Protocol Compilation: High-Performance Communication for Parallel Programs

by

Edward W. Felten

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1993

Approved by_____

(Co-Chairperson of Supervisory Committee)

_____

(Co-Chairperson of Supervisory Committee)

_____

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Protocol Compilation: High-Performance Communication for Parallel Programs

by Edward W. Felten

Co-Chairpersons of Supervisory Committee: Professor John Zahorjan
Professor Edward D. Lazowska
Department of Computer Science
and Engineering

One obstacle to the use of distributed-memory multicomputers has been the disappointing performance of communication primitives. Although the hardware is capable of moving data very quickly, the software has been unable to exploit this potential.

The main cause of this "communication gap" is *protocol overhead.* In addition to moving the application's data between processors, communication systems must engage in other communication and synchronization to ensure correct execution. Protocol overhead is an inherent attribute of parallel computation — it occurs on all machines, under all programming models.

To reduce the effect of protocol overhead, I advocate the use of *tailored protocols* — communication protocols custom-designed to work with a particular application program. A tailored protocol takes advantage of advance knowledge about the behavior of the application program to "slim down" the protocol. Unfortunately, designing tailored protocols by hand is very difficult.

The process of designing tailored protocols can be automated. A *protocol compiler* is a tool that takes an application program as input, and produces as output a tailored protocol for that program. This tailored protocol is plug-compatible with the existing message-passing library, so its use is transparent to the application program.

The bulk of this dissertation discusses the principles underlying the design of a protocol compiler, including the forms of program analysis the protocol compiler must

carry out. In addition to this general discussion, I describe the details of Parachute, a working prototype I have constructed. Parachute generates tailored protocols for data-parallel programs running on the Intel series of multicomputers.

Performance models predict, and experiments with Parachute on real applications confirm, that using a protocol compiler leads to a large decrease in communication time. Experiments indicate that Parachute, despite several engineering compromises in its design, cuts communication time by 17%. Reductions in overall application running time range from 0% to 20%, with an average reduction of 7.5%. More aggressive implementation, or new hardware, will lead to even larger speedups.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I'd like to thank everyone who has helped make the UW a uniquely fun and productive place for me to learn. The faculty, and many of my fellow students, took the time to teach me both inside and outside the classroom. The friendly atmosphere was maintained by a little effort from everyone. I also want to thank the technical and administrative staff, who we often take for granted because everything runs so smoothly. Frankye Jones deserves a special mention for her ability to solve problems even faster than Student Accounts can cause them.

I am especially grateful to my advisors, Ed Lazowska and John Zahorjan, for teaching me not only how to do research, but how to communicate and how to be a professional computer scientist. Having two advisors is more than twice as good as having one!

Thanks are due to Battelle Pacific Northwest Laboratory for providing time on their iPSC/860; the experiments described in this dissertation were all performed on their machine. Intel Supercomputer Systems Division and the Caltech Concurrent Supercomputing Facility also provided me with computer time, which was useful in earlier stages of this research. On the software side, I would like to thank Matt Rosing for providing me access to his extensions to the iPSC/860 operating system, and Roy Williams and Sharon Brunett for helping me find application programs.

My research has benefited from conversations with many people. Among them are Tom Anderson, Scott Berryman, Brian Bershad, Guy Blelloch, Gaetano Borriello, Jeff Chase, David Culler, Jamie Frankel, David Keppel, Alex Klaiber, Richard Ladner, Jim Larus, Rik Littlefield, Steve Otto, Paul Pierce, Matt Rosing, Joel Saltz, Burton Smith, Larry Snyder, Chandu Thekkath, Martin Tompa, David Walker, and Kathy Yelick.

I would like to thank those who provided the money to support me through graduate school. The Mercury Seven Foundation and AT&T Bell Laboratories both provided generous fellowships.

Finally, I'd like to thank my family, for the support they have given me throughout the years of my education. As always, my deepest gratitude is to Laura, for her constant love, advice, and encouragement.

Chapter 1

INTRODUCTION

He who would do good to another, must do it in minute particulars
General good is the plea of the scoundrel, hypocrite and flatterer:
For Art and Science cannot exist but in minutely organized particulars.
— *William Blake,* Jerusalem

Massively parallel computers are becoming the tool of choice in attacking large, "grand challenge" problems in science and engineering. In contrast to conventional supercomputers, massively parallel machines can take advantage of the rapidly increasing speed of mass-produced microprocessors. The crucial question is whether massively parallel machines can continue to scale successfully to solve larger problems on larger machines.

Communication performance is one factor that threatens to limit scalability. Although communication hardware is capable of moving data quickly between the processing nodes of a parallel machine, applications are unable to achieve communication performance matching the hardware's capacity. I call this situation the *communication gap.*

Figure 1.1 shows the communication gap on the Intel DELTA multicomputer. The hardware can transmit a packet between processors with a latency of less than one microsecond, but even a cleverly-written user-level program sees a 67 microsecond latency. The communication gap is a factor of seventy for this machine; large gaps also exist for other vendors' systems.

The communication gap affects programmers in two important ways. First, it limits the granularity of programs that can run successfully — programs whose computational grainsize is larger than the hardware latency but smaller than the observed

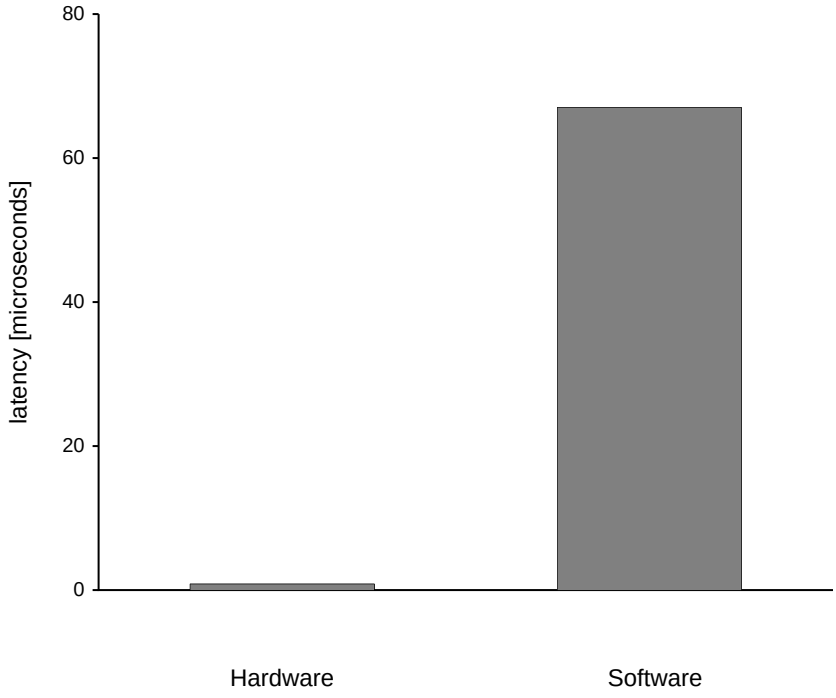

Figure 1.1: The communication gap.

latency see their performance destroyed by the communication gap. Second, it degrades the performance of even coarse-grained programs, forcing the programmer or compiler to optimize these programs aggressively. This has the effect of making the parallel machine even harder to program than it already is.

One key to unlocking the performance potential of massively parallel computers is understanding how the communication gap arises and what can be done to narrow it. This goal has driven my research, and motivates the work described in this dissertation. While others have addressed some aspects of the problem, I will argue that there is one aspect, protocol overhead, that has not been addressed. My work focuses on methods for reducing protocol overhead.

1.1 Sources of the Communication Gap

The communication gap is caused by three factors:

- *protection boundaries:* Communication is often provided as a service of the operating system kernel. The need to cross between user-level and kernel-level makes communication more expensive.

- *data movement:* The interface that the communication hardware offers may not match the software's needs. The resulting semantic gap adds to software costs.

- *protocol overhead:* Communication does more than simply move data from place to place; it must also use some kind of protocol to ensure that data arrives at the right time and place. The overhead of this protocol inflates communication costs.

To dramatically improve the performance of communication primitives, we must find ways to reduce each of these three costs.

In the following subsections, I discuss each of these factors in turn. My conclusion is that while the first two have been addressed by recent work of others, the third has been largely ignored. It is this third factor, protocol overhead, that is the topic of this dissertation.

1.1.1 Protection

Like traditional uniprocessor operating systems, multicomputer operating systems have two main responsibilities: protecting jobs from each other, and managing the allocation of shared resources to jobs. In the context of communication, the system must ensure that

- user-level processes can communicate directly only with other processes with within the same job, and

- user-level processes cannot conspire to monopolize the shared communication network.

The usual way of achieving these guarantees is to make communication a service of the operating system kernel. Since all communication operations are protected operations, the kernel can ensure that user-level processes play by the rules.

One problem with putting communication in the kernel is that crossing protection boundaries is costly [Anderson et al. 92, Anderson et al. 91]. There are several reasons for this:

- *context switch costs:* Crossing into the kernel requires that registers be saved and restored. It also generally causes some cache and translation-buffer misses, due to a change in locality.

- *lack of trust:* The kernel must be especially careful not to corrupt the system's state. As a result, the kernel must check parameters to system calls, to make sure they are legal, that pointers point to legally accessible memory, and so on.

- *fixed interface:* The interface at the user-kernel boundary must satisfy a wide variety of clients. In addition, this interface is very difficult to change, so applications are often required to bridge a semantic gap between the functionality they want and the functionality that the interface actually provides.

To eliminate the cost of crossing protection boundaries, we must eliminate the need to cross boundaries. We must allow application processes to communicate directly with each other, without kernel intervention, but without sacrificing protection.

Safe, User-Level Communication

The solution to this problem is to use carefully designed hardware support, to make sure that applications can perform only safe, legal actions. This problem can be solved by using a combination of five techniques. The Thinking Machines CM-5 employs all but the last of these techniques. The five techniques are:

- *Divide the machine into partitions.* The machine must be divisible into partitions, where a partition is a set of nodes and the network connecting them. Partitions have the property that sending a message between two nodes in the same partition uses only network channels within that partition; this ensures that mutual deadlock between applications cannot occur. Partitions should be changeable in software, but this can acceptably be a heavy-weight operation.

- *Validate message destinations in hardware.* The hardware must check the destination of each message, to ensure that it is being sent to a processor in the same partition.

- *Use strict gang scheduling.* Each parallel job must run in one partition, but several jobs may be running in the same partition, with the operating system time-slicing the partition's CPUs between them. To guarantee protection, we must use strict gang scheduling in each partition; this means that all nodes in the partition run the same job at the same time — nodes in a partition must all switch jobs at the same time. This prevents a message sent by one job from being delivered to another job in the same partition.

- *Support saving and restoring the network state.* When a partition context-switches from one job to another, we need some way of saving the state of the network. Generally, some messages (and partial messages) from the switched-out job will be in the network at the time of the context-switch, and these messages must be saved somewhere so the network is clear for the newly switched-in job. Later, when the original job is rescheduled, the saved network state must be restored.

  The CM-5 saves network state by putting a region of the network in "all fall down" (AFD) mode. AFD mode causes all messages to be delivered to the nearest node, rather than to their destinations. (This allows messages to be cleared very quickly from the network, without making any node handle more than a few messages.) Nodes receiving these messages save them in memory, and then resend them when the job is rescheduled.

  AFD mode relies on two assumptions about the CM-5 network: message packets are limited to 20 bytes, and in-order delivery of messages is not guaranteed[1]. Both of these assumptions may be false for other hardware models, so we cannot always use AFD mode. However, it is not hard to design other hardware mechanisms that allow network state to be saved and restored.

---

[1] *In-order delivery* means that messages from a given sender to a given receiver are received in the same order in which they were sent.

- *Provide a separate logical communication network for the kernel.* The machine runs a general-purpose operating system, so the operating system kernel might have to communicate at any time. For example, the kernel might need to service a page fault, or receive the result of an input/output operation, or coordinate a context switch between jobs. The kernel must always be able to communicate, regardless of how application programs use the network.

  The only way to provide this guarantee is to have a separate logical network for the kernel. Each node must have *two* network interfaces, one for user-level programs, and one reserved for the kernel. Fortunately, we needn't implement two separate physical networks — the two logical networks can be multiplexed over a single physical network, using virtual channels [Dally & Seitz 87, Dally 92] as in the J-machine [Dally 90].

The CM-5 nearly satisfies these requirements. Its only deficiency is that it does not have a separate network for the kernel. However, the CM-5 can offer user-level communication if the operating system is slightly restricted.

While support for user-level communication has not yet been adopted by multicomputer vendors, the clear advantages of user-level communication will drive them to adopt this technology. I will assume that user-level communication will find its way into practice soon, and thus that removing the cost of crossing protection boundaries is a solved problem.

1.1.2 Data Movement

The second cause of the communication gap is inefficient low-level movement of hardware data packets between processors. Data can be moved between processors by manipulating the hardware directly, but adding a thin layer of abstraction above the hardware allows a simpler and more portable interface. Several efficient data movement systems have been devised.

The best-known such system is Active Messages [von Eicken et al. 92]. An active message is a small packet whose first word contains a pointer to a "handler" procedure that is invoked when the packet arrives at the destination processor. The handler is responsible for incorporating the packet's data into the ongoing computation on the receiving processor. The handler must complete promptly, and thus is not always

allowed to perform certain potentially blocking operations, such as sending a message. Both interrupt-based and polling versions of active messages exist; the polling version is most often used, but it requires the application program to actively check for incoming packets.

Another common idea in data movement packages is supporting read and write operations on remote processors' memory. This was originally proposed by Spector, who provided a microcoded implementation on networked workstations [Spector 82]. A later version of the same idea appears as part of the Active Messages system. Thekkath [Thekkath et al. 93] has proposed a general remote-memory-operation system suitable for multicomputers or for networked workstations.

The forthcoming Cray T3D system supports a NUMA model, in which each memory location is attached to one processor, but processors can directly access each other's memory. This is one form of hardware support for data movement.

Finally, some machines have block-copy hardware that allows data to be moved directly between the memories of different nodes. This hardware can be used to provide fast data movement.

There are many suggested mechanisms for efficient data movement. Although the tradeoffs among these mechanisms are not yet clearly understood, the competing mechanisms are finding their way into practice. As in the case of user-level communication, we can assume that efficient data-movement mechanisms will be available soon, and hence treat this as a solved problem.

1.1.3 Reducing Protocol Overhead

The third source of the communication gap is protocol overhead. Reducing protocol overhead is the main focus of my work.

Protocol overhead is the cost induced by the communication protocol that manages the movement of data between processors. The protocol is responsible for ensuring that data is delivered to the application program at the correct time; consequently, the protocol performs buffering and software flow control. Later chapters will discuss in more detail why a protocol is necessary and how typical protocols work.

To reduce protocol overhead, I advocate the use of *tailored* protocols, which are specially designed to fit the behavior of a particular application program. A tailored protocol is correct and fast when used with the application program it was designed

for, but might be slow, or even incorrect, if used with other applications. By taking advantage of what is known about an application's behavior, a tailored protocol can take shortcuts that improve performance.

Designing and verifying tailored protocols is a difficult process. Rather than requiring the programmer to design a tailored protocol, I automate the process by introducing a tool called a *protocol compiler*. The protocol compiler accepts as input a description of the application program's communication behavior, and produces as output a program that implements a tailored protocol. Using a protocol compiler allows the programmer to reap the benefits of tailored protocols, without the work required to design them.

The bulk of this dissertation is about how to construct a protocol compiler, and about the performance improvements it affords, as indicated by experiments with Parachute, a prototype protocol compiler that I have constructed.

