples) is $u_c + -\sqrt{v_c} \times t[0.95; n-m]$, where $t[0.95; n-m]$ is the 95% quantile of the Student's T-distribution with $n - m$ degree of freedom [Jai91].[5] If the confidence interval contains 0, the parameter is not significantly different from zero, and the corresponding term should be omitted from the model.

The statistical approach to modeling the performance of parallel programs has been taken by other researchers such as Brewer [Bre94] and Crovella [Cro94]. However, their work targeted standalone procedures and a fixed set of overhead categories, and they did not address the composition of concurrent data structures. Also, their applications are regular, bulk-synchronous programs whose performance can be predicted prior to the computation. We target the class of irregular applications whose behavior depends on the input data, and our work addresses post-mortem performance analysis instead of performance prediction.

4.4.4 An Example: Instantiating the Cost Models of the PIPE program

The toolkit provides a simple scripting language to help the programmer manage the instantiation process. Figure 4.9 shows a example script for modeling the PIPE

[5] In practice, we use the unit normal distribution to approximate the T-distribution, assuming sufficiently large samples ($n - m > 30$). The 95% quantile is thus 1.645.

```
FitDatabase:   "stats-cm5-p4.db"
FitDatabase:   "stats-cm5-p32.db"
TestDatabase:  "stats-cm5-p16.db"

ImportParameter:    Rts "cm5.model"  Thread ThreadContext Idle
ImportParameter:    Rts "cm5.model"  RemoteOp Msg FastMsgSent
ImportParameter:    Rts "cm5.model"  ByteSent ByteCopy

IgnoreTerm:    Rts    P
```

Figure 4.9: Sample script for instantiating the cost models of the PIPE program and the event graph data structure.

program and the graph data structure on the CM5. The scripting language contains the following primitives: `FitDatabase`, `TestDatabase`, `ImportParameter`, and `IgnoreTerm`. The `FitDatabase` and `TestDatabase` directives specify the profile databases containing the samples and the test cases, respectively. The `ImportParameter` directive instructs Mprof to load from a file the parameter values that have been instantiated in a previous modeling session. In this example, we have instantiated the cost models of the runtime layer in a previous session, and the resulting parameter values are stored in the file `cm5.model`. Therefore, we use these stored values instead of recomputing them. The `IgnoreTerm` directive instructs Mprof to ignore certain terms in setting up the equations. In this example, we decided to eliminate the runtime layer start-up cost term, because we found in a previous session that it did not contribute to the quality of the cost models.

Figure 4.10 shows the results of the instantiation process. The sample database contains 64 executions of the program on 4 and 32 processors, and the test database contains 32 executions on 16 processors. The fitted model achieves a MRE of 0.64% over 64 samples, and 0.78% over 32 test cases (the distribution of errors is shown in Figure 4.11). The singular values indicate that the rank of the equations is 4, which is less than 5, the number of free parameters. The insufficiency in rank implies that there is redundancy or inconsistency in the equations. By examining the confidence intervals, we discovered that the singularity was caused by the start-up costs, which are not significantly different from 0. Removing these two parameters from the models led to a rank of 4 for 4 terms and eliminated the problem.

The model instantiation process can be repeated after more profiling data is avail-

```
Chi-square: 0.011286
Singular values: 619696.111767 264664.578557 26991.463260 0.000000 6.533606
Sample MRE: 0.637338%
Test MRE: 0.779667%

pipe:P: 0.000862786 +- 0.0128065
pipe:Comp: 4.55768e-06 +- 1.5201e-07

graph:P: 0.00595816 +- 0.00391284
graph:Op: 1.40086e-05 +- 8.42826e-07
graph:Data: 1.52405e-07 +- 4.41832e-08
```

Figure 4.10: Results of the instantiation session. The tool gives general feedback on the quality of the cost models, as well as the fitted parameter values and their 90% confidence intervals.

able or further improvements are made on the cost models. When the library programmer is sufficiently confident of the models (judging by the coverage of the experiments and the model error), the models are saved for use in profiling real executions. Mprof transforms the instantiated models into C functions that are linked into the Multipol library to provide timing information on the executions.

Figure 4.12 shows the profiling output for three PIPE executions on the CM5 which differ only in their degrees of message aggregation (10,000, 1000, and 0 bytes, respectively). The estimated running times of the access categories are summed over all 32 processors. The results show the tradeoff between overhead and latency – lowering the degree of aggregation reduces the amount of idle time (indicated by `IDLE`), but increases the other overheads such as the communication start-up cost (indicated by `COMM_ALPHA`) and flow control costs (indicated partly by `THREAD`, `COMM_BETA`, and `OP`). Aggregating 1000 bytes achieves a good balance of overhead and latency, a result that is also supported by other experiments of message aggregation for the PIPE program. The results also show that the estimated running time from the cost models tracks the actual running time reasonably well – the estimation error is 4.8%, 5.2%, and 4.6%, respectively.

We have described the techniques we use for profiling the costs of data structure accesses, which provides insight into the overhead of the parallel implementation. In the next section, we describe our approach to profiling dependencies between data structure

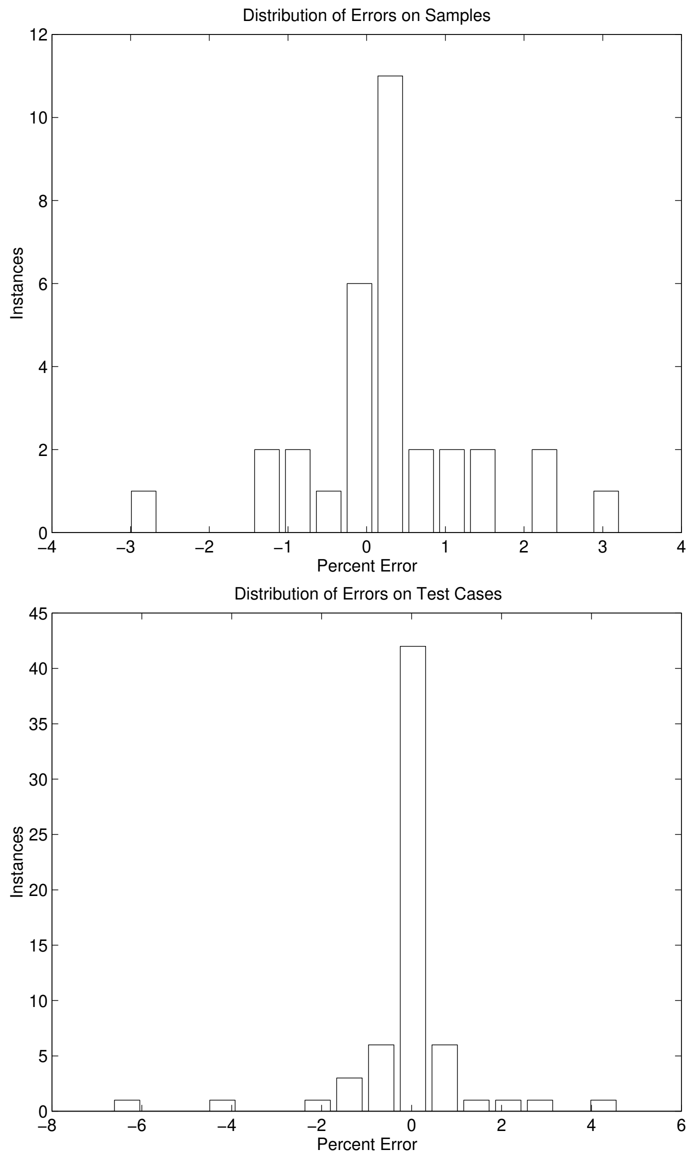


Figure 4.11: Distribution of model errors. The percent errors are obtained by applying the fitted model to the samples and the test cases.

```
% pipe-cm5 32 10000 10 100 15 100 10000

*** Cost Profile  ***
Id 0: THREAD= 38.287 IDLE=187.911 COMM_ALPHA=  7.146 COMM_BETA= 51.635
Id 3: COMP=149.078
Id 1: ACCESS= 15.382 DATA=  4.713
Time: Model= 454.151, Total= 433.928

% pipe-cm5 32 10000 10 100 15 100 1000

*** Cost Profile ***
Id 0: THREAD= 45.912 IDLE=147.729 COMM_ALPHA= 10.317 COMM_BETA= 53.037
Id 3: COMP=149.078
Id 1: ACCESS= 16.539 DATA=  4.713
Time: Model= 427.325, Total= 406.569

% pipe-cm5 32 10000 10 100 15 100 0

*** Cost Profile ***
Id 0: THREAD= 97.915 IDLE=126.807 COMM_ALPHA= 42.902 COMM_BETA= 59.666
Id 3: COMP=149.078
Id 1: ACCESS= 21.024 DATA=  4.713
Time: Model= 502.105, Total= 480.787
```

Figure 4.12: PIPE executions on a 32-node CM5. The command line arguments are the number of processors, M, N, T, F, the size of each element, and the degree of aggregation. The times are summed over all 32 processors.

accesses for detecting performance inefficiency due to insufficient parallelism.

4.5 Identifying the Dependences between Data Structures

Unlike standalone parallel algorithms, the dependencies among data structures cannot always be put in simple forms. Data structures are like components in reactive systems – they react to events in their environment. For example, the PIPE program has different types of dependencies, namely the data dependencies among the consume threads between adjacent nodes, the resource dependencies of the enqueue operations on the dequeue operations, and other dependencies due to split-phase calls. The synchronization pattern is data-dependent and cannot be predicted in advance.

The previous section described the techniques for profiling overhead, the first major source of performance inefficiency. The second major source of performance inefficiency arises from the dependencies between data structures. In this section, we describe the techniques used in Mprof in profiling dependencies. Mprof not only detects the existence of dependencies, but also computes their contribution to the total idle time. It also provides an abstraction mechanism for the library programmer to summarize the profiling information of a large number of synchronization events.

We start by presenting two metrics for characterizing dependencies, *observed latency* and *critical path latency*. Observed latency exposes all potential dependencies whose optimization may lead to performance improvements, while critical path latency exposes only those dependencies that directly contribute to the idle time. We then use the PIPE program as an example to show how the metrics help identify the sources of dependencies and provide insights into optimizations.

4.5.1 Definition of Observed Latency

The *observed latency* of a split-phase access or a thread[6] is defined as the processor idle time from the time it begins to the time it completes. In multithreaded executions, split-phase accesses have a negative effect on performance only if their latency cannot be

[6] A split-phase access is always implemented as a thread. Therefore, we do not distinguish between the terms "access" and "thread" for the rest of the section. An access is an abstract concept unknown to the runtime layer, whereas a thread is a runtime layer abstraction whose properties can be measured automatically.

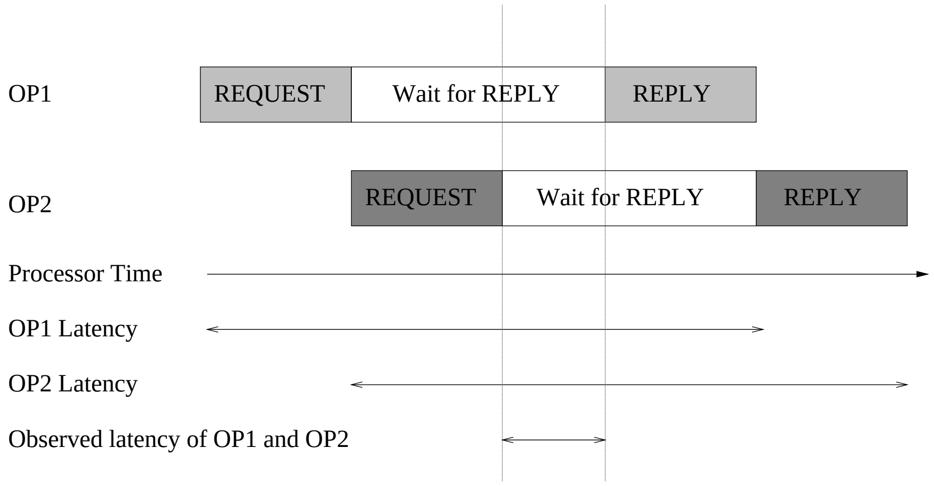


Figure 4.13: Definition of observed latency.

overlapped with other computation. The observed latency measures exactly the part of unoverlapped latency to expose problematic accesses.

Figure 4.13 illustrates the definition of observed latency. The figure shows two split-phase accesses that are pipelined for latency hiding. The total latency of each access is the time between its request and reply phases. The observed latency of both accesses is the amount of processor idle time during their lifetime when no threads are available for execution.

Observed latency is charged to every access whose lifetime contains the latency. Intuitively, it exposes all accesses whose reduction in latency is likely to reduce the total running time, assuming such an optimization does not cause other overheads. The programmer can use this information to identify the candidates for optimization. For example, for the execution in Figure 4.13, the observed latency metric exposes both OP1 and OP2 as potential targets for further optimization.

Observed latency is also charged to the synchronization event that contributes to the latency. A synchronization event is an ordered pair of accesses where the first access enables the second for execution, or equivalently, the second access awaits certain condition posted by the first. The programmer can use this information to traverse the chains of dependencies and identify the source of inefficiency.

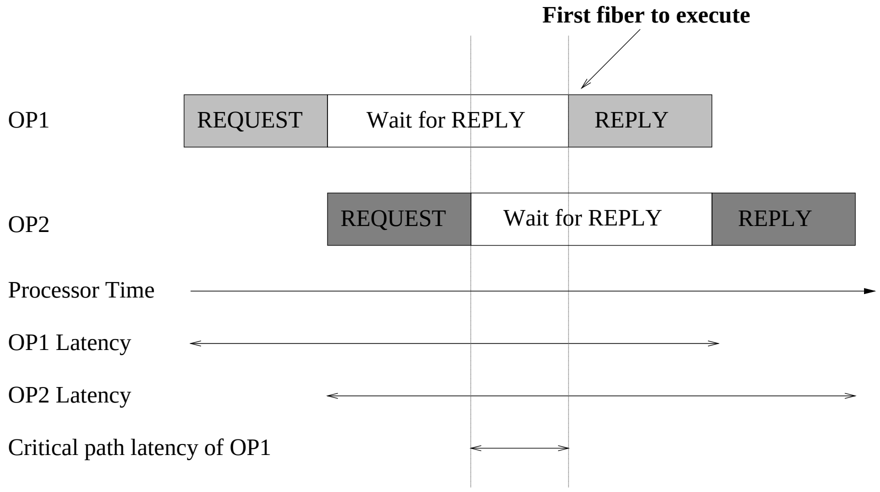


Figure 4.14: Definition of critical path latency.

4.5.2 Definition of Critical Path Latency

The critical path latency of a split-phase access is like its observed latency, except the critical path latency is incurred only if the access is the first to resume execution after a period of idle time. The metric exposes the accesses that "directly" contribute to the processor idle time. Figure 4.14 illustrates the definition of critical path latency. In this example, OP1 has a nonzero critical path latency, while OP2 has 0 critical path latency. OP1 directly contributes to the amount of processor idle time corresponding to its critical path latency, because it lies on the critical delay path in the execution.

Intuitively, the critical path latency exposes the accesses that contribute to the delay in the critical path of the execution. Unlike observed latency, the idle time is charged to exactly one access. Therefore, critical path latency is easier to interpret than observed latency, because it uniquely identifies the problematic access for each period of idle time.

Critical path latency is dependent on the local scheduling policies used in the execution as well as the available parallelism in the application. It merely points out one of the many possible paths of delay. Other long latency accesses that are not on the particular delay path may be left undetected, although their optimization may also reduce the total running time. For example, the operation OP2 has 0 critical path latency, but reducing its latency may also reduce the processor idle time. Therefore, critical path latency should be used in combination with observed latency to thoroughly profile dependencies. As with observed latency, critical path latency is also charged to the synchronization events to expose them for tuning.

4.5.3 Measuring Latency

The non-blocking semantics of Multipol fibers exposes all idle time in the number of idle scheduler cycles. With a minimal amount of bookkeeping on these idle cycles, we can exactly measure observed latency and critical path latency.

Two integer counters C_{total} and $C_{charged}$ are used by each processor to record, respectively, the total number of idle cycles and the number of idle cycles that have been charged to a thread. The runtime layer also records the identity of the executing thread. When a thread is created, the value of C_{total} is recorded in its control block. When a thread is enabled, the identity of the executing thread (that is, its enabling thread) is also recorded in its control block.

When a thread executes, the difference between the current value of C_{total} and the value in its control block is charged to its observed latency. The difference is also charged to the synchronization event formed with its enabling thread. Note that value of C_{total} is the same when the thread is enabled as when the thread executes, because there can be no enabled thread in an idle scheduler cycle. Therefore, the difference is the exact amount of idle time the thread observes.

To compute the critical path latency of a thread, we simply take the difference between the current values of C_{total} and $C_{charged}$. The difference is also charged to the corresponding synchronization event. We charge the critical path latency to a thread only if it waits for a synchronization event, that is, only if it has a non-zero observed latency. This ensures that critical path latency is not all charged to remotely invoked fibers, which provides little insight for tuning. After the critical path latency is computed, the value of C_{charge} is set to the value of C_{total} to avoid charging the same period of idle time to multiple threads.