1.2 Contributions of this Dissertation

This dissertation makes the following contributions:

- It identifies protocol overhead as a ubiquitous performance problem in parallel computing.
- It suggests the use of tailored protocols as a means of reducing protocol overhead.
- It elucidates the general issues underlying the design of protocol compilers.
- It demonstrates the viability of the protocol compiler concept through the construction of a working prototype.
- It analyzes the performance benefits of using a protocol compiler, both through general modeling, and through experiments with the prototype.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses general issues surrounding communication in various programming models, including

the hardware and software of message-passing machines, and concludes by recommending the use of tailored protocols. Chapter 3 discusses the choices available in designing tailored protocols, and introduces protocol compilers. In Chapter 4 I discuss data-parallel programs and how the protocol compiler can represent their behavior. Chapter 5 discusses three important issues in the protocol compiler's analysis, and how Parachute, my prototype, addresses them. Chapter 6 presents further details of Parachute. Chapter 7 evaluates the performance of the protocol compiler and the protocols it generates. Finally, Chapter 8 summarizes my results, discusses related research, and presents some ideas for future work.

Chapter 2

STRATEGIES FOR SUPPORTING COMMUNICATION

> *No, I had no problem communicating with Latin American heads of state — though now I do wish I had paid more attention to Latin when I was in high school.*
>
> — *Dan Quayle*

We can think of the execution of a parallel program as consisting of two components: local computation and communication. Local computation refers to the normal processing that takes place within each thread of control, while communication refers to the data movement and synchronization required to bind the threads together into a parallel program. We already understand how to support local computation — decades of using sequential machines have taught us how to do this very efficiently.

The same is not true of communication. Although existing operating systems and distributed systems support interprocess communication, these systems use mechanisms that are better suited to sequential and distributed computation than to the kinds of communication that parallel programs do. Since parallel programs have different characteristics, they require different kinds of support.

This chapter discusses the problem of how to support communication in parallel programs. The chapter has three parts. First, I introduce some general issues, using an abstract model of how parallel programs work. Next, I discuss the various parallel programming models, and conclude that the choice among models is less clear-cut, and for our purposes less important, than it initially appears to be. Third, for concreteness, I focus on the message-passing model, and discuss how it is supported on current machines. This serves as context for the remainder of the dissertation. Although the ideas in later chapters are discussed in the context of message-passing, they apply to other models as well.

2.1 Communication Issues

I begin by introducing the general issues that make it difficult to support communication efficiently. Rather than getting bogged down in details of a specific programming model or architectural model, this discussion will use a very high-level model. Detailed models will be discussed in later sections.

At a high level, a parallel program consists of a set of threads carrying out local computations. From time to time, one thread produces a value that another thread will use. Communication is simply the mechanism for getting such values from the producing thread to the threads that will use them. (I leave aside for now the issue of how the producer knows which threads will use the value.) A *protocol* is a strategy for managing communication.

Clearly, a value cannot be used until it has been produced, so communication implies a one-way synchronization between producer and consumer. The semantics of the program imply that the consumer will wait for the producer. Thus, any protocol must do at least two things: move the value from producer to consumer, and ensure that the consumer waits for the value to be available. The cost of doing these two things will be referred to as the *essential* cost of communication. Any additional communication cost will be termed *protocol overhead*.

2.1.1 The Simplest Protocol

The simplest protocol possible is to associate each datum being communicated with a unique piece of storage. When the consumer wants a piece of data, it simply waits until the associated memory contains a value, and then retrieves it. The producer communicates the value by putting it into storage, and then somehow causing that storage to migrate to the consumer.

Unfortunately, this protocol is not practical. The reason is that it requires a fresh piece of storage for each value that will be communicated. In a real system, a limited amount of storage is available. Therefore, a realistic protocol must either re-use storage, or avoid using it at all.

2.1.2 Dealing with Limited Storage

All realistic protocols suffer some protocol overhead. This overhead stems entirely from the fact that storage is limited. Managing storage is the central difficulty in designing a protocol.

There are two basic strategies for dealing with storage limits: re-using storage, and avoiding the use of storage altogether. Each has its own drawbacks, and each introduces its own kind of protocol overhead.

If storage is re-used, the protocol must do bookkeeping to keep track of which storage locations can be recycled. (A used location can be recycled after the consumer has retrieved its value.) In some cases, information about recyclable storage must be kept consistent between the distributed threads; this requires extra communication.

The real problem with re-using storage is that it ultimately cannot prevent the protocol from running out of storage. As the protocol runs, the amount of free storage fluctuates; if it ever reaches zero, the protocol cannot rely on storage re-use to continue. Thus, the storage re-use strategy is not sufficient by itself — the protocol must be prepared to avoid using storage when none is available.

To avoid using storage altogether, the protocol must use a different approach. The producer and consumer must rendezvous, and pass the value directly between them. The drawback of this scheme is that the producer might have to wait for the rendezvous. (This is in addition to the consumer waiting for the value to be produced, which is part of the essential cost.) The producer's waiting may harm performance, or worse, it might even lead to deadlock.

No matter which strategy is used, some overhead will result, inflating the cost of communication. Regardless of the architectural model, protocol overhead is unavoidable.

2.2 Parallel Programming Models

We now turn to the choice among real parallel programming models. There are many models to choose from, but no matter which one is chosen, we must face the same issues in implementing communication, and we must accept protocol overhead in some form.

2.2.1 Explicit vs. Implicit Communication

Traditionally, parallel machines have been divided into two classes: shared memory and message-passing. Shared memory machines supported a single, flat, coherent, shared address space in hardware, and were programmed by parallelizing Fortran compilers, or by a model using shared variables, threads, and monitors. Message-passing machines had physically distributed memory, did not support a single address space in hardware, and were programmed with processes and message-passing. Since there were two kinds of machines, and programming model was assumed to follow directly from architecture, there were also two programming models.

Three recent trends are making the shared-memory/message-passing dichotomy invalid. First, hardware designs are converging. The new generation of scalable shared-memory machines [Lenoski et al. 92, Agarwal et al. 91] have physically distributed memory and point-to-point communication networks. These machines look very much like the current generation of distributed-memory machines; the only difference is in how communication is caused. In the shared-memory machines, communication is an implicit result of references to the shared memory; the cache controller *hardware* sends messages between processors to carry out a cache coherence protocol. In the distributed-memory machines, messages are sent and received by *software*.

The second trend is toward increasingly sophisticated and varied parallel programming languages and models. These include data-parallel languages like High Performance Fortran [HPFF 93], C* [TMC 90], and Dino [Rosing 91]; shared virtual memory systems such as Ivy [Li & Hudak 86], Munin [Carter et al. 91] and Midway [Bershad et al. 93]; dataflow languages such as Id [Nikhil 88]; and stream-based systems like Strand [Foster & Taylor 90] and PCN [Foster et al. 92]. These systems have one thing in common: they provide the programmer with a single namespace for variables, but they run reasonably on distributed-memory machines. These systems demonstrate that the programmer can have the desirable features of shared memory, while still using a message-passing machine. The hardware design does not determine the programming model.

The third trend is the increasing realization that shared memory does not insulate the programmer from worrying about data placement and the cost of data movement. As CPUs become faster relative to the rest of the system, the cost of cache misses increases, and programs that run efficiently on older shared-memory

machines become inefficient on new shared-memory machines. Indeed, several researchers have observed that programs written in a message-passing style perform better on shared-memory machines than programs written in the "native" shared-memory style [LeBlanc 86, Byrd & Delagi 88, Lin & Snyder 90, Ngo & Snyder 92]. It is becoming increasingly clear that parallel programmers must face the same difficult issues whether they are programming shared-memory or distributed-memory machines [Snyder 86, Anderson & Snyder 91, Markatos & LeBlanc 92].

In view of these trends, we must develop a more sophisticated understanding of the tradeoff between shared memory and message-passing. The key issue underlying the shared-memory/message-passing debate is whether or not the programmer explicitly controls the layout and movement of data. Thus, the key tradeoff is between *explicit* and *implicit* data movement.

We must also recognize that modern implementations are built in layers: compiled code, on top of runtime library, on top of operating system, on top of hardware. Different layers may have different interfaces; layers close to the programmer offer more convenient interfaces, while layers far from the programmer offer interfaces that match the hardware well. Data movement must be explicit at the lowest level, since the hardware must ship bits around to implement communication and synchronization. If the programming language offers implicit data movement, some layer between the language and the hardware must translate implicit to explicit movement. How, and in what level, to perform this translation is one of the key questions in parallel computing.

One important consequence of the convergence of shared memory and message-passing systems is the recognition that in supporting parallel programs, we face many of the same issues regardless of the hardware platform. Increasingly, the structure of the application program, rather than the characteristics of the hardware, dictates the programmer's implementation strategy.

An Example

The work of Hatcher and Quinn on compiling data-parallel programs illustrates the complexity of implementation tradeoffs. Hatcher and Quinn developed two compilers that translate a program written in a language called Dataparallel C into code executable on a parallel machine [Hatcher & Quinn 91]. One compiler generates code for

a shared-memory machine [Quinn et al. 90], and one for a message-passing machine [Hatcher et al. 91].

The programmer's interface, Dataparallel C, is identical on both platforms. Dataparallel C provides the programmer with the abstraction of a single shared address space for variables. Thus, the programmer gets the sharing model that he wants.

The two compilers perform slightly different analysis and optimization, and the resulting executions on the two machines are quite different. Rather than simply emulating one machine on the other, Hatcher and Quinn chose the appropriate execution style for each machine.

On the shared-memory machine, the execution is carried out by a set of worker threads, one per processor. Program variables are kept in a single region of shared memory. Threads access these variables directly, and ensure proper synchronization by performing barrier synchronizations. The main task of the compiler is determining where these synchronizations should be placed.

On the message-passing machine, the execution is carried out by a set of worker processes with separate address spaces. Program variables are either kept in one process's address space, or replicated across all processes, at the compiler's discretion. The compiler inserts calls to a message-passing library to carry out the necessary communications. The compiler does not insert explicit synchronization operations, since synchronization happens implicitly due to messages. The main task of the compiler is determining where to place communication calls.

Depending on the structure of the application program and specifics of the machines' architectures, either the shared-memory machine or the message-passing machine might execute a particular program more efficiently. The programmer can use the same programming model, regardless of the underlying machine. The choice between machines has become an implementation issue, like the choice between sequential machines.

2.2.2 Other Models: Explicit Communication Without Message-Passing

Some programming models allow programmers to express explicit data movement without using message-passing. For example, the DINO language [Rosing et al. 90, Rosing 91] offers primitives that read and write remote data structures.

Similarly, the Split-C system [Culler et al. 93] extends local C-language program-

ming with global pointers and global arrays. Global pointers specify an arbitrary memory location, possibly located remotely. Global pointers are supported at the language level; dereferencing a global pointer causes the appropriate remote-memory fetch or store to be performed. Global arrays are simply arrays that are scattered among the memories of all processors. (Split-C has other interesting features that are not directly relevant to this discussion.)

Similar to Split-C is the MetaMP system [Otto & Wolfe 92]. MetaMP is currently implemented as an `awk`-based preprocessor, but future implementations are expected to use a full-fledged compiler [Otto 93].

Any of these three systems could be used to illustrate the issues raised by the remote-access programming model. This explanation is based on Split-C because it is the simplest of the three languages, and hence requires the smallest "explanatory overhead".

Programs written in Split-C are amenable to the same kind of analysis and optimization that Hatcher and Quinn used. To illustrate this, consider the case of a simple straight-line program written in Split-C, with remote-memory operations and barrier synchronizations. This program can be translated to message-passing form by carrying out the following steps:

1. Find all non-local data dependencies in the program: all values written by one process and later read by another.

2. Turn each data dependency into a message. The process that generates the value immediately sends the message, and the process that uses the value receives the message "just in time" when the value is needed. Each of these messages is given a unique tag, so the sends and receives match up properly.

3. Remove the barrier synchronizations from the program, since they are no longer needed to ensure correctness.

The resulting program uses pure message-passing, rather than shared-memory-style operations. Note that, due to its use of "just in time" receives, the message-passing model handles anti-dependencies and output-dependencies naturally, without any special effort.

Process 0	Process 1
x[0] = f(0)	x[1] = f(1)
Barrier	Barrier
y[0] = x[1]	y[1] = x[0]
z[0] = x[0]+y[0]	z[1] = x[1]+y[1]

Figure 2.1: A data-parallel shared-memory program, written in a notation similar to Split-C.

Process 0	Process 1
x[0] = f(0)	x[1] = f(1)
send x[0]	send x[1]
receive t_0	receive t_1
z[0] = x[0]+t_0	z[1] = x[1]+t_1

Figure 2.2: The program of figure 2.1, translated to message-passing form.

Consider the pseudo-Split-C program shown in figure 2.1. This program has two nonlocal dependencies, carried across the barrier by the variables `x[0]` and `x[1]`. After finding these dependencies, the compiler can transform the program into the equivalent form shown in figure 2.2. Each dependency has been turned into a message; this makes the barrier redundant so it is removed.

At this point we may ask whether the message-passing version of the program is more or less efficient than the obvious remote-access implementation of the Split-C program. On a shared-memory machine the shared-memory version will probably run faster. However, on a distributed-memory machine, *any* implementation must use messages to transmit the remotely-referenced data. The only question is what protocol will be used to ensure that the messages satisfy the dependencies — that the correct version of the shared variables are read or written at each point in the program.

One possible protocol is the straightforward shared-memory-style implementation, in which reading a shared variable causes an asynchronous fetch of remote data, and proper synchronization is ensured by executing a barrier synchronization. However, other protocols exist, and may be more efficient in specific cases. Ideally, the best protocol will be chosen for each particular situation.

The same implementation issues arise, whether message-passing or shared memory is the initial programming model. In both cases, one can determine the necessary data movement and synchronization constraints, and then choose a protocol to efficiently satisfy them.

2.3 The Inevitability of Protocol Overhead

Protocol overhead is not confined to any one programming model; it affects all programming models. Protocol overhead cannot be avoided; it is a fact of life for parallel programmers. We can hope only to reduce its effect.

The following subsections consider three common parallel programming models, and point out the protocol overheads in one typical implementation of each model. I will assume that the reader is generally familiar with each system. Section 2.4.2 discusses protocol overhead in message-passing programs.

2.3.1 Protocol Overhead in Cache-Coherent Shared-Memory Systems

The first example is the Stanford DASH multiprocessor [Lenoski et al. 92] running a program that uses shared variables and locks. The DASH has physically distributed memory for scalability; processors' views of memory are kept coherent by a directory-based cache coherence protocol. (Although this discussion is based on the DASH, it applies to other shared-memory architectures, including bus-based machines.)

Suppose some memory location is cached by two processors A and B. A writes the memory location, producing a value. Later, the value is consumed by B when it reads the memory location.

In this case, the only essential communication is the transmission of the written value from A to B. In reality, much more communication takes place. When A executes its write operation, it sends an invalidation message to B; this ensures that A has the only valid copy of the location. B acknowledges the invalidation, and A then writes the location. Later, when B reads the location, it finds the location invalid in its cache, so it sends a request message to A. Finally, A carries out the essential communication by sending the new value of the location to B[1].

Along with one message of essential communication came three messages of protocol overhead. Although one can attempt to hide the latency of this overhead by using prefetching, this imposes new costs and does not change the fact that the extra communication uses hardware resources. Furthermore, prefetching appears to be less useful in parallel programs than was once believed [Tullsen & Eggers 93].

There is another source of protocol overhead hidden in the shared-memory implementation. The program as described contains a data race, since there is no guarantee that B actually reads the version written by A. Because the processors are asynchronous, the read might happen before the write. To avoid this, some synchronization must take place between A's write and B's read. This synchronization imposes additional communication, which also counts as protocol overhead. Indeed, synchronization primitives can require a great deal of communication [Anderson et al. 89, Mellor-Crummey & Scott 91].

At first glance, it might appear unfair to classify synchronization as overhead.

---

[1] Because DASH uses release consistency [Gharachorloo et al. 90], some of these communications might actually happen in parallel. However, even under release consistency, the latency and bandwidth required are more than the essential communication alone demands.