A real application may generate too many threads and synchronization events to be profiled separately. Therefore, our profiling toolkit provides primitives for grouping threads into *access categories* (as in profiling costs of accesses), and Mprof records dependencies only at the level of access categories. We call this process *abstraction of the dependence profile*. When a thread executes, it declares the identity of its corresponding access category, which is used in place of its own identity for profiling dependencies. Because the identity is dynamically declared, the observed latency and the critical path latency related to a thread cannot be computed prior to its execution.

We use the PIPE program to illustrate the abstraction process. The program itself is naturally decomposed into the `consume` category and the `produce` category, which corresponds to the consume thread and the produce operation in Figure 4.4, respectively. Operations on the event graph data structure are put into two access categories: `ENQ` and `DEQ`. The `ENQ` category includes threads to wait for the buffer space, ship new items over the network, and inform the receiving nodes of the arrival. The `DEQ` category includes threads to wait for new items, remove the items, and inform the sending node of the availability of the buffer space. Other unclassified threads such as those created by the runtime system itself are not profiled.

4.5.4 Optimizing the PIPE Program

We now show how the profiling information can be used in analyzing and optimizing the PIPE program. We start by identifying the operations that have long observed latency and critical path latency. We then use their synchronization events to traverse the chains of dependencies and locate the source of inefficiency.

We focus on optimizing the PIPE execution with 1000 bytes of message aggregation, the best configuration found from profiling overheads in Figure 4.12. The profiling information produced by the runtime layer is shown in Figure 4.15. For each access category whose dependencies are profiled, Mprof shows its total observed latency and critical path latency (in seconds). These are decomposed into the latencies associated with the category's synchronization events.

The category with the longest latency is `consume`, which is the main program thread. By examining its observed latency, we can trace backwards from the synchronization events on `consume` and find two problematic event sequences: `ENQ` → `DEQ` → `consume`, and `DEQ` → `ENQ` → `produce` → `consume`. Event sequence 1 is due to data dependencies between the pipeline stages. Event sequence 2 is due to the resource dependencies in the event graph data structure's flow-control protocol. The critical latency shows that both data dependencies and resource dependencies are responsible for the idle time in the execution.

The data dependencies can be partly resolved with message aggregation, whose effect was shown in Figure 4.12. We now concentrate on fixing the resource dependencies.

```
% pipe-cm5 32 10000 10 100 15 100 1000

*** Cost Profile ***
Id 0: THREAD= 45.912 IDLE=147.729 COMM_ALPHA= 10.317 COMM_BETA= 53.037
Id 3: COMP=149.078
Id 1: ACCESS= 16.539 DATA=  4.713
Time: Model= 427.325, Total= 406.569
*** Observed Latency ****
ENQ1 =  47.337 --  DEQ1: 47.337
DEQ1 =  92.700 --  ENQ1: 92.700
produce3 =  47.337 --  ENQ1: 47.337
consume3 = 134.063 --  DEQ1: 92.700  produce3: 41.363
*** Critical Latency ****
ENQ1 =  47.337 --  DEQ1: 47.337
DEQ1 =  92.700 --  ENQ1: 92.700
```

Figure 4.15: Profiling the dependencies in the PIPE execution.

4.5.4.1 Increasing edge capacity

We can easily remove resource dependencies by allocating a large number of buffers in the event graph data structure. Figure 4.16 shows the result of the execution when the capacity is increased from 15 to 1000. The increase in capacity reduces the running time by 24%, and the flow control event sequence has practically disappeared. The remaining 1.723 seconds of resource dependencies is possibly from processor 0, which does not have a consume thread. The idle time caused by data dependencies is also reduced by the elimination of resource dependencies, because their latency are mutually dependent in the PIPE program.

4.5.4.2 Software Pipelining

For applications with large event graphs such as the CSWEC program, increasing edge capacity may not always be possible. Therefore, instead of eliminating the flow control dependencies, we try to hide their latency using a technique called *software pipelining*, which is also used in advanced compilers for exploiting instruction-level parallelism [Lam87, Joh91].

```
% pipe-cm5 32 10000 10 100 1000 100 1000

*** Cost Profile ***
Id 0: THREAD= 38.905 IDLE= 51.779 COMM_ALPHA=  6.753 COMM_BETA= 52.388
Id 3: COMP=149.078
Id 1: ACCESS= 15.518 DATA=  4.713
Time: Model= 319.134, Total= 307.305
*** Observed Latency ****
ENQ1 =   1.723 --  DEQ1:  1.723
DEQ1 =  37.503 --  ENQ1: 37.503
produce3 =   1.723 --  ENQ1:  1.723
consume3 =  37.503 --  DEQ1: 37.503
*** Critical Latency ****
ENQ1 =   1.723 --  DEQ1:  1.723
DEQ1 =  37.503 --  ENQ1: 37.503
```

Figure 4.16: Effect of increasing event graph capacity on the PIPE program. The running time is decreased by 24%.

We observe that the computation in iteration i is independent of the produce operation in iteration $i-1$, although the enqueue operations must be performed in order as required by the semantics of the event graph data structure. Therefore, we can pipeline different iterations as long as we insert sufficient synchronization primitives to enforce the order of the produce operations. Figure 4.17 shows such an implementation of the PIPE program, where the latency of the produce operation can be partially overlapped with the computation in the next iteration.

Figure 4.18 shows the results of the new program. Surprisingly, the running time is increased by 6%. Although the consume operation waits less frequently for the produce operation, as indicated by the observed latency of their synchronization event, both of their latencies have increased.

One possible explanation of this effect is the delay in propagating messages. Messages smaller than the specified aggregation size are not forced out to the network until the processor is idle. Therefore, keeping the processor busy for a longer period of time may actually reduce parallelism, due to the tight dependencies between the pipeline stages. As a result, the benefit of software pipelining is completely offset by the increase in message latency. The performance result indicates that a different tradeoff must be made between

```
consume:
    repeat
        while (fewer than N items removed by current iteration) {
            [ wait for new items ]
            remove items (up to N items)
        }
        perform (N * T) units of computation
        [ wait for previous produce to complete ]
        issue produce
    until (M items processed)

produce:
    < same as the non-pipelined version >
```

Figure 4.17: Software pipelining implementation of the PIPE program.

```
% pipe-swp-cm5 32 10000 10 100 15 100 1000

*** Cost Profile ***
Id 0: THREAD= 46.586 IDLE=167.977 COMM_ALPHA= 10.152 COMM_BETA= 53.341
Id 3: COMP=149.078
Id 1: ACCESS= 16.990 DATA=  4.713
Time: Model= 448.838, Total= 429.008
*** Observed Latency ****
ENQ1 =  79.898 --  DEQ1: 79.898
DEQ1 = 121.554 --  ENQ1:121.554
produce3 =  79.898 --  ENQ1: 79.898
consume3 = 152.208 --  DEQ1:121.554  produce3: 30.653
*** Critical Latency ****
ENQ1 =  57.342 --  DEQ1: 57.342
DEQ1 = 101.695 --  ENQ1:101.695
```

Figure 4.18: Effect of software pipelining. The running time is increased by 6%.

```
% pipe-flush-cm5 32 10000 10 100 15 100 1000

*** Cost Profile ***
Id 0: THREAD= 45.757 IDLE= 90.608 COMM_ALPHA= 12.377 COMM_BETA= 52.902
Id 3: COMP=149.078
Id 1: ACCESS= 16.118 DATA=  4.713
Time: Model= 371.552, Total= 354.731
*** Observed Latency ****
ENQ1 =  18.982 --  DEQ1: 18.982
DEQ1 =  62.669 --  ENQ1: 62.669
produce3 =  18.982 --  ENQ1: 18.982
consume3 =  77.398 --  DEQ1: 62.669  produce3: 14.728
*** Critical Latency ****
PE31: ENQ1 =  18.982 --  DEQ1: 18.982
PE31: DEQ1 =  62.669 --  ENQ1: 62.669
```

Figure 4.19: Effect of selectively flushing messages. The running time is decreased by 13.5%.

the latency of the produce operation and the latency of the messages.

4.5.4.3 Selective Flushing of Messages

The results from the software pipelining implementation led us to another possible optimization based reducing the delays in message delivery. Instead of using a uniform aggregation size throughout the execution, we can dynamically adjust the aggregation size by flushing messages at selected points of the program. The best place to flush messages is immediately before the computation of each node, because the processor may remain busy for an arbitrary amount of time in the computation, causing messages to be queued for a long time.