However, synchronization is really just a way of compensating for the fact that the same memory location is being used to store logically distinct values. By storing multiple "versions" of each memory location, we could in principle avoid the need to make the sender wait for the receiver [Feeley & Levy 92].

2.3.2 Protocol Overhead in Remote-Access NUMA Systems

The second example is a shared-memory system without caching. This is a system in which each memory location is located with some processor, and processors can access each other's memory directly. Examples of such NUMA (non-uniform memory access time) machines include the BBN Butterfly, the Cray T3D, and the CM-5 running Active Messages.

This case is very similar to the case of cache-coherent shared memory. As in the cache-coherent case, NUMA programs must synchronize to ensure correct execution; this synchronization imposes some overhead. Although NUMA systems do not experience the overhead of a cache coherence protocol, they do suffer from the cost of making repeated references to remote data, rather than caching them locally.

2.3.3 Protocol Overhead in Linda

The third example is a multicomputer like the Intel DELTA, running the Linda coordination language [Carriero & Gelernter 89, Gelernter & Carriero 92]. In the Linda model, all communication is through a logically shared data structure called the tuple space. Processes add tuples to the tuple space via the `out` primitive, and access and remove existing tuples via the `in` and `rd` primitives.

The main efficiency problem of Linda is that tuples are produced and consumed anonymously, so they cannot be directed to a particular destination but must instead be globally accessible. Many clever compile-time and run-time techniques have been developed to reduce this cost, but it remains significant.

Even an efficient Linda implementation requires three packets to get a tuple from producer to consumer. First the producer executes its `out` operation, using a packet to put the tuple into the tuple space. Then the consumer executes its `in` operation, sending a request packet, and then receiving a packet containing the body of the tuple.

Summary

Considering these examples, we see that protocol overhead is a property of all parallel programs. Protocol overhead stems from the need to reuse memory locations to store data at different times during execution. The remainder of this dissertation will consider the problem of how to reduce protocol overhead for data-parallel message-passing programs. Similar techniques can be used to reduce protocol overhead in other models.

2.4 Message-Passing

To make the discussion concrete, we will now focus on the implementation of one model, message passing. The remainder of the dissertation will discuss how protocol overhead arises in message-passing systems and what can be done to reduce it. The lessons learned in this effort can be applied to reduce protocol overhead in other programming models, as well.

This section discusses message-passing, particularly how it is supported by hardware and software on multicomputers. Message-passing is a common way of managing communication and coordination between processes in parallel programs. Traditionally, message-passing has been used mainly on distributed-memory machines, because it matches the capabilities of the hardware.

Most multicomputers use a two-level strategy to support message-passing. The hardware provides communication capabilities to a software library; this library uses the hardware to provide higher-level communication services to the application program. The bulk of this section is about the software layer, since that is where the vast majority of time is spent. However, I begin with a brief introduction to multicomputer interconnection hardware.

2.4.1 Message-Passing Hardware

Today's multicomputers are built as a collection of processing nodes connected by an interconnection network. Each node is typically a commodity microprocessor with a moderate amount of memory, although multiprocessor nodes are sometimes used. I/O devices are often connected directly to certain processing nodes. Some machines have a special "front-end" processor that is used for system administration.

Typical interconnection networks are exhibited by Intel's DELTA [Intel 91b] and Paragon [Intel 91a] systems. In these systems, the nodes are connected in a two-dimensional grid topology. This topology offers a high bisection bandwidth and presents fewer packaging problems than other organizations [Dally 87]. Messages are routed using a non-adaptive, oblivious algorithm: a message first moves horizontally until it reaches the correct column, then it moves up or down to the destination [Sullivan & Brashkow 77].

The Intel network is implemented as a set of router chips connected directly by wires. The routers are pipelined and use wormhole routing [Dally & Seitz 87] to rapidly set up a path for a message. The per-hop latency of these networks is so low that we can consider the interconnection topology irrelevant, except for its effect on contention for links [Bokhari 90]. The Intel Paragon's network can set up a path between any two nodes in less than one microsecond; once established, the path has a bandwidth of 200 megabytes per second.

A different network design is used by the Thinking Machines CM-5 [TMC 91, Leiserson et al. 92]. The CM-5 network uses packet switching to deliver fixed-size packets of twenty bytes. Packets are routed adaptively to avoid blocking at hot-spots in the network. The network topology is a fat-tree [Leiserson 85]: a tree whose links have increasing capacity near the root. The precise shape of the fat-tree is chosen to match the levels of packaging in the machine: board, rack, cabinet. The CM-5 network has very low latency, like the Intel network, but the CM-5 can sustain at most 20 megabytes per second between nodes, a factor of ten less than Intel's top speed. The CM-5's fat-tree topology *guarantees* a bandwidth of at least 5 megabytes per second between any pair of nodes.

All multicomputer networks are reliable — they deliver every injected packet. Corrupted messages are detected by checksums added to the end of each message by the hardware. Most systems simply report an error when a message is corrupted; either the application itself must recover from the error, or some kind of checkpoint/restart mechanism can be used to recover. However, such errors are so rare that they do not pose a problem in practice.

2.4.2 Message-Passing Software

There are several software packages that offer communication services to applications. In some cases, these packages are part of the operating system kernel; Intel's NX [Pierce 88] and NCUBE's Vertex [NCUBE 87] are examples. Other packages are implemented as user-level libraries; these include Thinking Machines' CMMD [TMC 92], Parasoft's Express [Parasoft 88], and the public-domain packages PVM [Sunderam 92], PICL [Geist et al. 91], and Zipcode [Skjellum & Leung 90]. These packages have slightly different purposes and features, but in most ways they are very similar.

In response to the proliferation of message-passing packages, a standardization effort was launched. The proposed standard, called MPI1, is meant to capture the established features common between the competing message-passing packages. Because the MPI1 standard is still under discussion, and several proposed versions are circulating, I will not attempt to describe it here. Rather, I will base the discussion on a composite of today's competing systems.

Interface and Semantics

This subsection explains the interface provided by a typical message-passing package. Rather than explaining the quirks of any particular system, I have made a kind of composite of the basic features of systems like NX, Express, and PICL. I'll call this imaginary system MP0. Although MP0 does not offer as many features as the other systems, it has the advantage of being simple, yet addressing all the important issues in the design of message-passing systems. The semantics of most message-passing systems are not specified precisely in writing, so I have given MP0 the "consensus folklore" semantics that most programmers expect.

An MP0 job consists of a set of processes distributed over some set of processing nodes. There is usually one process per physical processor, although most multicomputer operating systems support multiple processes, and systems like NewThreads [Felten & McNamee 92] support multiple threads of control within a process. Processes communicate by making calls to the MP0 communication library.

At the simplest level, MP0 supports point-to-point transmission of messages between processes. A message consists of data of arbitrary size. Each message is marked with a *tag*. A process issuing a `receive` operation supplies a *tag specifier*: a predicate

that says which tags may satisfy the receive. (In most real systems, the tag-specifer must either accept all possible tags, or accept one specific tag.) Messages are sent and received directly from locations in the memory of the application processes.

Abstractly, there are two kinds of communication operations in MP0, `send` and `receive`. Each operation happens in two stages: first it is *issued* by an application process, and later it *completes*.

For example, issuing a `send` operation informs MP0 that the message data is available to be sent from the sender's buffer; therefore MP0 may transmit the data. A `send` operation completes when the message's data has been extracted from the sender's memory; completion of a `send` indicates that the sender is free to reuse the buffer. Note that completion of a `send` does not imply that the message has been delivered to the receiver; thus MP0 semantics imply that messages may be buffered.

Issuing a `receive` operation informs MP0 that the receiver is ready for the message's data to arrive in the receiver's memory. A receive operation completes when the message's data has actually been placed in the receiver's memory.

Blocking and Nonblocking Communication

There are two modes of communication: blocking and nonblocking. In the blocking mode, a process initiates a `send` or `receive` operation, and immediately waits for the operation to complete. In the nonblocking mode, a process makes an MP0 call to issue a `send` or `receive` operation. The process may then do other work before making another MP0 call to wait for completion of the operation. Note that nonblocking mode allows a process to have several communication operations in progress at the same time.

Since blocking calls can be implemented trivially in terms of non-blocking calls, I will omit the blocking calls from now on. The MP0 interface treats `send(`*args*`)` as syntactic sugar for `endSend(beginSend(`*args*`))`, and similarly for `receive`.

Figure 2.3 lists the basic non-blocking calls supported by the MP0 interface.

Buffering and Deadlock

As noted above, since completion of a `send` operation does not imply delivery of the message to the receiver, MP0 semantics allow the implementation to buffer messages. Indeed, programmers *expect* MP0 to do buffering.

```
int beginSend(          /* send a message, nonblocking */
    int tag,            /* message tag */
    void* buffer,       /* memory buffer to send from */
    int length,         /* number of bytes to send */
    int dest            /* process-ID to send to */
);                      /* returns message ID for endSend */

int beginRecv(          /* receive a message, nonblocking */
    int (*tagSpec)(),   /* select by tag */
    void* buffer,       /* memory buffer to receive into */
    int maxLength       /* maximum number of bytes to receive */
);                      /* returns message ID for endRecv */

int endSend(            /* wait for a send to finish */
    int msgID           /* which call is it? */
);                      /* returns number of bytes sent */

int endRecv(            /* wait for a receive to finish */
    int msgID           /* which call is it? */
);                      /* returns number of bytes received */
```

Figure 2.3: Basic MP0 communication calls.

Process P_0	Process P_1
`send to` P_1`, tag 0`	`send to` P_0`, tag 1`
`receive, tag 1`	`receive, tag 0`

Figure 2.4: A program that requires buffering.

Some programs require buffering to execute correctly. For example, the SPMD program in Figure 2.4, in which two processes exchange messages, requires that the implementation buffer at least one of the two messages. In general, a program may require an arbitrarily large amount of buffering to run correctly.

In practice, the implementation has only a limited amount of buffer space. Unfortunately, the desire to run ordinary programs efficiently conflicts with the desire to avoid deadlock in all possible cases. Message-passing implementations take a "best effort" approach to avoiding deadlock: they attempt to avoid deadlock whenever they can, but they do not guarantee to avoid deadlock in all cases.

Implementation of a Message-Passing System

Having introduced some of the issues surrounding the design of a message-passing system, we now turn to the implementation of a specific set of systems. Since MP0 is a fictional composite of several popular message-passing systems, we cannot discuss the implementation of MP0 directly. Rather, this section sketches how Intel's NX, a typical message-passing system, is implemented on the iPSC/860 and Delta systems[2]. An MP0 implementation on the same machines would use a similar strategy.

Both the iPSC/860 and the DELTA have well-behaved networks that are guaranteed to deliver all messages correctly, with the order of messages between a particular sender and receiver preserved. Thus, the main difficulty in protocol design is buffering. We would like the message-passing system to buffer messages, both to avoid deadlock and to remove the necessity of strict synchronization between the sending and receiving processes. Since buffer space is limited, the protocol must somehow ensure that messages are not lost due to a processor running out of buffer space. Rather than using a retry-based approach, the system uses software flow control — it ensures that no packet is sent to a processor unless the processor is guaranteed to have sufficient buffer space to store that packet.

I describe here a family of multicomputer protocols that use software flow control. Members of this family use the same basic data structures, but vary in which policies they use to manage those structures. After presenting the common framework shared by the family of protocols, I describe two specific protocols from the family. These

[2] This information is gleaned from folklore and discussions with an NX implementor. The NX source code is protected by a non-disclosure agreement.

are the protocols used in NX implementations.

A Family of Multicomputer Protocols

Although message-passing programs are written in terms of processes, protocols are implemented in terms of processors. Communication and synchronization between multiple processes on the same processor are easy to manage, so they are ignored when describing protocols.

Each processor allots a fixed region of memory for message buffers, and divides that space into fixed-sized packet buffers. All messages are packetized, so buffers are allocated at a packet granularity. Processor i divides its free buffers into p pools (where p is the number of processors), with one pool dedicated to packets sent by each other processor, and the remaining pool containing packets not yet committed to any sender. The sizes of these pools are maintained in variables by processor i: $rbufs(s,i)$ is the number of free buffers dedicated to packets sent by processor s, and $ubufs(i)$ is the number of free but uncommitted buffers. At all times, processor i can receive up to $rbufs(s,i)$ more packets sent by processor s, without running out of buffer space. $rbufs(s,i)$ is changed as necessary to maintain this invariant.

Each sending processor s must therefore avoid having more than $rbufs(s,j)$ outstanding packets destined for processor j. However, since s cannot read the memory of j directly, s does not know the value of $rbufs(s,j)$. Instead, s maintains its own approximation $sbufs(s,j)$, which obeys the invariant $sbufs(s,j) \leq rbufs(s,j)$. Every time s sends a packet to j, s must decrement $sbufs(s,j)$ to reflect the fact that the packet might be using a buffer at the destination j. The transmission of data, then, reduces the various entries in *sbufs*. The task of the protocol is to increase the entries in *sbufs* without violating the invariants; this allows further communication to take place. Protocols in this family differ by precisely how they safely increment *sbufs*.

A Pre-reservation Protocol

In the first protocol, we start with $rbufs(i,j) = sbufs(i,j) = 0$ for all $0 \leq i,j < p$, and all buffers allocated to *ubufs*. When processor s wants to send an n-packet message to processor r, s first sends a $request(n)$ packet to r[3]. When r receives the request

[3] Throughout this protocol, and in subsequent protocols, we assume that *request* and *grant* packets are dealt with specially by the interrupt handler. Since these packets can always be consumed

packet, if $ubufs(r) \geq n$, then r can accept the message: r subtracts n from $ubufs(r)$, adds n to $rbufs(s,r)$, and sends a $grant(n)$ packet to s. These actions prepare a set of n buffers to receive the message, and inform s of this fact. When the $grant(n)$ packet reaches s, s adds n to $sbufs(s,r)$; now s can send the n-packet message.

If, when the $request(n)$ message arrives at r, $ubufs(r) < n$, then r does not send a *grant* packet, but simply records the fact that processor s requested n buffers. Later, when $ubufs(n)$ reaches at least n, the request can be granted as described above. During this interval, the send operation is unable to complete.

Figure 2.5 shows the history of a particular user-level send/receive pair under this protocol. When the sender is ready, it transmits a *request* packet, asking for permission to transmit the message body. When the receiver gets the *request*, it finds buffer space for the message, then transmits a *grant* packet, giving the sender permission to transmit the message body. Upon receiving the *grant* packet, the sender transmits the message body. The message body then sits in a buffer on the destination processor until the receiving process is ready for it. This protocol is efficient for large messages since the request/grant overhead must be paid only once per message.

A Sliding-Window Protocol

The previous protocol allocates space to *rbufs* lazily. An alternative is to allocate space more eagerly, in the hope of reducing the latency that occurs before any data can be passed. This leads to the use of sliding-window flow control.

This protocol has two parameters b_{high} and b_{low}, with $b_{high} > b_{low} \geq 0$. It starts with all $rbufs(i,j) = sbufs(i,j) = b_{high}$, and all the remaining buffers in *ubufs*. When some $rbufs(i,j) \leq b_{low}$, the protocol moves $\min(b_{high} - rbufs(i,j), ubufs(j))$ buffers from $ubufs(j)$ to $rbufs(i,j)$ in an attempt to keep $rbufs(i,j)$ "fully stocked" with buffers. When $rbufs(i,j)$ is increased, a *grant* packet is sent to processor i to inform it that buffers have been added.

This protocol is optimized for small messages. It has the advantage that a message of length b_{low} or shorter can usually be sent immediately without having to handshake first. In other words, the *grant* packets are usually not on the critical path for short messages. (There are no *request* packets in this protocol.) The disadvantage is that

---

immediately, they do not need any buffers.