Figure 4.19 shows the result of the optimization. The running time is improved by 13.5% relative to the original implementation, due to the reduction in the latency of both data dependencies and resource dependencies.

4.6 Related Work

Many performance profiling tools have been developed for event tracing and visualization. They include PICL/ParaGraph [GHPW90], Pablo [RAN+93], and Vista [Hal95].

These tools collect a trace of communication activities and other user-defined events from the execution and use the trace to generate multiple views of the program's activities over time. Such tools feature large quantities of low-level profiling information. In comparison, Mprof produces profiling information immediately after the execution without requiring a trace file. Furthermore, Mprof provides high-level profiling information instead of raw event traces. However, Mprof provides summary information about the entire execution, and it cannot provide information about a particular time period in the execution.

Instrumentation and sampling methods are popular alternatives to event tracing. Research in this area includes Paradyne [MCC+95] and the work by Crovella and LeBlanc [CL93]. The Paradyne toolkit instruments the program binaries to collect performance statistics. It allows the programmer to specify conditions to dynamically control the degree of instrumentation. Uninstrumented code can be used for program components that do not contribute to performance inefficiencies. The binary instrumentation approach has the least programming overhead for performance profiling, but it is platform-dependent, and it incurs higher overhead because it collects too much low-level information. In comparison, Mprof requires the programmer to instrument the source code, but the instrumented code has less overhead because Mprof uses high-level statistics instead of direct timer calls. Mprof also provides more information on the interaction of different data structures than the Paradyne toolkit.

Crovella and LeBlanc used a set of *performance predicates* to detect performance inefficiencies. A designated processor periodically samples the values of a set of global status variables such as the amount of work in the system and the status of each processor. The values of the variables are used to determine the truth of the performance predicates, which are statements about the state of the system such as the presence of load imbalance. The frequency of occurrences of such predicates can be used to identify significance performance inefficiencies. Their work was restricted to shared memory platforms where global variables can be updated and probed with low overhead, and they did not address the interaction of data structures.

The statistical approach to performance modeling has been taken by Brewer [Bre94] and Crovella [Cro94] in predicting the performance of bulk-synchronous programs. As in Mprof, they assume the cost models are linear combinations of terms. Brewer used statistical models to predict the performance of different implementations of the same procedure. The prediction is used to select the best implementation for a given input. In addition to choosing implementations, Brewer also used numerical methods to automatically tune the selected implementation. Crovella used statistical models to predict the "lost cycles" in a program, which are overhead categories such as the amount of idle time caused by load imbalance. He also developed various tools to help the programmer improve the cost models. The major difference between their toolkits and Mprof is that they model the performance of standalone procedures, but not concurrent data structures. Crovella showed limited reuse of the cost models when the new program consists of simple sequential or parallel compositions of existing programs, but he did not address the irregular composition of data structures. On the other hand, their toolkits provide performance prediction, while Mprof only provides post-mortem performance profiles. To make performance prediction possible, the statistics in their cost models cannot be computed values.

Hollingsworth [HM92] compared several performance metrics for detecting problematic program components which include the *critical path* metric [YM88] and the *NPT* metric [AL90]. The critical path metric measures the amount of time spent by each procedure in the longest running path of the execution. The NPT metric measures the amount of "normalized" execution time for each procedure, where the time is normalized with respect to the amount of available parallelism at the time of measurement. The metric assigns more importance to procedures that cause serial bottlenecks in the execution. Both the critical path metric and the NPT metric contain idle cycles as well as busy cycles. In comparison, Mprof reports on the busy cycles (overhead) and the idle cycles (insufficient parallelism) separately. The former is given by the cost of accesses, and the later by the observed latency and the critical path latency of the accesses.

4.7 Summary

Profiling the performance of Multipol applications presents many challenges. The multithreaded execution model in Multipol makes it difficult to use direct measurement methods such as inserting timer calls, because an operation may be decomposed into multiple fibers that execute at different times. Automatic instrumentation methods also cannot be applied, because they break the library abstraction by providing profiling information about the implementation details of the data structures, such as the running times of the internal functions.

Mprof resolves these problems by collecting user-defined statistics that characterize the costs of high-level operations. The statistics are combined with reusable cost models to provide accurate profiling information with low overhead. The cost models are automatically instantiated by the Mprof toolkit using data from benchmark executions, and they can be reused by many applications because all Multipol computations are built from non-blocking fibers. Mprof also provides a performance interface for the library programmer to customize the profiling information and thereby preserve the library abstraction.

In addition to measuring library and runtime overhead, Mprof also detects insufficient parallelism in the application. It uses two metrics, observed latency and critical path latency, which identify problematic synchronization events in the execution. The non-blocking semantics of Multipol fibers permits the toolkit to accurately measure these metrics.

Chapter 5

Performance Results

The previous chapters describe the Multipol runtime library and profiling support for building irregular applications. In this chapter, we conduct experiments to evaluate the effectiveness of our approach. We use the five applications from Chapter 2 to assess the performance of the Multipol runtime library. We also show how these applications can be analyzed and optimized using the profiling information provided by Mprof. Section 5.1 describes the analysis and optimization of Multipol applications, and Section 5.2 summarizes the optimization techniques and the performance of the applications

5.1 Performance Analysis and Optimization

For each application, we analyze its performance profile, locate the performance inefficiencies, and apply suitable optimization techniques to improve performance. We assume the parallel executions use the maximum number of processors in the system, which is 32 for the CM5 and 8 for the Paragon and SP1. We also quote performance results for a 8 processor Sparc cluster when available. The speedup of the parallel execution is taken with respect to the sequential execution that uses the same program on one processor, because the sequential implementations of the programs are either unavailable, or they do not show consistent performance gain over the parallel implementation (such as the CSWEC program [WY95]).

We start with the initial implementations of the applications, which use the following configuration of the Multipol runtime library:

- The runtime layer attempts to aggregate 1K bytes of data for each physical message.

- The runtime layer uses small messages and small preemptive messages for the communication events whenever applicable (see Section 3.3.3).
- The task stealer data structure uses the urgent scheduler for scheduling task migration threads. All other data structures use the FIFO scheduler for their accesses.

We use the profiling information from Mprof to locate the performance inefficiencies. The performance profile of an execution includes the overhead of the thread layer, the communication start-up overhead, and the data transfer overhead. The performance profile also includes the observed latency and the critical path latency of the data structure accesses, which we use to identify the synchronization events that contribute significantly to the processor idle time. The overheads and latencies are estimated by Mprof using the instantiated cost models of the runtime layer for the CM5, Paragon, and SP1. After locating the performance inefficiencies, we apply the corresponding optimization techniques and assess their performance impact.

5.1.1 The EM3D program

We run the EM3D program with two different settings of the bipartite graph: 10% and 20% remote edges. A graph with 0% remote edges (the best case input, although unrealistic) is also used for comparison. Each processor has 1000 graph nodes with 20 outgoing edges. The processors are logically arranged as a linear array, and each edge may adjoin nodes that are at most 3 processors away. The computation is performed for 100 iterations, and the results exclude the construction and preprocessing of the bipartite graph. The performance of the initial implementation is shown in Figure 5.1.

The running times of the program with 0% remote edges are 6.84, 4.10, and 1.45 seconds for the CM5, Paragon, and SP1, respectively. The Paragon achieves the least efficiency loss relative to the best case (0% remote edges), which is 37% and 25% for 10% and 20% remote edges, respectively. The CM5 and the SP1 suffers efficiency loss of more than 85% when 10% of the edges are remote. Increasing the percentage of remote edges to 20% also has greater impact on the CM5 and the SP1, raising their efficiency loss to more than 140%.

The performance degradation comes from the computation and communication overheads for validating the bipartite graph and the idle time caused by the barriers between

Machine	10% Remote Edges					
	Time		Runtime Layer		Synchronization	Efficiency
	Total	Idle	Threads	Communication	Events	Loss
CM5	12.9	1.87	0.07	0.13 / 1.63	1.79 / 1.79 (barrier)	86%
Paragon	5.14	0.45	0.08	0.43 / 0.11	0.40 / 0.40 (barrier)	25%
SP1	2.72	0.17	0.04	0.18 / 0.55	0.16 / 0.16 (barrier)	88%

Machine	20% Remote Edges					
	Time		Runtime Layer		Synchronization	Efficiency
	Total	Idle	Threads	Communication	Events	Loss
CM5	16.6	1.44	0.07	0.13 / 2.80	1.33 / 1.33 (barrier)	143%
Paragon	5.61	0.46	0.08	0.43 / 0.15	0.40 / 0.40 (barrier)	37%
SP1	3.97	0.51	0.04	0.18 / 0.93	0.50 / 0.50 (barrier)	174%

Figure 5.1: Performance profiles of the EM3D program. The times (in seconds) are averaged over all processors in the system. The communication overheads are the message start-up overhead and the data transfer overhead. The long latency events are annotated with their observed latencies and critical path latencies. The efficiency loss is the slowdown relative to the graph with no cross-processor edges (0% remote).

phases (named "barrier" in the figure). The program performs better on the Paragon, because the Paragon has a higher communication bandwidth, which is demonstrated by its lower data transfer overhead. The communication overhead cannot be reduced by message aggregation because each processor exchanges at most one bulk message with any remote processor in a phase, leaving no room for aggregation. The barriers are also required to enforce the data dependencies between phases. Because the bipartite graph data structure has performed most of the important optimizations, we do not find any further optimization for this program.

We note that the Split-C implementation of the EM3D program is slightly faster than the Multipol implementation on the CM5. This is possibly because the Split-C implementation does not have the overhead of multithreading, and it makes use of the CM5 control network for fast barrier synchronizations. The Multipol runtime layer does not provide accesses to the CM5 control network because it is machine-dependent, and its interface is blocking, which is inconsistent with the fiber semantics.

5.1.2 The Tripuzzle program

We use a initial board containing 7 rows of pegs with an empty position in the

Machine	sequential Time	Parallel Time		Runtime Layer		Synchronization Events
		Total	Idle	Threads	Communication	
CM5	122.9	7.69	2.10	0.86	0.14 / 1.17	1.91/1.91 (sync)
Paragon	66.9	17.60	6.81	1.71	0.91 / 1.70	6.70/6.70 (sync)
SP1	40.2	9.66	2.11	1.22	0.43 / 1.64	2.09/2.09 (sync)

Figure 5.2: Performance profiles of the initial implementation of the Tripuzzle program. The times (in seconds) are averaged over all processors in the system. The communication overheads are the message start-up overhead and the data transfer overhead. The long latency events are annotated with their observed latencies and critical path latencies.

Execution	CM5	Paragon	SP1
FIFO Scheduler, 1K byte aggregation	7.69	17.6	9.66
Preemptive Scheduler, 1K byte aggregation	6.55	16.4	8.70
Preemptive Scheduler, 4K byte aggregation	5.59	14.1	7.90

Figure 5.3: Running time of the Tripuzzle program.

middle of the 5th row. The performance of the initial implementation is shown in Figure 5.2. The program achieves speedups of 16, 3.8, and 4.2 on the CM5, Paragon, and SP1, respectively. The major loss of efficiency results from the latency for draining all hash table insert accesses. The other overheads include the thread and communication layer overheads for accessing remote partitions of the hash table. The communication overhead may be reduced by using a higher degree of message aggregation. To reduce the thread overhead, we examine the source code for the hash table data structure and locate the threads that may be scheduled by the preemptive scheduler while preserving atomicity (see Section 3.2.4 for details on the preemptive scheduler). We make minor modifications to the data structure and the application so that the preemptive scheduler can be used for all insert accesses without compromising atomicity. Figure 5.3 summarizes the performance results.

Exploiting the scheduling semantics reduces the running time by 10% in certain cases. Increasing the degree of message aggregation further reduces the running time by more than 10% due to the reduction in both the communication start-up overhead and the thread overhead for handling the messages. Combining these optimizations, the best implementation of the Tripuzzle program achieves speedup of 22 on the CM5, 4.7 on the

	Task Stealing			
Machine	sequential Time	Parallel Time		Synchronization Events
		Total	Idle	
CM5	43.5	2.77	1.47	1.38/1.37 (term), 0.68/0.01 (remove)
Paragon	10.9	4.11	2.80	2.62/2.05 (term), 2.12/0.20 (remove), 0.40/0.38 (fetch)
SP1	12.3	4.63	2.70	2.70/0.05 (term), 0.89/0.87 (remove), 1.79/1.66 (fetch)
	Task Pushing			
Machine	sequential Time	Parallel Time		Synchronization Events
		Total	Idle	
CM5	43.5	3.22	1.88	1.78/0.48 (term), 1.09/0.83 (remove), 0.49/0.47 (fetch)
Paragon	10.9	2.91	1.50	1.31/0.17 (term), 0.39/0.36 (remove), 0.83/0.78 (fetch)
SP1	12.3	2.84	1.15	1.14/0.10 (term), 0.20/0.16 (remove), 0.89/0.88 (fetch)

Figure 5.4: Performance profiles of the initial implementation of the Eigenvalue program. The times (in seconds) are averaged over all processors in the system. The runtime layer overheads are omitted due to their insignificance. The long latency events are annotated with their observed latencies and critical path latencies.

Paragon, and 5.1 on the SP1.

5.1.3 The Eigenvalue program

We use a 1000 by 1000 random matrix as input to the Eigenvalue program. Two load balancing strategies are compared: task stealing and task pushing. Figure 5.4 shows the performance of the initial implementation. For task stealing, the observed latency of the operations show that both the termination detection operation (named "term" in the figure) and the removal of tasks (named "remove") contribute significantly to the idle time. Fetching remote tasks (named "fetch") also contributes to the idle time, but to a lesser extent on the CM5 and the Paragon, because task stealing attempts to preserve locality. Critical path latency is not as useful as observed latency for the Eigenvalue program, because critical path latency is primarily charged to the termination detection operation, which always runs in the background and overlaps with all other accesses. The performance of the task stealing version may be improved by reducing the task removal latency using a

Execution	CM5	Paragon	SP1
Stealing, initial implementation	2.77	4.11	4.63
+ no aggregation	2.77	2.32	2.68
Pushing, initial implementation	3.22	2.91	2.84
+ no fetches and no aggregation	3.19	1.89	2.06

Figure 5.5: Running time of the Eigenvalue program.

lower degree of message aggregation, since the communication overheads are insignificant and can be traded for the reduction in latency.

For task pushing, fetching remote tasks contributes significantly to the idle time, because task pushing destroys the locality of tasks. we observe that for task pushing, there is no advantage in using pointers to the intervals (instead of the intervals themselves) for the task pools, since most tasks migrate exactly once. Therefore, we can eliminate the remote fetches by using the intervals for the tasks and letting the task stealer data structure send the intervals directly to their destination processors. For task stealing, it is generally a good idea to use pointers for tasks and migrate the data required by the tasks on demand, since tasks may move more than once.

Figure 5.5 shows the results of the optimizations. The optimizations work better on the Paragon and SP1 than on the CM5, possibly because the parallelism in the program is not sufficient to keep the 32 processors on the CM5 busy. Task stealing performs worse than task pushing on the Paragon and the SP1 because the small protocol messages generated by task stealing cannot be propagated as efficiently as on the CM5, which has lower communication overhead and latency. Overall, the optimized implementation achieves speedups of 14.6, 4.70 and 4.59 for task stealing, and speedups of 15.6, 5.77, and 5.97 for task pushing, on the CM5, Paragon, and SP1, respectively.

We note that the number of tasks generated by the program varies with the machine. The CM5 and SP1 generate the same number of intervals, but the Paragon generates fewer intervals. This is due to differences in the floating point arithmetic.

5.1.4 The Phylogeny Program

We use an input with 50 characters and 14 species. Each processor creates a combine thread to combine the failure stores whenever its local task pool runs out of task.

	Task Stealing				
Machine	sequential Time	Parallel Time		R untime System	
		Total	Idle	Threads	Communication
CM5	1089	137.6	37.5	0.79	0.32 / 1.66
Paragon	516	151.75	77.5	0.69	0.04 / 0.00
SP1	308	50.3	4.22	0.67	0.29 / 0.31

Machine	Task Stealing: Synchronization Events
CM5	30.8/30.4 (fetch), 1.05/0.96 (remove), 5.43/5.38 (comb)
Paragon	2.07/2.07 (fetch), 74.0/73.1 (remove), 1.21/1.16 (comb)
SP1	6.10/2.73 (fetch), 0.37/0.26 (remove), 1.90/1.00 (comb)

	Task Pushing				
Machine	sequential Time	Parallel Time		R untime System	
		Total	Idle	Threads	Communication
CM5	1089	434.4	170.9	3.03	1.75 / 1.30
Paragon	516	305.1	119.6	5.83	12.9 / 1.80
SP1	308	156.9	49.0	3.87	4.34 / 1.28

Machine	Task Pushing: Synchronization Events
CM5	95.1/95.1 (fetch), 33.3/28.6 (remove), 46.8/46.8 (comb)
Paragon	103/103 (fetch), 13.2/9.71 (remove), 6.61/6.61 (comb)
SP1	43.3/43.3 (fetch), 3.7/3.7 (remove), 34.8/1.95 (comb)

Figure 5.6: Performance profiles of the initial implementation of the Phylogeny program. The times (in seconds) are averaged over all processors in the system. The communication overheads are the message start-up overhead and the data transfer overhead. The long latency events are annotated with their observed latencies and critical path latencies.