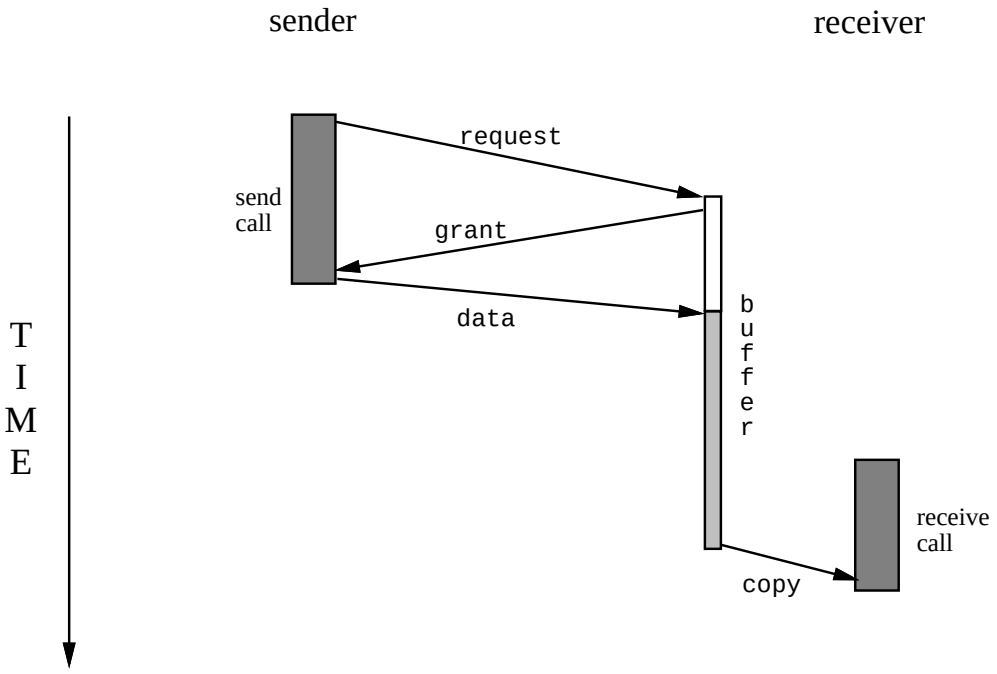

Figure 2.5: The pre-reservation protocol.

for large messages, more grant messages are sent than in the previous protocol; this consumes more bandwidth and hence reduces the performance of large messages.

An Example: The NX Protocols

Intel's NX systems use either the pre-reservation protocol or the sliding window protocol, depending on circumstances. NX/2, which runs on the iPSC/2 and iPSC/860 systems, uses sliding-window for messages less than 100 bytes in length, and pre-reservation for larger messages. NX/M, which runs on the Touchstone DELTA, uses sliding-window for all messages.

Protocol Overhead in Multicomputer Protocols

It is tempting but incorrect to say that the programmer sends messages, and the hardware transmits messages, so one ought to be able to implement message-passing programs with no protocol overhead. This fallacy stems from sloppy use of language: using the word "message" to mean two distinct things. The messages expressed by the programmer and the messages transmitted by the hardware are different things; they have different semantics and hence do not map one-to-one onto each other. For clarity, I will refer to the programmer's constructs as "messages" and the units sent by the hardware as "packets".

There are two main sources of protocol overhead in our family of multicomputer protocols. First, there is the cost of processing the *request* and *grant* packets. This cost arises because of the processors' ignorance of each other's state. The *request* messages of the pre-reservation protocol overcome the receiver's ignorance of the fact that the sender wants to transmit a message, and the *grant* packets of both protocols overcome the sender's ignorance of how many buffers the receiver has available.

The second overhead in these protocols is due to the cost of buffering. Rather than moving the contents of a message directly from the sender's address space to the receiver's address space, message-passing software often chooses to store it temporarily in an intermediate buffer. There are two reasons to do this. First, buffering decouples the sender from the receiver; the sender can continue with its program without having to wait for the receiver to ask for the message. Second, buffering is sometimes necessary to avoid deadlock. Although buffering is sometimes a good idea, it also consumes resources. Buffering requires time, to copy the buffered data one

extra time, and space, in which to store the message. The cost of buffering is counted as protocol overhead.

2.5 Summary

Protocol overhead is an unavoidable attribute of parallel programs, arising from the fact that storage is limited. Protocol overhead takes different forms in different programming models, but is present in all models.

Chapter 3

IMPROVING MESSAGE-PASSING PROTOCOLS

I cannot do it without computers.
— *Shakespeare,* The Winter's Tale

Traditional multicomputers use general-purpose protocols — protocols that will work regardless of what the application program does. The pre-reservation and sliding-window protocols of section 2.4.2 are examples of general-purpose multicomputer protocols. General-purpose protocols function correctly, but their performance can be disappointing. As an alternative, I am proposing the use of *tailored protocols*, which are specially designed to work with a particular application program.

We often have some kind of advance knowledge about the communication behavior of an application program. For example, we may know that a certain phase of the program carries out a Fast Fourier Transform (FFT); since the communication pattern induced by the FFT computation is fixed, we can precisely predict the application's communication behavior during that phase. Perhaps in some other phase of the application, the processes pair up and exchange messages with their partners. And perhaps in yet another phase, all communication consists of remote procedure calls to some master node, obeying a request/response pattern.

Whatever knowledge is available in advance about the application's communication behavior can be used to design a tailored protocol. The design of this protocol can take advantage of what is known about what the application will do and, more importantly, what it will not do.

For example, suppose that at some point in the application program, process S sends a message to process R, and we can deduce in advance that when S is ready to send, R will definitely have sufficient buffer space to store the message. In this case, we can eliminate the *request/grant* transaction from the standard pre-reservation protocol of section 2.4.2; there is no need for S to ask R whether it has buffer space.

Similarly, we can use the same knowledge to eliminate the *grant* message in the sliding-window protocol. In either case, we are using advance knowledge to reduce S's ignorance about R's state, and hence we are reducing the overhead caused by that ignorance.

There are many other ways to use advance knowledge to streamline the protocol. Generally speaking, we would like to make as many decisions as possible in advance, thereby reducing the amount of work that must be done at run-time. At the extreme, a carefully crafted protocol would be devised which treats each individual message in a specialized way.

3.1 Design Choices

There are many choices open to the designer of a tailored protocol. Any policy embedded in the standard protocol may be reconsidered. Among the design choices available are:

- which messages will be buffered, and which will not;
- where messages will be buffered in memory;
- whether or not to break each message into packets, and how large to make the packets;
- whether to combine two or more messages into a single data transfer;
- whether to receive a message via interrupts or via polling; and
- how to route each message to the destination (assuming the architecture allows this choice).

In principle, all these choices could be left open. In practice, these choices differ widely both in importance and in ease of implementation. The work described in this dissertation will focus on the first two choices, since these two have the greatest effect on performance.

3.2 Designing Tailored Protocols by Hand

This section will give some examples of how one might design a tailored protocol by hand. The purpose is not to explain in detail all of the issues that arise in designing such a protocol — that is the topic of later chapters. The goal of this section is to provide a rough feel for some of the issues that arise in designing tailored protocols, and for the difficulty of doing so by hand.

To simplify the discussion, we will assume for now that tailored protocols are built using a `remoteWrite` operation supported by the hardware. This operation allows a process to directly write a block of data into the memory of another process[1]. The call `remoteWrite(destProc, destVar, val)` copies the local value `val` into the variable `destVar` in remote process `destProc`. There is no remote-read operation.

3.2.1 A Simple Example

Process P_0	Process P_1
$v \leftarrow f(2)$	$w \leftarrow f(3)$
$x \leftarrow f(v)$	$y \leftarrow f(w)$
beginSend x to P_1 tag 13 name fred	beginSend y to P_0 tag 14 name martha
beginRecv v tag 14 name foo	beginRecv w tag 13 name bar
endSend name fred	endSend name martha
endRecv name foo	endRecv name bar
$z \leftarrow g(x, v)$	$z \leftarrow h(y, w)$

Figure 3.1: An example program fragment.

Figure 3.1 shows a simple program fragment. The reader is invited to consider how to implement this program using `remoteWrite`. Despite the simplicity of this program, there are at least three plausible tailored protocols for it.

Figure 3.2 shows one possibility, in which it has been decided not to buffer either message. This protocol implements each message as a single `remoteWrite` operation.

[1] The remote-write operation is similar to the *force-type* messages of Intel's NX operating system. I chose to base this discussion on remote-write because it is easier to explain and reason about.

Process P_0	Process P_1
$v \leftarrow f(2)$	$w \leftarrow f(3)$
$x \leftarrow f(v)$	$y \leftarrow f(w)$
barrierSynch()	barrierSynch()
remoteWrite(P_1, w, x)	remoteWrite(P_0, v, y)
barrierSynch()	barrierSynch()
$z \leftarrow g(x, v)$	$z \leftarrow h(y, w)$

Figure 3.2: A tailored protocol for the program of figure 3.1.

The `remoteWrite`s are sandwiched between a pair of barrier synchronizations. The first barrier ensures that the remote values are not written too early, that is, before the receiving process is finished with its previous use of the variable v or w. The second barrier makes sure that each process has received its incoming value before proceeding. (Barrier synchronization is implemented by having each process use `remoteWrite` to set a bit that the other process reads.)

Process P_0	Process P_1
$v \leftarrow f(2)$	$w \leftarrow f(3)$
$x \leftarrow f(v)$	$y \leftarrow f(w)$
remoteWrite(P_1, ρ_1, x)	remoteWrite(P_0, ρ_0, y)
barrierSynch()	barrierSynch()
$v \leftarrow \rho_0$	$w \leftarrow \rho_1$
$z \leftarrow g(x, v)$	$z \leftarrow h(y, w)$

Figure 3.3: Another tailored protocol for the program of figure 3.1.

Figure 3.3 shows another possibility, in which both messages are buffered. By introducing temporary storage locations (i.e. buffers) ρ_0 and ρ_1, we have dispensed with the first barrier synchronization. This technique of trading time for space is a common theme in the design of tailored protocols. Adding temporary storage is a

good idea if the amount of storage needed is small, but if v, w, x, y, and z are large arrays, then the extra storage might not be available, and the cost of copying data out of ρ_0 and ρ_1 might be too large.

Process P_0	Process P_1
$v \leftarrow f(2)$	$w \leftarrow f(3)$
$x \leftarrow f(v)$	$y \leftarrow f(w)$
remoteWrite(P_1, ρ_1, x)	wait until $b_1 = 1$
remoteWrite$(P_1, b_1, 1)$	$b_1 = 0$
wait until $b_0 = 1$	remoteWrite(P_0, v, y)
$b_0 \leftarrow 0$	remoteWrite$(P_0, b_0, 1)$
$z \leftarrow g(x, v)$	$w \leftarrow \rho_1$
	$z \leftarrow h(y, w)$

Figure 3.4: Another tailored protocol for the program of figure 3.1.

Figure 3.4 shows yet another possibility, in which one of the two messages is buffered and the other is not. Here the processes interact in an asymmetric fashion. There are two one-way synchronizations in which one process signals the other using one of the bits b_0 or b_1; we can think of these synchronizations as the two halves of a barrier synchronization, broken apart. This protocol has two possible advantages over the previous one. First, it requires less temporary storage: only P_1 requires extra storage. Second, this protocol performs half-duplex rather than full-duplex communication; half-duplex is more efficient on some architectures, such as the Intel DELTA. A possible disadvantage of this protocol is that it tends to cause process P_1 to finish the computation sooner than P_0; if this program fragment is embedded in a larger program, this may cause other parts of the program to suffer from load imbalance.

Note that the three alternative protocols apply only to the program fragment shown. If this fragment is part of a larger program, other parts of the program use either the standard protocol, or a separately-generated tailored protocol.

How should we choose among these three possible tailored protocols? There are two issues involved. First, the choice depends on whether we can afford to use extra

storage. Second, we want to choose the protocol that offers the best performance. Evaluating the performance of the three protocols depends on having an accurate model of the costs and latencies of various operations on the target architecture.

This example has shown that even for very simple application programs, designing a tailored protocol is not a simple task. There are many alternatives; choosing among them requires detailed performance modeling and evaluation of time/space tradeoffs.

3.2.2 Extending the Example

Now let us imagine that the program fragment we have been analyzing is actually the body of a loop, so it is executed several times in sequence. This makes our second tailored protocol incorrect — for example, if P_0 falls behind P_1 for some reason, P_1 might `remoteWrite` twice into ρ_0 before P_0 has a chance to read it.

This flaw can be fixed by allocating twice as much extra storage. If each process has *two* units of extra storage, the protocol can `remoteWrite` into one unit in the odd-numbered iterations, and into the other unit in the even-numbered iterations.

But this may not be necessary. If the function g performs some communication, it may, as a side-effect, provide enough synchronization to fix our second protocol. Of course, this might depend on which tailored protocol we choose to use within g.

This illustrates another feature of the protocol design problem: choices made at one place in the program affect the correctness and performance of other sections of the program.

These examples give some idea of how difficult it is to design and verify a tailored protocol. The reader should bear in mind, however, that we have looked only at artificially simple situations. The performance tradeoffs and correctness issues that arise in realistic programs are much more difficult.

3.3 Protocol Compilers

We have seen that tailored protocols offer the hope of a large performance advantage over the standard approaches to message-passing. Unfortunately, designing protocols by hand is too difficult to be practical.

The solution to this problem is to use a *protocol compiler*: a software tool that *automatically* designs tailored protocols. Automating the design process allows us to

have the best of both worlds: the performance benefits of tailored protocols, without imposing more work on the programmer.

There are several properties we would like from an ideal protocol compiler.

- It doesn't cause correct programs to stop working. In other words, any program that runs under a standard message-passing system should run under the protocol compiler as well.

- It provides a performance benefit over standard approaches. We would like the benefit to be as large as possible on the average. In addition, we'd like the protocol compiler to never make any program run slower.

- It generates tailored protocols that accept the same interface as the standard message-passing library. This provides the programmer with familiar semantics, and allows him to use the protocol compiler on an existing program without changing that program at all.

- It allows, but does not require, the user to provide hints that help the protocol compiler generate better protocols.

While it may not be possible to satisfy this wish-list completely in the first implementation, the list provides a set of goals to strive for.

To understand the role of protocol compilers in message-passing programs, we can use the analogy of an ordinary compiler. The traditional approach to message-passing is like an interpreter, which makes all of its decisions dynamically, at some cost to performance. We would like to replace this interpreter with a compiler — a tool that analyzes the input program in advance in order to reduce run-time costs. Of course, the compiler cannot do *everything* in advance; some tasks, like dynamic memory allocation, must be left to run-time. The compiler generates calls to a run-time library to carry out these tasks.

Building a compiler requires an understanding of how the behavior of programs can be represented and analyzed. In the same way, to build a protocol compiler we must have techniques for analyzing the behavior of message-passing programs, and for transforming message-passing protocols into equivalent, but faster, protocols.

It may not be obvious that it is possible to build a protocol compiler that improves the performance of real application programs. This dissertation demonstrates by example that this goal can be reached.

3.4 Summary

One way to reduce protocol overhead is to use tailored protocols, which are specialized to the behavior of a particular application program. It is possible to design a tailored protocol by hand, but this is a difficult and error-prone process. To make tailored protocols practical, we must automate their design. A protocol compiler is an automatic tool that fulfills this requirement.

Chapter 4

COMPILING TAILORED PROTOCOLS

A good pattern can keep your new garment from looking like you tailored it yourself.

— *Anonymous Sewing Teacher*

The previous chapter showed the benefits of using tailored protocols, but also pointed out the difficulty of designing these protocols by hand. A protocol compiler is required to make the use of tailored protocols practical.

Figure 4.1 shows how a protocol compiler is used. The protocol compiler takes as input some description of the application program's communication behavior; this description could be the source code of the application program, or some condensed representation of its communication behavior. The protocol compiler analyzes the source program's behavior, and designs a tailored protocol. The protocol compiler's output is a program fragment that implements the tailored protocol underneath the system's standard communication interface. This program fragment is then compiled and linked with the application program and a special runtime library.