We compare two load balancing strategies: task stealing and task pushing. Figure 5.6 shows the performance profile of the initial implementation.

The speedups for the task stealing implementation are 7.9 on the CM5, 3.4 on the Paragon, and 6.1 on the SP1. On the CM5 and the SP1, the speedups are mainly limited by the latency of fetching remote tasks (named "fetch" in the figure). Removing tasks (named "remove") and combining the failure stores (named "combine") incur less idle time. On the Paragon, the speedup is limited by the latency for removing tasks, which is caused by load imbalance. The latencies of removing and fetching tasks may be reduced by lowering the degree of message aggregation.

Optimizations	CM5	Paragon	SP1
Initial implementation	137.6	151.8	50.2
+ no aggregation	112.3	164.0	50.0

Figure 5.7: Running time of the Phylogeny program

The speedups for the task pushing implementation are considerably worse than the task stealing implementation. The lack of locality caused by task pushing not only increases the number of remote task fetches, but also decreases the opportunities for pruning the search space and consequently increases the amount of redundant computation. On average, the task pushing version generates 50% to 100% more tasks than the task stealing version.

Figure 5.7 shows the performance of the optimized task stealing implementation. Disabling message aggregation improves the running time by almost 20% for the CM5, but it slightly slows down the Paragon execution and has no obvious effect on the SP1. We did not perform the optimizations on the task pushing version, because task stealing is clearly the better load balancing strategy to use.

The Split-C implementation of the Phylogeny program [JY95] showed better speedup than the Multipol implementation on the CM5. We noticed that the performance of the Multipol implementation is very sensitive to the load balancing and scheduling parameters. We think its performance can be improved with further tuning in scheduling.

5.1.5 The CSWEC Program

We assess the performance of the CSWEC program using two input circuits: a 32-bit register file, called REGFILE, and an unknown circuit from the ISCAS benchmark suite, called C2670. The REGFILE circuit features 32 very large subcircuits among a total of 325 subcircuits, so its speedup is essentially limited to 32. The C2670 circuit contains 2033 subcircuits of roughly equal size. The capacity of the event graph is set to 64. Figure 5.8 shows the performance of the initial implementation. For the REGFILE circuit, the program achieves speedups of 26.4, 7.73, and 7.15 on the CM5, Paragon, and SP1, respectively. The speedup on the CM5 is less than ideal due to the idle time caused by load imbalance (named “term” in the figure), insufficient capacity (named “resource”), and

	The REGFILE Circuit				
Machine	sequential	Parallel Time		Runtime Layer	
	Time	Total	Idle	Threads	Communication
CM5	515	19.5	4.18	0.20	0.02 / 0.10
Paragon	396	51.2	1.82	0.55	0.11 / 0.18
SP1	214	29.9	2.24	0.36	0.05 / 0.15

Machine	The REGFILE Circuit: Synchronization Events
CM5	0.8/0.8 (term), 3.21/0.58 (resource), 10.4/2.78 (data)
Paragon	0.20/0.20 (term), 0.76/0.05 (resource), 6.32/1.51 (data)
SP1	0.11/0.11 (term), 5.88/0.10 (resource), 18.1/2.01 (data)

	The C2670 Circuit				
Machine	sequential	Parallel Time		Runtime Layer	
	Time	Total	Idle	Threads	Communication
CM5	990	42.9	8.84	4.17	0.52 / 3.32
Paragon	28172	248	63.5	18.3	8.0 / 8.5
SP1	361	62.3	2.07	10.03	1.70 / 6.75

Machine	The C2670 Circuit: Synchronization Events
CM5	0.46/0.46 (term), 52.5/1.53 (resource), 185/6.59 (data)
Paragon	0.42/0.42 (term), 1843/5.4 (resource), 1175/57.6 (data)
SP1	24.5/0.36 (resource), 47.0/1.62 (data)

Figure 5.8: Performance profiles of the initial implementation of the CSWEC program. The times (in seconds) are averaged over all processors in the system. The communication overheads are the message start-up overhead and the data transfer overhead. The long latency events are annotated with their observed latencies and critical path latencies.

data dependencies (named "data"). The Paragon and the SP1 achieves higher efficiency because they have only 8 processors, and parallelism in the circuit is sufficient to keep all 8 processors busy.

The performance of the program for the C2670 circuit is less regular. The program achieves speedups of 23.1 on the CM5 and 5.79 on the SP1. However, the speedup on the Paragon is over 100. This is due to paging in the single-processor case, when the memory required for the simulation exceeds the physical memory of the machine. In this case, the profiling information provided by Mprof is not accurate, because it fails to model the performance of the virtual memory system. On the CM5 and the SP1, the performance is limited by the overhead of the thread and communication layers as well as the idle time due to dependencies. The thread overhead may be reduced if the preemptive scheduler is used for the event graph accesses. As in the Tripuzzle program, we examine the source code of the event graph data structure and the CSWEC program and make minor changes to use the preemptive scheduler. The communication overhead may be improved by increasing message aggregation, but at the cost of increasing access latency.

Figure 5.9 shows the results of the optimizations for the C2670 circuit. Exploiting the scheduling semantics improves performance by almost 10% in certain cases. Increasing the degree of message aggregation improves performance on the CM5 and the SP1, but not on the Paragon. The performance degradation on the Paragon results from the increase in idle time, which is already high in the initial implementation. These results agree with the results from our prior work on the CSWEC program [WY95],[1] which shows that the optimal degree of message aggregation varies with the platform. For the C2670 circuit, the best implementation of the program achieves speedups of 26.8, 117 (super-linear due to paging), and 7.3 for the CM5, Paragon, and SP1, respectively.

The performance of the program may be further improved by increasing the capacity of the event graph. Figure 5.10 shows the results from our prior work on the effect of memory allocation on the CSWEC program. The effect of increasing capacity is dependent on the input and the machine configuration. In general, a large capacity improves performance by increasing concurrency, and consequently reduces the number of null messages that have to be sent to avoid deadlock. The increase in concurrency also provides more threads for hiding latency, as well as more opportunities for message aggregation. The results show that C2670 is less sensitive to the initial increase of capacity than REGFILE, because it has more subcircuits and therefore more parallelism. A large capacity, however, may degrade the performance of the memory hierarchy because it requires more memory. This is demonstrated by the running time of C2670 on Paragon, which increases by a factor of 3 when the edge capacity is too large.

[1] The experiment settings in the paper are slightly different from the settings here in the version of the runtime layer used and the amount of memory allocated to the simulation.

Optimizations	CM5	Paragon	SP1
Initial implementation	42.9	248	62.3
+ preemptive scheduler	38.3	241	56.7
+ more aggregation (4K byte)	36.9	245	49.5

Figure 5.9: Running time of the CSWEC program on the C2670 circuit.

5.2 Summary

In this chapter, we used information from Mprof to identify performance bottlenecks in the Multipol applications and then applied the following optimizations:

- Exploiting scheduler semantics to reduce the thread layer overhead (Tripuzzle and CSWEC).
- Reducing synchronization events by sending data instead of fetching data, thereby avoiding a roundtrip message (Eigenvalue).
- Decreasing message aggregation to reduce access latency (Eigenvalue and Phylogeny).
- Increasing message aggregation to reduce overhead (CSWEC).
- Preserving locality in task stealing to increase the effectiveness of pruning and thereby reduce the amount of redundant work (Phylogeny).

The best performance of the applications is summarized in Figures 5.11 and 5.12. The EM3D program is not included because the program uses different random graphs for different number of processors. We note that the different machine characteristics require different optimizations. For example, task stealing performs better than task pushing on the CM5, but not on the Paragon and SP1, which do not handle small messages efficiently. Increasing message aggregation for the CSWEC program improves performance on the CM5 and the SP1, because it reduces the communication overhead, but not on the Paragon, where the performance is limited by the access latency. The optimal degree of message aggregation also changes with the workload. The profiling information provided by Mprof helps identify the performance inefficiencies and the appropriate optimization techniques to use.

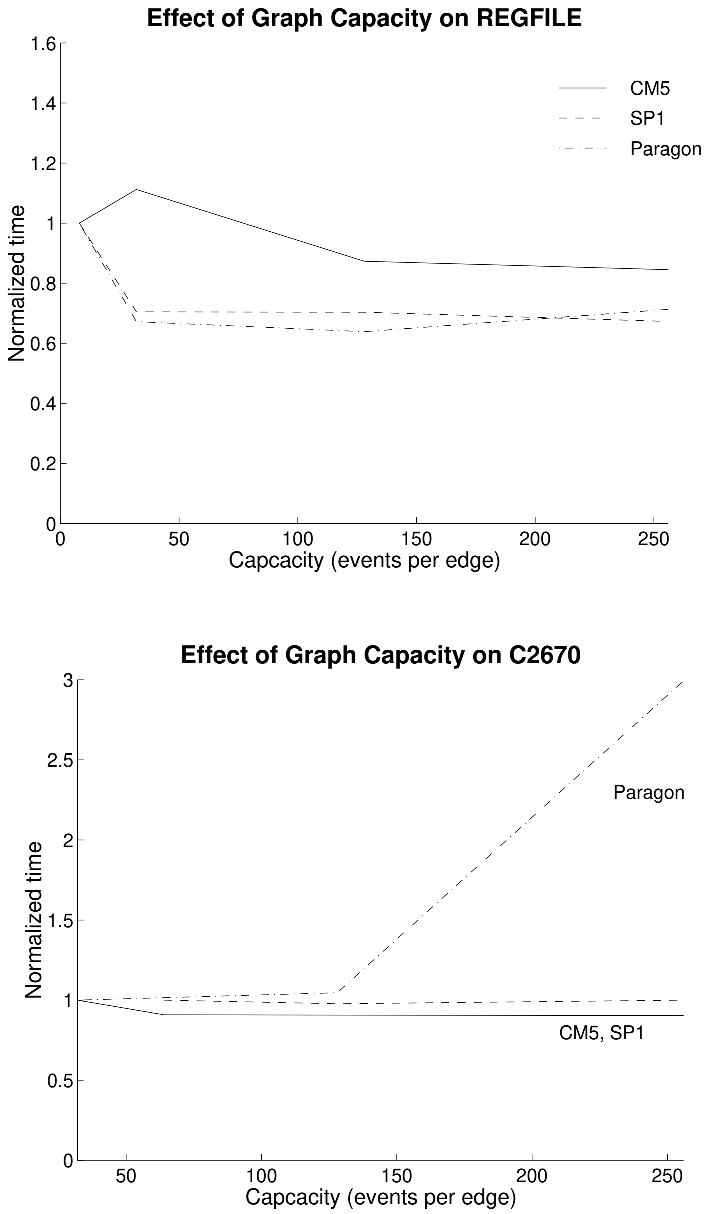


Figure 5.10: Impact of graph capacity on the performance of the CSWEC program. All running times are normalized with respect to the running time with the least capacity.

Program	Platform (Number of Processors)			
	CM5 (32)	Paragon (8)	SP1 (8)	Cluster (8)
Tripuzzle	22.0	4.7	5.1	NA
Eigenvalue	14.6	5.8	6.0	NA
Phylogeny	9.7	3.4	6.1	NA
CSWEC (REGFILE)	23.6	7.7	7.2	6.0
CSWEC (C2670)	26.8	117	7.3	3.0

Figure 5.11: Speedups of the Multipol applications

The good performance achieved by the Multipol applications on multiple platforms indicates that the Multipol library and runtime layer are performance portable. Performance portability is achieved through the use of a multithreaded execution model, the optimized communication layer, and the flexibility in customizing the implementation parameters such as the scheduling policy, the load balancing strategy, and the degree of message aggregation.

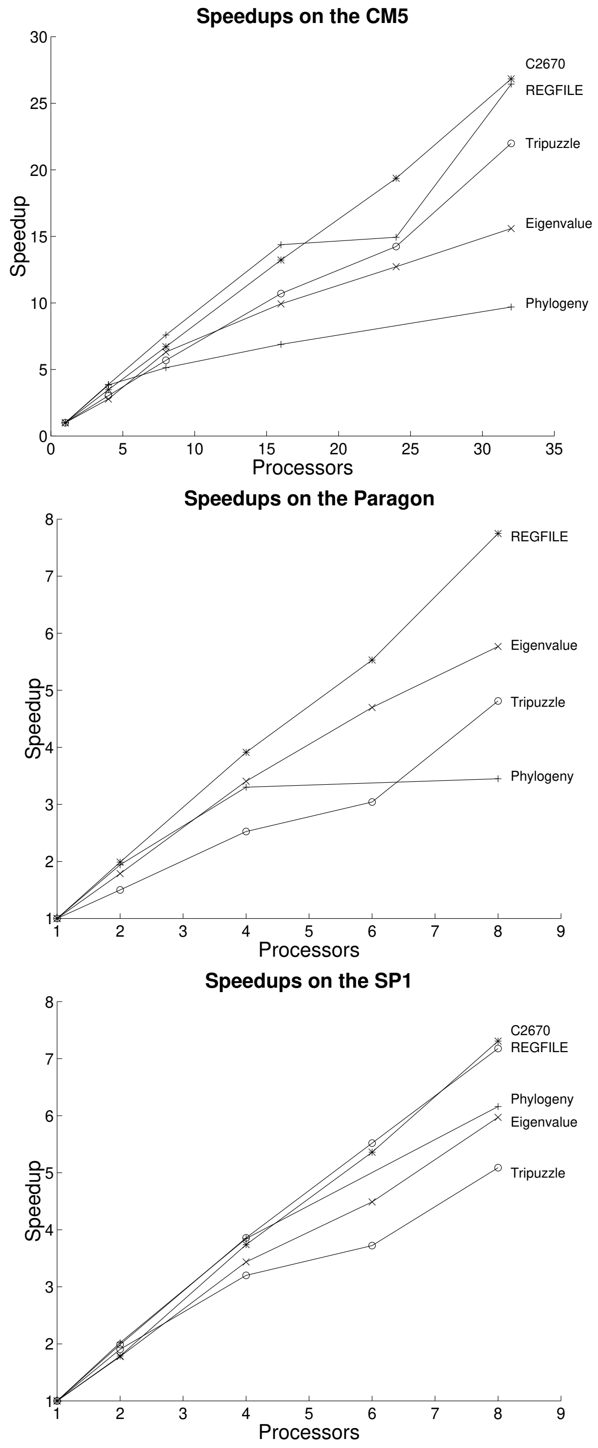


Figure 5.12: Speedup curves of the Multipol applications. The implementation of the store-combining algorithm of Phylogeny requires that the number of processors be a power of two. The Paragon speedup of CSWEC on the C2670 circuit is not reported, because the sequential execution has poor performance due to paging.

Chapter 6

Summary and Conclusions

Our work addressed three important issues in developing irregular applications: programming abstraction, runtime infrastructure, and performance profiling.

We studied the programming abstractions used by several irregular applications and their distributed data structures from the Multipol library, and we described the interfaces of these data structures and their implementation techniques. We also used the application workload to quantitatively characterize the irregularities that arise in parallel programs. The study led to insights into the runtime infrastructure for building irregular applications.

We developed the Multipol runtime layer to support irregular applications on distributed memory platforms. The runtime layer provides a multithreaded execution model and portable communication and synchronization abstractions. It can be easily customized to match the requirements of a particular application and machine. The runtime layer also performs extensive optimizations such as dynamic message aggregation to reduce overhead.

To help the programmer tune the performance of irregular parallel programs, we developed a performance profiling toolkit called Mprof. Mprof provides high-level profiling information that can be customized for each data structure. In addition to measuring the cost of operations from the library and runtime layer, Mprof also identifies problematic synchronization events to detect insufficient parallelism. The design of the runtime layer enables Mprof to accurately profile the executions with low overhead.

Finally, we evaluated the effectiveness of the Multipol performance tools using the Multipol applications. We showed how profiling information from Mprof can be used to identify performance inefficiencies and how the corresponding optimizations reduce these inefficiencies. The applications exhibit high performance on three different platforms, thereby demonstrating the performance portability of the library and runtime layer.

6.1 Future Work

The performance tools we built for Multipol provide a framework for further research on the programming and tuning of irregular parallel applications. In the paragraphs that follow, we describe extensions of our work and suggest future research directions.

The atomicity of Multipol fibers facilitates concurrency control, scheduling, and performance profiling, but large-granularity fibers may adversely affect performance. For example, once a speculative fiber starts executing, it cannot be preempted by a high priority fiber. Therefore, the programmer is burdened with task of decomposing long-running threads into fibers with reasonable granularities. Better linguistic support and compilation tools would ease the programming task. The prototype compiler developed by Jones and Papavassiliou [JP95] is the first step towards such a goal.