From the application program's point of view, the tailored protocol looks exactly like the machine's native communication mechanism. This means that, beyond being a functional replacement for the native mechanism, the tailored protocol must offer the same interface as the native mechanism. For example, on an Intel iPSC/860, the tailored protocol must offer the same interface as the native NX/2 communication library. Thus, the application program can be unaware of the fact that it is running on top of a tailored protocol.

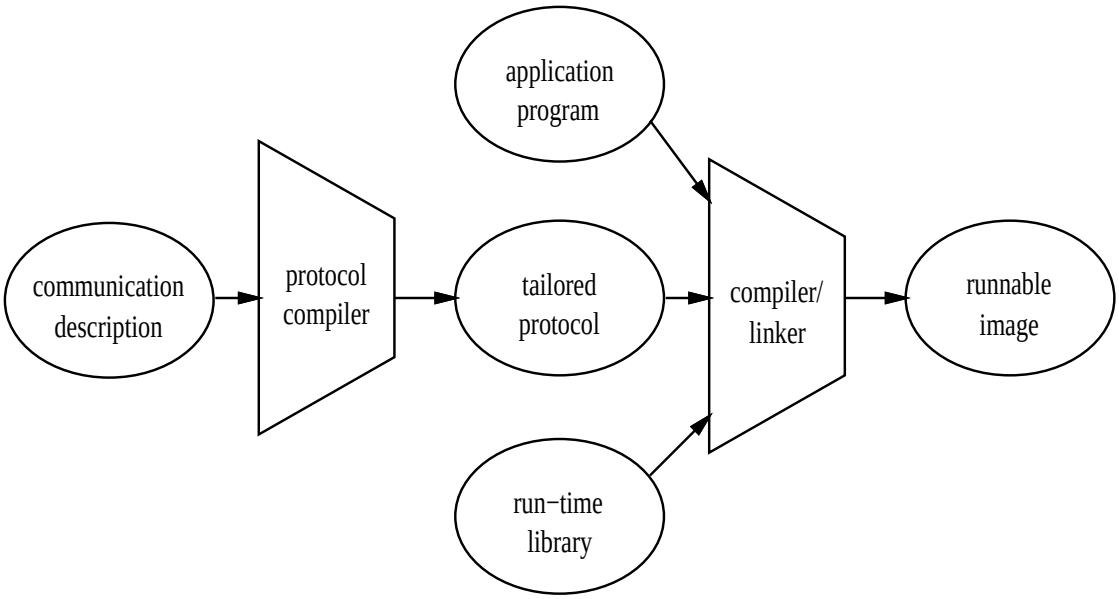

Figure 4.1: This diagram shows how a protocol compiler is used. The protocol compiler takes as input the communication description, and produces as output code for a tailored protocol. This is then compiled and linked with the application program and a special runtime library to yield a runnable image.

4.1 Designing a Protocol Compiler

In principle, protocol compilers might be designed for a wide range of input programs, and a wide range of target architectures. To design a protocol compiler, one must deal with several issues:

1. how to represent the communication behavior of a program, and how to extract that information from the program;

2. how to support tailored protocols on the target architecture; and

3. what kinds of analysis the protocol compiler should carry out in order to map from the program's behavior to a tailored protocol.

We would like the answers to these questions to be as general as possible, so they are of the greatest possible use.

In this chapter and the next two, I address these questions of protocol compiler design. Along the way, I describe the details of Parachute, a prototype I have designed and built. Throughout this discussion, it is important to bear in mind the distinction between what is general, applying to all protocol compilers, and what is specific to Parachute.

Parachute is specialized in three ways. First, it applies only to a certain class of data-parallel application programs. Second, it generates code only for message-passing multicomputers running Intel's NX operating system. Third, it makes particular choices about how to attack certain combinatorial optimization problems encountered during its analysis. (While specific choices had to be made in each of these dimensions, it would be relatively simple to adopt another specific set of choices in a different implementation.)

The remainder of this chapter deals with the first of the three main issues: how to represent a program's communication behavior. Chapter 5 discusses the analysis carried out by the protocol compiler, and chapter 6 contains additional details about Parachute.

Before discussing how to represent programs, I first introduce data-parallel programs. Since Parachute is restricted to this class of programs, it is important to understand the structure and importance of data-parallel programs and languages.

4.2 Data-Parallel Programs and Languages

Data-parallel programs are an important class of parallel programs. The relative simplicity of the data-parallel model has enabled the development of programming and debugging tools, and programming languages, that efficiently support data-parallel programming. In addition, the majority of successful scientific applications of parallel computers are amenable to data-parallel implementation. In a study of 84 scientific applications of high-performance computers, Fox found that 70 applications, or 83% of those considered, fit the data-parallel model [Fox 88].

In the data-parallel model, a program has a single locus of control — a single "program counter" that advances through the program. This single execution stream invokes small pieces of parallel code. This model is supported directly by SIMD machines, including the Connection Machine CM-1 and CM-2 architectures and the MasPar MP-1 and MP-2; the existence of these machines, which run only data-parallel code, has fostered the development of a rich variety of data-parallel algorithms [Hillis & Steele 86].

Recent research has shown how (SIMD) data-parallel programs can be run efficiently on MIMD hardware [Hatcher & Quinn 91, TMC 91]. This is done by using compile-time analysis to determine where and when synchronization and communication are needed to maintain the illusion of lockstep behavior in spite of physically unsynchronized processors. Optimizing compilers for these systems attempt to aggregate several logical synchronization/communication operations into a single operation, thereby increasing the granularity of the computation and hence making it more efficient.

The "loosely lockstep" behavior of the programs generated by these compilers matches a common programming pattern on MIMD systems. This pattern is sometimes referred to as SPMD (single-program, multiple-data) parallelism. It is also the idea captured by Fox's "loosely synchronous" programs [Fox et al. 88] and Snyder's XYZ model [Griswold et al. 90]. These ideas, in turn, have spawned research on data-parallel languages with weaker synchronization [Larus et al. 92]; for example, processes may (logically) synchronize after a set of statements rather than after each instruction.

Because data-parallel programs have a single locus of control, the programmer retains the usual sequential notion that the program is "at a single place" at any

point of the execution. This not only eases programming, but also allows standard debuggers and performance analysis tools to be easily adapted to the data-parallel case. In addition, most data-parallel programs are either deterministic, or have their nondeterminism carefully encapsulated in a few constructs. Again, this eases debugging.

The attractiveness of the data-parallel model has led to the development of many data-parallel languages. Early data-parallel languages include Dino [Rosing et al. 90, Rosing 91], Kali [Koelbel et al. 90], and C* [Rose & Steele 87]. The current generation of data-parallel languages is more mature, and offers a richer set of directives to control data distribution [TMC 89, TMC 90, Fox et al. 91, Chapman et al. 92]. A consortium of researchers and vendors is now developing a standardized language called High Performance Fortran (HPF) with some data-parallel features [HPFF 93]. HPF will be supported by all the major parallel system vendors, and will provide a common platform for researchers studying implementation of data-parallel languages.

Compilers for data-parallel languages usually generate message-passing programs as output. This is done because message-passing is the "lowest common denominator" interface, and also because explicit message-passing gives the compiler the best opportunity to carefully manage communication. Despite this careful management, compiler-generated programs typically communicate more than hand-written message-passing programs would, so communication performance is a particularly acute problem for compiler-generated code [Hatcher et al. 91].

Data-parallel languages are easier to compile than other parallel languages, because the single locus of control allows the compiler to deduce that all processes are in the same small section of code at the same time. This information allows the compiler much more latitude to perform optimizations. The same advantage is available to a protocol compiler when it is processing a data-parallel program; because processes work their way through the program in semi-lockstep, the protocol compiler can deduce much more about how processes interact with each other. As we will see, this enables the protocol compiler to generate highly efficient protocols.

4.2.1 Limits of the Data-Parallel Model

As discussed above, the data-parallel model fits a wide range of scientific and other numerical programs. However, it does not support all parallel programs. The most

common structure it does not support is the work-heap model. In this model, a set of worker processes carry out tasks that are stored on a central work-heap. Each worker repeatedly requests a task from the heap and processes it, perhaps adding more tasks to the work-heap. The work-heap model is simple, and applies naturally to many search or divide-and-conquer programs. The work-heap model is amenable to careful, specialized implementation to reduce protocol overhead.

4.3 Representing Communication Behavior

The first step in analyzing a program's communication behavior is to build a representation of that behavior. This representation is then used by the protocol compiler to analyze and improve the program's behavior. We can consider the representation problem to be one of designing a formal language that specifies the communication behavior of programs. This section discusses the general issues surrounding the choice of this representation, and describes the particular solution used in Parachute.

The source program itself is one possible representation. However, it is inconvenient for several reasons. The most important of these is that the source code contains too much information. The protocol compiler focuses its attention on communication, and treats local computation phases as black boxes. Using the source code forces the protocol compiler to handle unneeded complexity.

We may choose to extract the necessary information directly from the source code, and then pass it on to the protocol compiler. This is a useful thing to do, but it still leaves us with the question of in what form the protocol compiler should accept the information.

In order to be precise, it is helpful to use a *communication description language* to describe the communication behavior of a program. This language maps directly to the protocol compiler's internal representation of a program's behavior. Thus, defining the language is equivalent to stating how the protocol compiler represents communication behavior.

4.3.1 Representing Behavior in Parachute

Parachute deals with data-parallel programs written with explicit message-passing, rather than dealing directly with code written in a data-parallel language. This

decision has two main benefits. First, it allows the protocol compiler to accept hand-written message-passing code, provided the code adheres to the data-parallel style. This is important because the majority of existing programs use hand-written message-passing rather than parallel languages. Second, having the protocol compiler accept explicit message-passing allows me to avoid having to implement a full compiler for a data-parallel language in addition to the main body of the protocol compiler.

On the other hand, having a complete parallel-language compiler would have allowed more opportunities to transform the program to improve performance. For example, a full compiler could use feedback to incorporate the results of the protocol compiler's analysis into earlier optimization phases of the compiler. In addition, the compiler's analysis might provide more information to the protocol compiler about the order in which events can be allowed to take place at runtime.

We now turn to the design of Parachute's mechanism for representing communication behavior. Rather than attempting to define a completely general communication description language, I define a description language that applies to the restricted class of programs that Parachute accepts.

4.3.2 Communication Patterns

Parachute accepts only data-parallel programs, and it generates tailored protocols only for *communication patterns*. A communication pattern is some fixed "conversation" that takes place between a fixed set of processes. Communication patterns capture the notion that there are small sections of the program whose communication behavior is completely known in advance; for example, an FFT procedure carries out the same sequence of communications every time it is executed.

To specify a communication pattern, one must specify two things:

- a list of the processes that participate in the pattern, and
- for each process participating in the pattern, exactly what sequence of communication calls that process will make within the pattern, including the destination, the tag, and (an upper bound on) the message-size arguments of all `send` calls, and the tag-specifier and (an upper bound on) the message-size arguments of all `receive` calls.

All messages sent within a pattern must be received within the same pattern, and vice versa. Thus, a pattern is a self-contained unit of communication.

Parachute operates exclusively on communication patterns. Only communication within such a pattern will be optimized by Parachute. Communication which is not part of a communication pattern falls outside Parachute's domain, and so is implemented using standard protocols.

```
var
   x,y : ARRAY [2048] of COMPLEX

for i ← 0 to log2 P − 1 {
   send x to xor(pid, 2^i) tag i
   recv y tag i
   fftStep(x, y, i)
}
```

Figure 4.2: Pseudocode for a parallel FFT. P denotes the number of processes, and *pid* is the number of the current process.

We can represent a communication pattern by drawing a diagram of the communication history of each process within the pattern. Such diagrams are drawn by trace analysis tools like ParaGraph [Heath & Etheridge 91], where they are called "space-time diagrams" or "Feynman diagrams." Figure 4.2 shows pseudocode for a one-dimensional FFT computation, and figure 4.3 shows the spacetime diagram of its communication pattern. The diagram is not meant to denote the exact timing of events, but merely the sequence of events seen by each process.

4.3.3 Describing Communication Patterns Formally

Although spacetime diagrams are useful for visualizing and understanding communication patterns, they are not precise enough to describe all the patterns that can

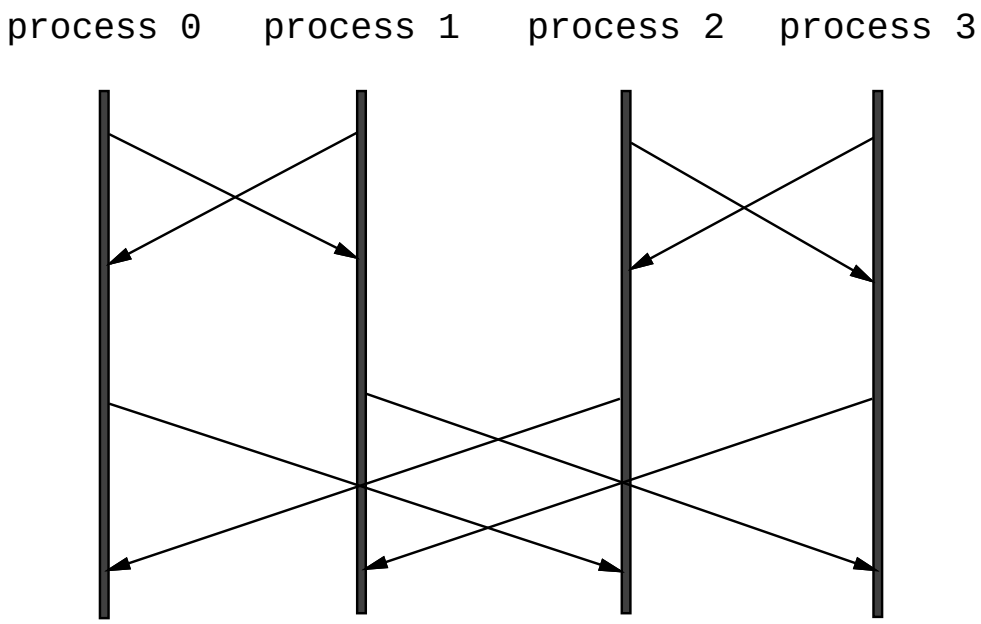

Figure 4.3: Spacetime diagram of the communication pattern induced by the FFT computation of figure 4.2. The diagram is a time-line with time increasing downward. Each vertical line denotes the history of a single process, and arrows between vertical lines denote messages transmitted; the arrow points from the sender to the receiver of the message. The diagram is not concerned with the exact time at which each event happens, but only with the sequence of events as seen by each process.

occur in real programs. This section presents a formal language for specifying communication patterns. A slightly augmented version of this language is used to specify the communication behavior of programs to Parachute.

Parachute's internal representation of communication patterns is isomorphic to this language, so describing the language is equivalent to describing the internal representation. Indeed, this language can be viewed as the formal *definition* of a communication pattern — anything that can be expressed in the language is a communication pattern, and anything that cannot be expressed in the language is not.

The language is called PDL (pronounced "puddle"), for "pattern description language". I present PDL by giving examples; a full grammar appears in appendix A. Some syntactic sugar that appears in the real PDL language is omitted. Keywords appear in **boldface**, and other tokens in `typewriter` style. Phrases or nonterminals are represented by *italics*.

Basic Communication Operations

PDL has four basic communication primitives: `beginSend`, `endSend`, `beginRecv`, and `endRecv`. `beginSend` and `endSend` combine to allow a split-phase message-send operation, and `beginRecv` and `endRecv` offer a split-phase message-receive operation. (Recall that a split-phase, or non-blocking, operation is one that has separate `begin` and `end` calls; arbitrary computation and communication can thus be overlapped with a split-phase operation.) Operations with other semantics, such as blocking send/receive, or CSP rendezvous, can be implemented trivially on top of the four primitives. PDL also offers a `send` primitive which is just syntactic sugar for a `beginSend`/`endSend` pair, and a `recv` primitive which is syntactic sugar for a `beginRecv`/`endRecv` pair.

Each communication pattern is named by an integer pattern-ID. To specify pattern number 17, one writes

pattern 17 {
  pattern-description
}

where *pattern-description* describes the behavior of each process participating in the

pattern.

A pattern description provides the names of all processes participating in the pattern, and gives a sequence of communication calls for each process:

process `1` {
  call-sequence
}

A call sequence is simply a sequence of zero or more call-descriptors, where a call descriptor gives the name of an MP0 call, and values for some of its arguments. Here are some legal call descriptors:

beginSend dest `4` **tag** `717` **maxsize** `1024` **name** `message1`
endSend name `message2`
beginRecv tag `ANY` **maxsize** `42k` **name** `arthur`
endRecv name `jane`

The "name" arguments are required to match up each `beginSend` with the corresponding `endSend`, and each `beginRecv` with the corresponding `endRecv`. At runtime, the `beginSend` or `beginRecv` call will return a message-identifier, which will later be passed to the `endSend` or `endRecv` call. This mechanism is awkward to describe in the pattern description, so the "name" mechanism is used instead.

We can put all these bits of grammar together to yield the full language. For example, figure 4.4 shows the pattern description for a one-dimensional FFT on four processes. Although this example is small and has a regular structure, general communication patterns can be larger and asymmetric, so long as the components of the pattern are known in advance.

4.3.4 Design Rationale for the Description Language

I designed PDL with four goals in mind:

1. It should be as simple as possible.

2. It should capture all the information Parachute needs to do its analysis.

```
pattern 0 {
  process 0 {
    send dest 1 tag 0 maxsize 16k
    recv tag 0 maxsize 16k
    send dest 2 tag 1 maxsize 16k
    recv tag 1 maxsize 16k
  }
  process 1 {
    send dest 0 tag 0 maxsize 16k
    recv tag 0 maxsize 16k
    send dest 3 tag 1 maxsize 16k
    recv tag 1 maxsize 16k
  }
  process 2 {
    send dest 3 tag 0 maxsize 16k
    recv tag 0 maxsize 16k
    send dest 0 tag 1 maxsize 16k
    recv tag 1 maxsize 16k
  }
  process 3 {
    send dest 2 tag 0 maxsize 16k
    recv tag 0 maxsize 16k
    send dest 1 tag 1 maxsize 16k
    recv tag 1 maxsize 16k
  }
}
```

Figure 4.4: Pattern description for the FFT program of figure 4.2.

3. It should be easy for the programmer or compiler to generate, understand, and edit.

4. It should be extensible to meet future requirements.

The language addresses these goals well. PDL captures the actions of each process within each communication pattern; this is the information Parachute needs. Each communication operation is described using the same name, and in roughly the same form, as in the source program, so the programmer should be able to understand how a PDL description relates to his program. The language can be parsed by `yacc`, so it is easy to extend.

The main omission in PDL is the lack of timing information. This stems from the lack of any description of local computation. Since local computation does not appear explicitly in the description, Parachute has no way of knowing when local computation might happen or how long it might take. This prevents Parachute from using detailed timing estimates to make its decisions. Although calculating and using timing estimates may be problematic, it would be useful to have the opportunity to try them. Future versions of the description language are likely to have an operation corresponding to local computation, with an optional estimate of the duration of the computation.

Several other languages have been used to describe communication behavior and protocols. The best-known is LOTOS [ISO 88, van Eijk et al. 89, Logrippo et al. 92], devised by ISO to specify and model the OSI standard protocol suite. LOTOS is much more ambitious and complex than PDL. Although LOTOS is in some sense a superset of PDL, expressing communication patterns in LOTOS would be overkill.

LOTOS is, however, useful as a tool for formally specifying both the behavior of the underlying communication networks, and the precise semantics of operations in MP0 or similar systems.

4.3.5 Communication Patterns in the Context of the Entire Program

At run-time, Parachute allows communication patterns to be mixed freely with local computation and non-pattern communication. For example, arbitrary local computation can take place during a communication pattern, or non-pattern messages can be sent or received during a communication pattern.

The programmer or compiler is responsible for adding annotations to the source program to mark the beginning and end of executing a communication pattern. For example, the call `beginPattern(13)` denotes the beginning of pattern 13. Until a call to `endPattern(13)`, all communication is assumed to be part of the pattern, unless it is explicitly marked as non-pattern communication. (The marking would be done by using calls like `patternTimeOut` and `patternTimeIn`.)

When a pattern is executing, the tailored protocol ensures that each process executes the sequence of communication calls that the pattern calls for. If the pattern is violated, the implementation will detect the violation and generate a run-time error. Further details of the run-time library appear in section 6.3.

4.3.6 Recognizing Communication Patterns

Since Parachute requires the beginning and end of each communication pattern to be marked with an annotation, we must have some strategy for identifying communication patterns. At present, this is done by the programmer. This problem takes two forms, depending on the circumstances. If the program is being written to use Parachute, communication patterns can be built into the structure of the program. If an existing program is being adapted to use Parachute, communication patterns must be recognized.

The first problem is the easier one. Message-passing programs are usually designed by drawing pictures of data layout, and estimating performance by calculating the number and size of messages. This process leads naturally to the identification of communication patterns.

The second problem, finding patterns in existing code, is harder. This is the problem I faced when adapting to Parachute the five applications programs described in chapter 7. The remainder of this section outlines the strategies I developed while annotating these programs.

The first tactic is to concentrate on those parts of the source code where the program actually spends its time. Since one can choose to use the protocol compiler for a subset of the program, it makes sense to ignore sections of the code where not much time is spent.

Once the relevant sections of the program have been identified, the next step is to look for likely candidates for identification as patterns. A candidate might be some

procedure that is invoked in a data-parallel fashion, or the body of a loop.

Once a few probable patterns have been identified, these are marked in the source code, and Parachute is run on them. If the annotations are erroneous (if the communication inside one of the marked "patterns" is not really data-independent) a run-time error will be signaled. Usually, no errors are signaled and the annotation process is finished. If an error is found, the offending region of code is broken up into several smaller patterns, and the process is repeated again.

Although this procedure is less rigorous that we would like, it seems to work very well in practice. Eventually, we would like the compiler to automatically identify communication patterns, or at least to check the programmer's annotations for correctness.

4.4 Summary

This chapter considers the first of three major issues in protocol compiler design: how to describe and represent the communication behavior of application programs. This problem is simplified in the case of Parachute, because it deals with a restricted class of parallel programs. Parachute operates on communication patterns, which can be described in the PDL description language. A communication pattern consists of a known set of processes exchanging messages in a known sequence. By analyzing a program's communication patterns separately, Parachute makes its job much easier.

Chapter 5

PROGRAM ANALYSIS ISSUES IN PROTOCOL COMPILER DESIGN

And ye shall know the truth, and the truth shall make you free.
— CIA Motto
Mostly, the truth makes us nervous.
— Anonymous CIA Analyst

This chapter discusses three important issues that arise in designing a protocol compiler. Each issue is relevant to the general problem of protocol compiler design; when specialized to the case of Parachute, each issue is dealt with by one phase of Parachute's analysis.

The first of the three issues is how to deal with nondeterministic programs. Nondeterminism is useful for the programmer, but it makes the protocol compiler's job more difficult by expanding the set of possible situations that must be planned for. The general solution to this problem is to resolve the nondeterminism — to transform the program by making certain nondeterministic "choices" in advance, rather than letting them depend on accidents of timing.

The second issue is whether and how to change the means of transmitting each message across the network. There are several ways in which one might change the mode of a message: for example, one might introduce a rendezvous between the sender and the receiver. Such changes might affect the behavior of the program — we must somehow guarantee that a change is safe before it is made.

The third issue is how to use memory buffers to realize the communication requirements of a program. Messages are often buffered to improve performance and to ensure correctness. Typical message-passing libraries allocate buffers dynamically to messages. Using a protocol compiler enables us to make some of these decisions at compile-time, thus reducing runtime overhead. Of particular importance is the decision to share buffers between a set of messages.

5.1 Nondeterminism

Our first major issue is nondeterminism. A program is nondeterministic if, at some point in its execution, it can exhibit more than one behavior, and the choice between these behaviors is arbitrary. In message-passing programs, nondeterminism arises because of the tag-matching in `receive` operations. The tag-specifier in some `receive` statement may match more than one incoming message; in this case the implementation is free to deliver any of the matching messages, subject to the requirement of in-order delivery.

Process P_0	Process P_1	Process P_2
receive x tag ANY	$v \leftarrow f(2)$	$w \leftarrow f(3)$
$y \leftarrow g(x, 0)$	send v to P_0 tag 42	send w to P_0 tag 91
receive z tag ANY		
print $g(y, z)$		

Figure 5.1: This program is nondeterministic because either the two `receive` statements in P_0 may match either of the two messages sent to P_0.

Figure 5.1 shows one example of a nondeterministic program. Two messages are sent to process P_0, and P_0 may receive these messages in either order. Depending on the properties of the functions f and g, this nondeterministic choice may or may not affect the output of the program. Even if the output of the program is unaffected, we still say the program is nondeterministic; although the user does not notice the nondeterminism, the message-passing system does.

Figure 5.2 shows a program very similar to the previous example. This program, however, is deterministic. Although the tag-matching predicate in P_0's first receive statement matches the message sent by P_2, P_2 cannot send its message until after P_0's first receive statement has completed. As this program illustrates, nondeterminism does not necessarily follow from complex tag-matching situations.

Process P_0	Process P_1	Process P_2
receive x tag ANY	$v \leftarrow f(2)$	$w \leftarrow f(3)$
send tag 25 to P_2	send v to P_0 tag 42	receive tag 25
$y \leftarrow g(x, 0)$		send w to P_0 tag 91
receive z tag ANY		
print $g(y, z)$		

Figure 5.2: This program is deterministic. P_0's first receive statement must match the message with tag 42, and its second receive must match the message with tag 91.

5.1.1 Making a Virtue of Nondeterminism

At first glance, nondeterminism appears to make the protocol compiler's job harder. The fact that a program might exhibit several different behaviors when given the same input seems to force the protocol compiler to deal with a wider range of program behaviors.

Consider a program that can exhibit two behaviors, A or B, depending on a nondeterministic choice. (The reader may imagine that the choice is made by accidents of timing.) It may be that the protocol compiler can generate a tailored protocol for behavior A, and a separate tailored protocol for behavior B, but cannot determine a protocol for their nondeterministic combination.

On the other hand, the protocol compiler can use nondeterminism to make its job easier. To understand this, we must carefully review the definition of nondeterminism. Imagine that a program, in some state A, moves nondeterministically into either state B or state C. How is the choice made between B and C? The implementation is free to make the choice in whatever manner it likes. In particular, the protocol compiler can decide to *resolve* the nondeterminism by decreeing that the program will always choose B. For example, in the program of figure 5.1, the protocol compiler could decide which of the two incoming messages should be matched by P_0's first receive statement.

We can think of the application program as a *specification* of which behaviors are allowed at run-time, given a particular input. The protocol compiler is free to

transform the program however it likes, provided the result meets the specification[1]. If nondeterminism plays a large role in the program, the specification is permissive, so the protocol compiler has a great deal of freedom in generating code for the program.

Informally, the protocol compiler must follow two rules when resolving nondeterminism:

1. If the initial program is deadlock-free, then the generated program must also be deadlock-free.

2. Any execution of the generated program must be equivalent to some possible execution of the initial program.

Resolving nondeterminism cannot affect the correctness of the program, but may affect performance.

5.1.2 Retagging

For message-passing programs, nondeterminism occurs only via tag matching. A protocol compiler can resolve this nondeterminism by changing the program's tag matching. I refer to this procedure as *retagging*.

Process P_0	Process P_1	Process P_2
receive x tag 91	$v \leftarrow f(2)$	$w \leftarrow f(3)$
$y \leftarrow g(x, 0)$	send v to P_0 tag 42	send w to P_0 tag 91
receive z tag 42		
print $g(y, z)$		

Figure 5.3: The program of figure 5.1, after retagging.

[1] The theoretically inclined reader may consider the "behavior" of a program to be specified by the set of traces the program may generate, where a trace is the list of communication events that take place in a complete execution of the program. Define $traces(P, I)$ to be the set of traces that program P can generate given input I. If the original application program is called A, then the protocol compiler is allowed to generate any program B such that, for all I, $traces(B, I) \subseteq traces(A, I)$.

Consider once again the nondeterministic program of figure 5.1. The first `receive` operation of process P_0 is nondeterministic — it will match either P_1's message with tag 42, or P_2's message with tag 91. We can retag this program by changing the tag-specifier of P_0's first `receive` to insist on tag 91, thereby forcing it to match the message sent by P_2. Having done this, we might as well change the tag specifier of P_0's second `receive` statement. The result of this retagging is shown in figure 5.3. Note that this program does not exhibit the *same* behaviors as the original, but only a subset of the original's possible behaviors.

Process P_0	Process P_1	Process P_2
receive x tag 91	$v \leftarrow f(2)$	$w \leftarrow f(3)$
send tag 25 to P_2	send v to P_0 tag 42	receive tag 25
$y \leftarrow g(x, 0)$		send w to P_0 tag 91
receive z tag 42		
print $g(y, z)$		

Figure 5.4: This program results from a careless retagging of the program of figure 5.2. This program deadlocks — P_0 and P_2 wait infinitely for each other.

Now imagine that we tried to apply the same retagging operations to the deterministic program of figure 5.2. Doing this leads to the program shown in figure 5.4. This program always deadlocks, with P_0 and P_2 blocked waiting for each other. Thus, we cannot retag a program without doing careful analysis of the effects of the retagging.

Process P_0	Process P_1	Process P_2
receive x tag 42	$v \leftarrow f(2)$	$w \leftarrow f(3)$
$y \leftarrow g(x, 0)$	send v to P_0 tag 42	send w to P_0 tag 42
receive z tag 42		
print $g(y, z)$		

Figure 5.5: This nondeterministic program is equivalent to the program of figure 5.1.

We have seen one kind of retagging, which narrows the tag-specifier of a `receive` statement. The program in figure 5.5 presents an opportunity for another kind of retagging operation. This program is equivalent to our original nondeterministic example; however, in this case both messages have the same tag.

Process P_0	Process P_1	Process P_2
receive x tag (42 or τ)	$v \leftarrow f(2)$	$w \leftarrow f(3)$
$y \leftarrow g(x, 0)$	send v to P_0 tag 42	send w to P_0 tag τ
receive z tag (42 or τ)		
print $g(y, z)$		

Figure 5.6: The program of figure 5.5, after retagging.

Our new retagging operation will give the two messages different tags, without affecting the meaning of the program. We arbitrarily choose P_2's message to be assigned a new, unique tag τ. This requires that any `receive` statements that might have matched the original tag be widened so that they also match the new tag. The resulting program is shown in figure 5.6. This retagging operation is always legal, because it does not affect the meaning of the program. Its main usefulness is in exposing opportunities for the first kind of retagging.

The fact that retagging is legal does not necessarily make it a good idea. In some cases, preserving the nondeterminism gives the run-time system the ability to resolve the nondeterminism in a convenient way. For example, the nondeterminism in the program of figure 5.1 gives the run-time library the ability to resume P_0's execution as soon as either of the two messages arrives, rather that waiting for a predetermined one. The protocol compiler must decide when to remove nondeterminism, and when to leave it alone.

5.1.3 Retagging in Parachute

Since Parachute deals only with known communication patterns, it faces a restricted class of retagging problems. Parachute's strategy is simple: it removes *all* nondeterminism from the communication patterns it is given. Although this decision has

a potential cost in performance, it simplifies the jobs of later parts of Parachute's analysis, by insulating them from issues of nondeterminism. This was a good choice for the first prototype, but is probably not the best choice in general.