The reusability of the cost models depends on the availability of a non-blocking communication interface on the machine so that the overhead of sending a message of a given size can be approximated by a fixed value. With CMAML active messages, for example, this property does not hold, because the sender may be blocked for an arbitrarily long time if the network is congested. Our conclusion from this work is that a non-blocking communication interface is essential for latency hiding and performance portability.

Our performance models do not account for the effects of the memory hierarchy, such as paging and caching. For standalone procedures, detailed models can be developed to take these effects into account, because when the procedures run, they have exclusive access to memory resources. In contrast, when data structures are composed the memory resources are shared, and their access cost depends on the interaction of the data structures. Because each performance interface describes a data structure in isolation, it cannot be used to characterize the access cost of memory resources. Modeling the performance of the memory hierarchy in the presence of interacting program modules is a difficult research problem whose importance is not limited to parallel programming.

Our framework would benefit from more detailed models of all Multipol data structures, whereas this dissertation only includes a detailed model for the runtime layer. Ultimately, our performance interface could be extended to describe performance problems specific to a data structure and the optimization techniques for resolving them. For example, the runtime layer could use information from the performance interface to dynamically adjust the degree of message aggregation based on the profiling data from Mprof. The extended performance interface would document the library programmer's intuition about optimizing the data structure and is the first step towards automatic performance tuning.

6.2 Contributions

Our work produced a comprehensive study of irregular parallel applications and three integrated software components for building them: a data structure library, a runtime layer, and a performance profiling toolkit. We summarize our contributions in each area.

Irregular applications: We studied several nontrivial applications to quantitatively characterize the irregularities in parallel programs. The study covered a wide spectrum of irregular problems. We also implemented the CSWEC application, which is the most irregular application described in the dissertation. Our CSWEC implementation achieved high efficiency on all the platforms we examined.

Data structure library: We developed the task stealer and the event graph data structures. The task stealer data structure has been used by many applications including the Eigenvalue program and the Phylogeny program; it is useful for applications with severe load imbalance such as divide-and-conquer and branch-and-bound problems. The event graph data structure is used by the CSWEC program, and is applicable to any discrete event simulation problem.

Runtime layer: We designed and implemented the Multipol runtime layer. We devised mechanisms for implementing split-phase operations and application-specific scheduling policies. We also developed a novel mechanism for taking distributed snapshots, the snapshot data structure, which is used by all the data structures described in this dissertation. In the implementation of the communication layer, we demonstrated the use of automatic message aggregation for trading off parallelism and communication bandwidth.

Performance profiling: We developed the Mprof performance profiling toolkit which detects both overhead and insufficient parallelism in irregular parallel programs. We applied statistical techniques in modeling the performance of concurrent data structures, and we devised two new metrics, observed latency and critical path latency, which can be used to identify problematic synchronization events. The toolkit exploits the atomicity of fibers to accurately measure these metrics. We also developed a performance interface for customizing the profiling information to preserve the library abstraction.

The Multipol library, runtime layer, and profiling toolkit provide a unique environment for implementing and optimizing irregular applications. They give programmers a set of reusable abstractions and performance analysis tools for developing portable programs on distributed memory platforms.

Bibliography

[ABB+92] E. Anderson, Z. Bai, C. Bischoff, James Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchev, and D. Sorenson. *LAPACK Users' Guide*. SIAM, 1992.

[ABC+88] Frances E. Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[AL90] T. Anderson and E. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proc. 1990 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May 1990.

[BCF+95] N. J. Baden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet - a gigabit-per-second local-area network. *IEEE-Micro*, 15, February 1995.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming*, 1995.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[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, Department of Electrical Engineering and Computer Science, 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. *Concurrency: 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.

[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.

[CKK+94] David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. The generic active message interface specification. Technical report, Computer Science Division, University of California, Berkeley, 1994.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 1985.

[CL93] M. Crovella and T. LeBlanc. Performance debugging using parallel performance predicates. In *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993.

[CLMY96] David Culler, Lok T. Liu, Richard Martin, and Chad Yoshikawa. LogP performance assessment of fast network interfaces. *IEEE Micro*, 1996.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11), April 1981.

[CMF92] Thinking Machines Corporation. *CM Fortran Reference Manual*, December 1992.

[CR95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[Cro94] M. Crovella. *Performance Prediction and Tuning of Parallel Programs.* PhD thesis, Computer Science Department, University of Rochester, August 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. Implementing an irregular application on a distributed memory multiprocessor. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

[DDR94] J. Demmel, I. Dhillon, and H. Ren. On the correctness of parallel bisection in floating point. Technical Report UCB//CSD-94-805, UC Berkeley Computer Science Division, March 1994.

[Dem89] James Demmel. LAPACK: A portable linear algebra library for supercomputers. In *Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, Tampa, FL, Dec 1989.

[DHU+93] R. Das, Y. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the Scalable Parallel Libraries Conference*, Starkville, MS, 1993.

[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.

[FKOT91] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuccke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1991.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. Also appeared in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[Fox92] G. C. Fox. Hardware and software architectures for irregular problems. *Unstructured Scientific Computation on Scalable Multiprocessors*, 1992.

[GHPW90] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: A portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, October 1990.

[GSC95] Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems fo r Scalable Computers*, Rensselaer Polytechnic Institute, NY, May 1995.

[Hal95] Robert H. Halstead Jr. Understanding the performance of parallel symbolic programs. In *Proceedings of the Parallel Symbolic Languages and Systems Workshop*, Beaune France, October 1995.

[Hig92] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 0.4*, 1992.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machine. *Supercomputing*, pages 86–100, November 1991.

[HM92] Jeffrey K. Hollingsworth and Barton P. Miller. Parallel program performance metrics: A comparison and validation. In *Supercomputing '92*, Minneapolis, 1992.

[Ho94] Kinson Ho. *High-level Abstractions for Symbolic Parallel Programming (Parallel Lisp Hacking Made Easy)*. PhD thesis, Computer Science Division, University of California at Berkeley, 1994.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[Joh91] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

[Joh93] K. L. Johnson. Private communication. Available via anonymous ftp from cag.lcs.mit.edu as /pub/tuna/tripuz-entry, November 1993.

[JP95] J. Jones and V. Papavassiliou. A compiler for Multipol. Project report for CS264: Implementation of Programming Languages, 1995.

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

[KAP95] K. Keeton, T. Anderson, and D. Patterson. LogP quantified: The case for low-overhead local area networks. In *Proceedings of Hot Interconnects III: A Symposium on High Performance Interconnects*, Stanford, CA, August 1995.

[KB95] Scott R. Kohn and Scott B. Baden. A parallel software infrastructure for structured adaptive mesh methods. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[KK93] L. V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *ACM Symposium on Parallel Algorithms and Architectures*, June 1992.

[Lam87] Monica S. Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1987.

[LC95] Lok T. Liu and David E. Culler. Evaluation of the Intel Paragon on active message communication. In *Proceedings of Intel Supercomputer Users Group Conference*, June 1995.

[LMSK91] Shen Lin, M. Marek-Sadowska, and E.S. Kuh. SWEC: A stepwise equivalent conductance timing simulator for CMOS VLSI circuits. In *Proceedings of the European Conference on Design Automation*, Amsterdam, Netherlands, February 1991.

[Lun94] Steve Luna. Implementing an efficient portable global memory layer on distributed memory multiprocessors. Master's thesis, Computer Science Division, University of California at Berkeley, 1994.

[MCC+95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, November 1995.

[MKH91] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transaction on Parallel and Distributed Systems*, 1991.

[MSH+95] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Pro-*

ceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, July 1995.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1994.

[PGH+90] Constantine Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In *Languages and Compilers for Parallel Computing.* MIT Press, 1990.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[RAN+93] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, 1993.

[SK91] Wei Shu and L. V. Kale. Chare kernel – a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, March 1991.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, 1992.

[WHJ+95] Deborah A. Wallach, Wilson C. Hsieh, Kirk Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 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. An earlier version appeared as UCB Technical Report CSD-93-723.

[WY95] Chih-Po Wen and Katherine Yelick. Portable runtime support for asynchronous simulation. In *International Conference on Parallel Processing*, Oconomowoc, Wisconsin, August 1995.

[YCD+95] K. A. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C. Wen. Parallel data structures for symbolic computation. In *Workshop on Parallel Symbolic Languages and Systems*, October 1995.

[YM88] C. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *8th International Conference on Distributed Computing Systems*, San Jose, CA, 1988.