Conceptually, Parachute first changes the tags of all `send` statements so they are unique, and then narrows the tag-specifier of each `receive` until it refers to a single message. This procedure is called *message matching* because it generates a one-to-one matching between `send`s and `receive`s. In fact, rather than carrying out retagging operations one by one, Parachute uses an algorithm that directly computes a legal message matching.

5.1.4 Message Matching

The goal of the message-matching procedure is to produce a matching between `send` and `receive` operations that is consistent with the time ordering imposed by the processes' programs. (Within a communication pattern, a process's "program" is simply the sequence of communication operations it makes within that pattern.) The semantics of a split-phase ("begin/end") operation say that the operation appears to take place atomically sometime between the `begin` and `end` calls. The implementation may choose any point between the `begin` and `end` to carry out the operation.

To give the message-matching algorithm as much flexibility as possible, the analysis will assume that each message is sent as early as possible and received as late as possible; this gives the analysis the maximum amount of "slack" in timing. Thus, we will treat `beginSend` as if it actually sends the message, and `endRecv` as if it actually receives the message. The message-matching phase ignores `endSend` and `beginRecv` operations.

Certain message-matching problems are *ill-formed*: there is no way to match `beginSend`s and `endRecv`s, even if execution order is ignored. For example, a problem is ill-formed if it contains more `beginSend`s than `endRecv`s.

Ill-formed problems can be detected by solving a simple bipartite matching problem. This matching problem has two sets of nodes, S and R, with one node in S for each `beginSend` and one node in R for each `endRecv`. There is an undirected edge between s_i and r_j if the tag of s_i matches the tag-specifier of r_j. If this bipartite matching problem has no solution, then the original message-matching problem is ill-formed. If so, this fact is reported to the user, along with a possible hint about

which message(s) failed to match.

Unfortunately, not all solutions to the bipartite matching problem correspond to legal message-matchings. This is because some proposed matchings may be causally impossible; they may require some message to be received before it is sent.

To determine whether a proposed matching is causally possible, we can construct a data structure called an *event ordering graph.* This is a directed graph that expresses all the known temporal relations between the operations in the pattern. It has a node for each communication call; a path from a to b signifies that event a happens before event b in any execution of the program. The graph is built from the following sets of edges:

- *program order edges:* These edges express the fact that each process must execute its statements in the order in which they appear in its program. Thus, if a and b are executed by the same process, and a directly precedes b in that process's program, then the graph has an edge (a, b).

- *message order edges:* These edges express the fact that a message must be sent before it can be received. If s_i and r_j are, respectively, `beginSend` and `endRecv` statements, and those two statements have been matched, then the graph has an edge (s_i, r_j).

A proposed matching is causally possible if and only if its event ordering graph is acyclic.

The event ordering graph is described in more detail in section 5.2.3. This data structure is re-used in later phases of the protocol compiler's analysis.

The Message-Matching Algorithm

Parachute matches messages by *symbolically executing* the program. The algorithm maintains an ordered list of messages in transit; the list is initially empty. Each process is given a "program counter," which initially points to the beginning of that process's sequence of communication calls. The algorithm then picks a process arbitrarily, and tries to symbolically execute the next communication call in that process's program. If the call is a `beginSend`, an entry describing the sent message is added to the list of messages in transit, and the process's program counter is advanced to

point to the next statement. If the call is an `endSend` or a `beginRecv`, the process's program counter is simply advanced to the next statement. If the call is an `endRecv`, and some message in the pending-message list has a destination and tag matching the `endRecv` call, then the first matching message is removed from the pending-message list and the process's program counter is advanced. Otherwise, if there is no matching message in the pending-message list, the algorithm chooses another process to execute.

Symbolic execution continues until either all processes have symbolically executed all their communication calls, or until no process can make progress. In the first case, the message-matching algorithm has succeeded: the program, after retagging, will run correctly without deadlock. In the second case, the message-matching algorithm is stuck: either the program may deadlock, or the sends and receives do not pair up properly. This fact is reported to the programmer as an error.

An Example

As an example, we will simulate the message-matching algorithm on the FFT computation of figure 4.4. Figure 5.7 shows the program with each statement labelled for clarity, and the initial state of the message-matching algorithm. The algorithm has five main variables: a "program counter" for each process, and a list of pending messages. The program counters are initialized to point to the first statement in the program of each process, and the pending-messages list is initially empty.

Now suppose that the algorithm symbolically executes three statements, one each from P_1, P_3 and P_0. The resulting state is shown in figure 5.8.

At this point, the algorithm chooses P_3 to undergo the next simulated step. P_3 is due to execute a receive statement, and no matching message is pending. Hence, P_3 cannot execute a step and some other process must be chosen. Suppose that the next four steps are by processes P_0, P_0, P_2, and P_1. The state is now as shown in figure 5.9.

This procedure continues until all program counters have reached the end of their respective programs, yielding the state shown in figure 5.10. Since all programs have terminated and there are no pending messages, the algorithm has succeeded. All messages have been matched.

Process P_0	Process P_1	Process P_2	Process P_3
S_0: send P_1 tag 0	T_0: send P_0 tag 0	U_0: send P_3 tag 0	V_0: send P_2 tag 0
S_1: recv tag 0	T_1: recv tag 0	U_1: recv tag 0	V_1: recv tag 0
S_2: send P_2 tag 1	T_2: send P_3 tag 1	U_2: send P_0 tag 1	V_2: send P_1 tag 1
S_3: recv tag 1	T_3: recv tag 1	U_3: recv tag 1	V_3: recv tag 1

PC_0	PC_1	PC_2	PC_3	pending messages
S_0	T_0	U_0	V_0	$\emptyset$

Figure 5.7: Initial state of the message matching algorithm.

PC_0	PC_1	PC_2	PC_3	pending messages
S_1	T_1	U_0	V_1	P_1 to P_0 tag 0
				P_3 to P_2 tag 0
				P_0 to P_1 tag 0

Figure 5.8: A state of the message matching algorithm.

PC_0	PC_1	PC_2	PC_3	pending messages
S_3	T_2	U_1	V_1	P_3 to P_2 tag 0
				P_0 to P_2 tag 1
				P_2 to P_3 tag 0

Figure 5.9: A state of the message matching algorithm.

PC_0	PC_1	PC_2	PC_3	pending messages
end	end	end	end	$\emptyset$

Figure 5.10: A state of the message matching algorithm.

Flexible Message-Matching

The message-matching algorithm contains a series of arbitrary choices: how to interleave the execution of processes. The output of the message-matching algorithm depends on how these arbitrary choices are made. The algorithm cannot find *all* valid matchings, except by an exhaustive search of many possible interleavings. This limitation has two consequences. First, if several matchings are possible, the algorithm may be unable to choose the one that is "best" in some sense, for instance the one with the shortest estimated execution time or the one in which the total expected waiting time is minimized.

Second, there are some erroneous programs that run correctly under some interleavings but deadlock under other interleavings. For some such programs, the algorithm will find a legal matching without noticing that the program is erroneous; after matching, these programs are guaranteed not to deadlock. For other such programs, the algorithm will report that the program deadlocks, without noticing that it might sometimes run to completion. This behavior is acceptable if we define a program that sometimes deadlocks as erroneous, and do not demand that the algorithm report possible errors in erroneous programs.

While it would be possible to detect all such erroneous programs, doing so would require an exhaustive search of many possible interleavings of the processes' executions. Finding an efficient algorithm to detect such programs, or proving that no such algorithm exists, remains an open problem.

5.1.5 Summary

Some message-passing programs are nondeterministic, because the tag-specifiers in their `receive` statements can match the available messages in more than one way.

A protocol compiler can simplify a program by resolving its nondeterminism:

making some nondeterministic "choices" at compile-time. Although this may hurt performance by reducing the freedom of the run-time library, it makes the protocol compiler's job easier by reducing the variety of behaviors that the application program can exhibit.

Nondeterminism is removed from message-passing programs via retagging, which changes the tags of send statements and the tag-specifiers of receive statements. In Parachute, this takes the form of message-matching, an operation that removes all nondeterminism from a communication pattern.

5.2 Message-Passing Modes

Our second major issue is how to transport each message from the sender to the receiver. That is, what sequence of operations should take place to cause the message to get safely from sender to receiver? We can think of this as the problem of choosing which *mode* to use for each message. Intuitively, message-passing modes are the "atoms" and tailored protocols are the "molecules" built by stringing these atoms together.

There are several modes that we might use to transport a message. They differ primarily in how they divide responsibility for buffering and flow control decisions between compile-time analysis and run-time mechanisms on the sender and receiver. Among the message-passing modes are:

1. *blast mode*, in which the sender simply transmits the full body of the message as soon as it is ready, and the message is delivered directly to the receiving process on arrival. Blast mode requires compile-time analysis to ensure that the receiver will be ready for the message before the sender can possibly transmit it.

2. *synchronizing mode*, in which the sending and receiving processes rendezvous, then transfer the message. Synchronizing mode requires compile-time analysis to ensure that enforcing a rendezvous between sender and receiver does not cause the program to deadlock.

3. *receiver-buffered mode*, in which the sender transmits the message as soon as it is ready, and the message is stored in a memory buffer on the receiving processor

until the receiving process is ready for it. This mode requires compile-time analysis to allocate buffer space for the message on the receiving processor.

4. *sender-buffered mode*, in which the message is copied to a memory buffer on the sending processor, the receiving process signals the sender when it is ready for the message, and the message is then transmitted. This mode requires compile-time analysis to allocate buffer space for the message on the sending processor.

5. *dynamic mode*, in which the message is handled by a general message-passing protocol like the pre-reservation and sliding-window protocols of section 2.4.2. This mode requires no compile-time analysis.

The choice among these modes is made on the basis of correctness and efficiency. Most of the modes place some responsibility on the protocol compiler to verify certain properties of the computation, or to allocate buffer space. If the protocol compiler is unable to satisfy these requirements, it can "fall back" on dynamic mode, which is the conservative thing to do, since all messages were presumably handled in dynamic mode before the protocol compiler came along[2].

Once correctness issues have been dealt with, the choice between modes is made on the basis of performance. This requires the protocol compiler to have some kind of performance model for the target architecture, so it can compare the likely performance of the various alternatives.

5.2.1 How Parachute Chooses Message Modes

Parachute uses three of the message modes: blast, synchronizing, and receiver-buffered. It does not use sender-buffered or dynamic modes, because doing so does not offer any advantages on the particular architecture Parachute targets.

---

[2] When static and dynamic buffer allocation must be used simultaneously, a messy problem can arise. The question is how much buffer space to allot to the dynamic allocator. If the dynamic allocator is given insufficient space, deadlock can result. In an ideal world, we would give the dynamic allocator sufficient space so that the deadlock-avoidance guarantees of standard (dynamic-only) systems were preserved. Unfortunately, real systems do not provide any clearly-defined guarantees; they take a "best effort" approach. One reasonable solution is to provide buffer space roughly comparable to what the standard, dynamic mechanism would use.

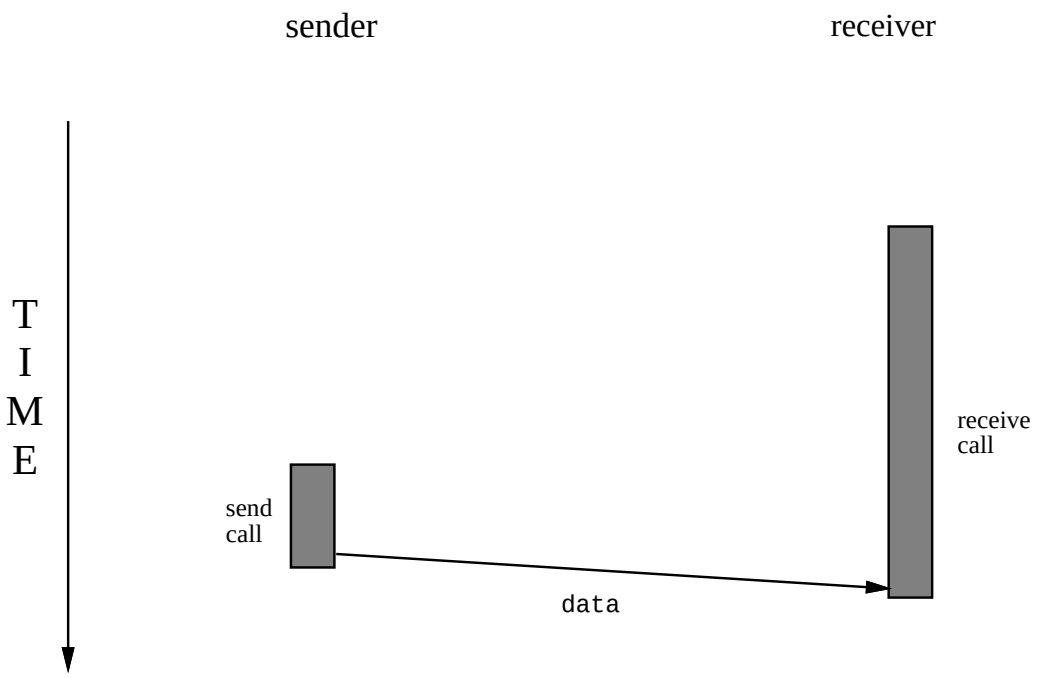

Figure 5.11: Delivery of a message in blast mode. Blast mode is incorrect unless the receiver is ready for the message before the sender transmits it.

Figure 5.11 shows a message being transmitted in blast mode. Blast mode is the most efficient of the communication modes, so Parachute uses blast mode every time it has an opportunity to do so. Blast mode can be used whenever the Parachute can deduce that the receiving process executes its `beginRecv` call before the sending process calls its `beginSend`. Since this strategy is so simple, I will make no further mention of blast mode in this section; the reader may assume that Parachute recognizes and takes advantage of opportunities to use blast mode whenever they arise.

5.2.2 Synchronizing Mode vs. Receiver-Buffered Mode

The remaining question, then, is how to decide between synchronizing mode and receiver-buffered mode. This is referred to as the buffering-mode analysis problem.

In the synchronizing mode, depicted in figure 5.12, the sender and receiver rendezvous to transfer the message. When the receiver is ready for the message to arrive, it sends a hardware packet to the sender saying "ready". Once the sender is ready to transmit, and has received the "ready" packet, the sender transmits the message body. On arrival at the receiving processor, the data is placed directly in the memory of the receiving process.

In the receiver-buffered mode, depicted in figure 5.13, the sender transmits the message body as soon as it is ready, without waiting for a signal from the receiver. If the message body arrives at the receiving processor before the receiving process is ready for the message, the message is buffered in memory. When the receiving process is ready for the message, it copies the message data from the buffer into its memory.

When deciding which mode to use for each message, there are two factors that must be taken into account. First, using synchronizing mode adds a rendezvous to the execution. This added synchronization may introduce a deadlock into the program. The analysis must avoid deadlock — a correct input program cannot lead to a deadlocking execution. Second, there may be many choices that avoid deadlock, and the analysis must decide among them; this is done on the basis of execution cost, in terms of both time and space.

Depending on the circumstances, either synchronizing or receiver-buffered mode may lead to faster execution. Receiver-buffered mode has the extra cost of copying

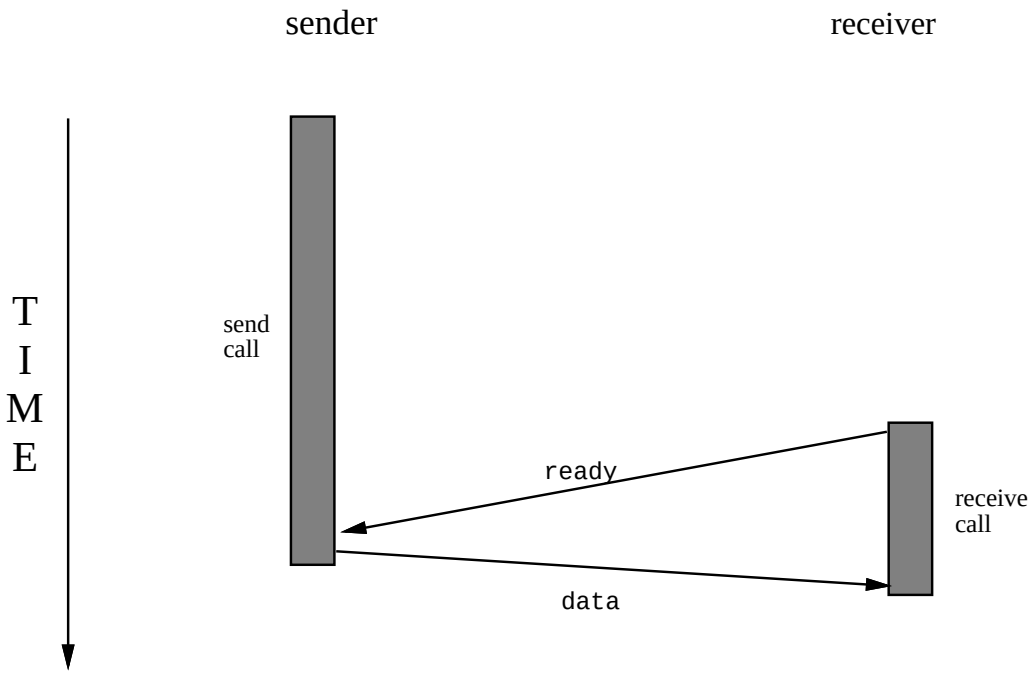

Figure 5.12: Delivery of a message in synchronizing mode.

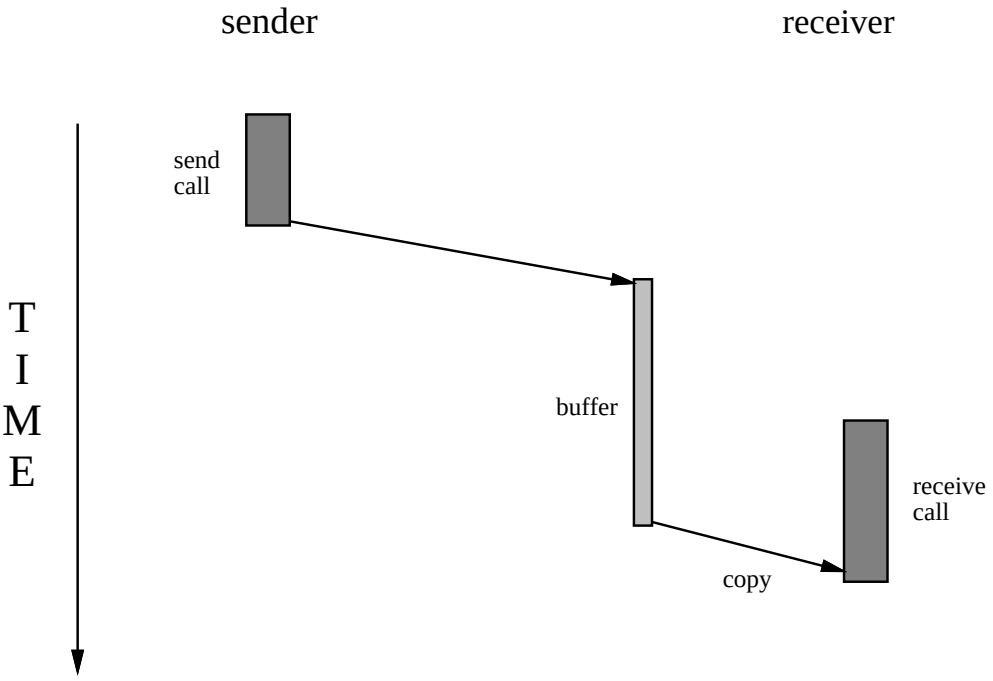

Figure 5.13: Delivery of a message in receiver-buffered mode.

the message body; synchronizing mode uses a rendezvous, which may require the sender to wait. Depending on the relative timing of the two processes, one mode or the other might be faster.

In terms of space, though, synchronizing mode is always better. Receiver-buffered mode requires that memory space be used to buffer the message, while synchronizing mode has no such requirement.

5.2.3 Buffering and Deadlock

The buffering-mode analysis must be sure that it does not, by its decisions, add a deadlock to the application program. To determine whether or not a program deadlocks, Parachute uses an *event ordering graph*, which summarizes all the known information about the time orderings between events in the execution of the communication pattern being analyzed [Lamport 78].

The event ordering graph is a directed graph. Each node in the graph represents an *event:* some instantaneous occurrence in the execution of the communication pattern. If there is a path from node i to node j, then event i happens before event j in any correct execution of the communication pattern; this *happens-before* relation is written $i \rightarrow j$. The communication pattern has a deadlock if and only if the event ordering graph has a cycle, that is, if and only if there is some x such that $x \rightarrow x$.

Each process's program generates a sequence of events. Specifically, each call to `beginSend`, `endSend`, `beginRecv`, or `endRecv` generates one event. Thus, each message induces four events, for the beginning and end of both the sending and receiving of the message.

There are three sources of edges in the event ordering graph:

1. Program-order edges represent orderings induced by the processes' programs. For each statement in a process's program, there is an edge from the node induced by that statement, to the node induced by the next statement.

2. Message-order edges represent orderings induced by each message. Intuitively, a message-order edge expresses the fact that a message must be sent before it can be received. For each message, there is a message-order edge from the node corresponding to the `beginSend` operation on that message, to the node corresponding to the `endRecv` operation on that message.

3. Rendezvous edges are induced by decisions about which buffering mode to use for various messages. These will be described in detail later.

Process P_0	Process P_1
`send message` $M1$ `to` P_1	`send message` $M2$ `to` P_0
`receive message` $M2$	`receive message` $M1$

Figure 5.14: An example communication pattern, whose event ordering graph appears in figure 5.15.

Figure 5.14 shows an example communication pattern, and figure 5.15 shows the event ordering graph corresponding to that pattern, before any buffering decisions are made. Note that the graph is acyclic, which means that the communication pattern is deadlock-free.

Once the event-ordering graph of the source communication pattern has been built, it can be used to determine the consequences of choices about whether to use synchronizing or receiver-buffered mode to transfer each message. Deciding to use receiver-buffered mode for some message has no effect on the event-ordering graph, since receiver-buffered mode does not use any synchronization between sender and receiver, other than that implied by the message itself.

However, using synchronizing mode does add a rendezvous, and so adds an edge to the event-ordering graph. Specifically, if message M is transferred in synchronizing mode, an edge is added from the node denoting M's `beginRecv` operation, to the node denoting M's `endSend` operation. This edge, together with the edge previously induced by M (according to item 2 of the list above), forms a "bow-tie" pattern, which says that the sending of M and the receiving of M must overlap in time. This expresses the fact that the send and receive operations rendezvous with each other. Of course, the newly added edge may cause a cycle in the event ordering graph, and hence a deadlock in the execution.

For example, assume that for the program of figure 5.15, the protocol compiler decided to pass message $M1$ in receiver-buffered mode, and $M2$ in synchronizing mode. This adds another arc (for $M2$) to the event-ordering graph, leading to the

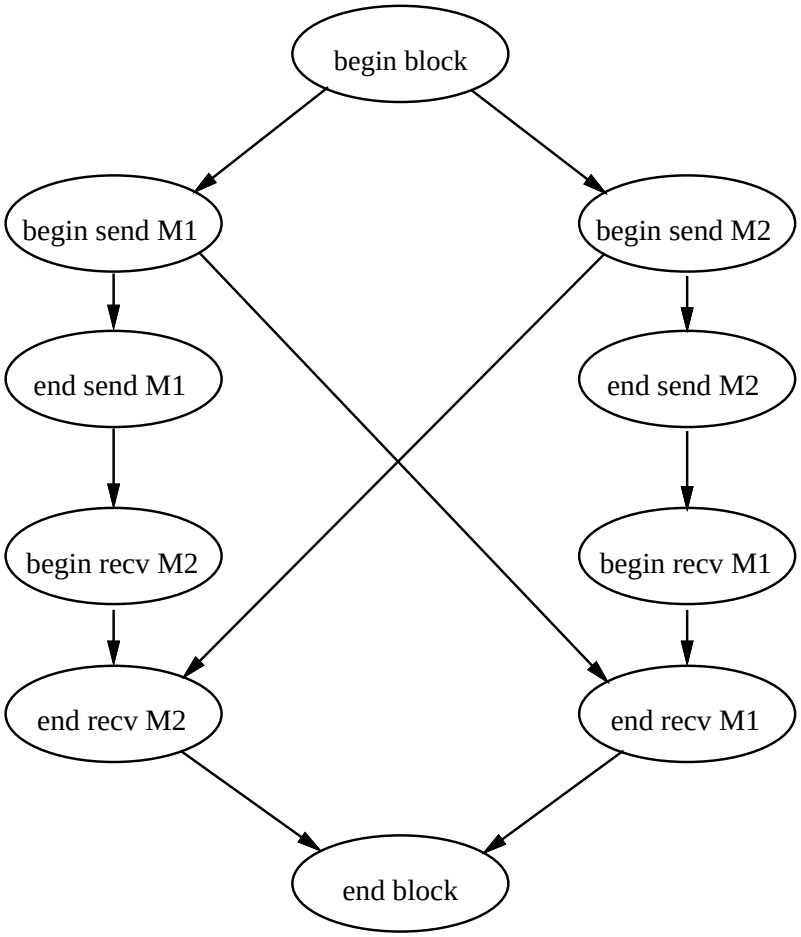

Figure 5.15: The event ordering graph for the communication pattern shown in figure 5.14.

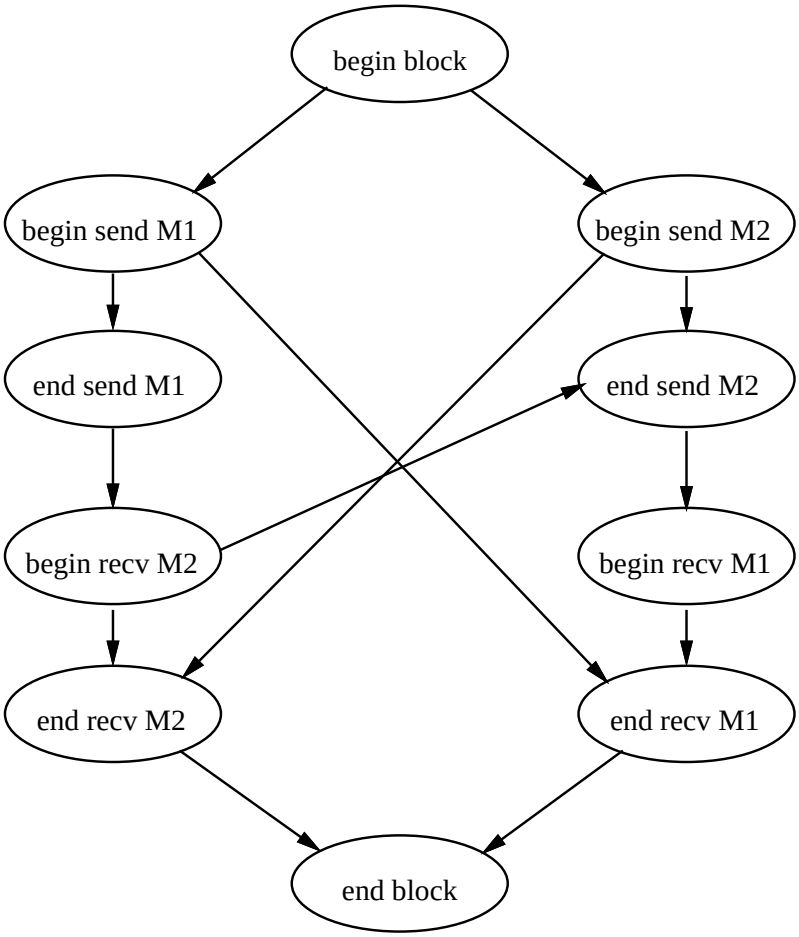

Figure 5.16: The event ordering graph of figure 5.15, under the assumption that message $M1$ uses receiver-buffered mode, and message $M2$ uses synchronizing mode.

graph shown in figure 5.16. Since this graph is still acyclic, this is a safe set of choices for the protocol compiler to make.

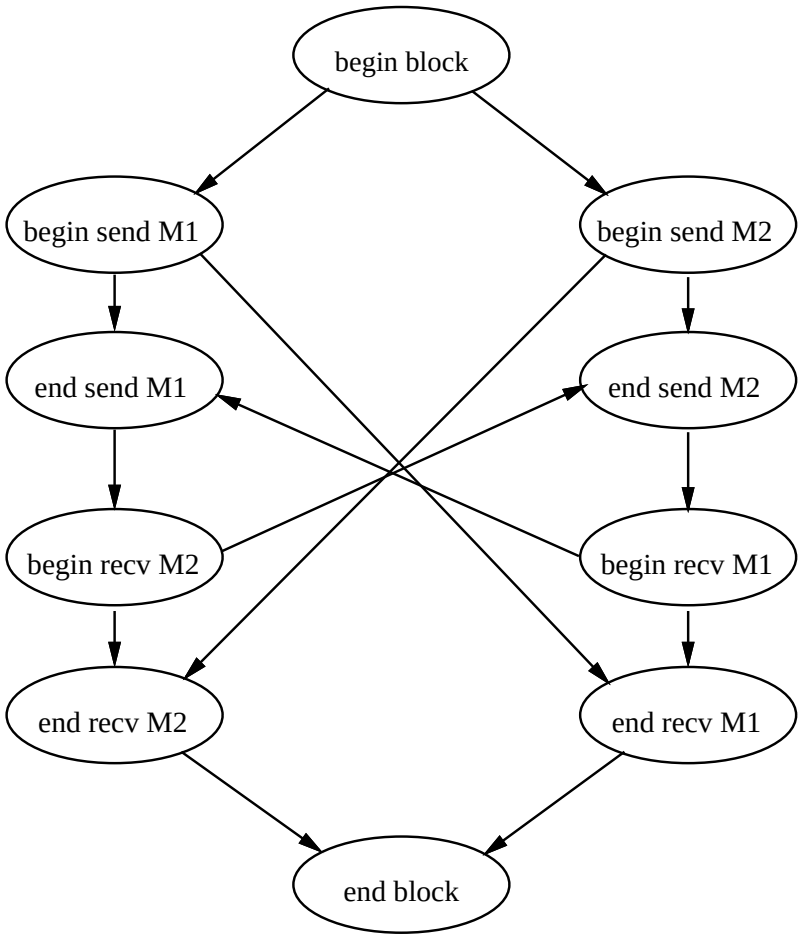

Figure 5.17: The event ordering graph of figure 5.15, under the assumption that messages $M1$ and $M2$ both use synchronizing mode. This graph has a cycle, indicating that the program will deadlock under these assumptions.

Now assume that the protocol compiler instead decided that both $M1$ and $M2$ should use synchronizing mode. This adds yet another arc to the event-ordering graph, leading to the graph shown in figure 5.17. This graph has a cycle, indicating that if these choices are made, the program will deadlock. Thus, the protocol compiler may not use synchronizing mode for both messages.

The fact that receiver-buffered mode is "safe" while synchronizing mode adds possible deadlock suggests a strategy for attacking the problem of assigning buffering modes to messages. We could assign a "synchronizing benefit" to each message, which would reflect the relative advantage of synchronizing over receiver-buffered mode for that message; this benefit would be negative if receiver-buffered mode were more attractive. We could then use an algorithm that assigned buffering modes to messages while maximizing the total benefit.

Unfortunately, there is no efficient algorithm for this problem. The problem of assigning buffering modes to maximize the sum of a per-message synchronizing benefit is NP-complete. (A proof of this fact is given in appendix B. The proof uses a reduction from the feedback arc set problem.) Since there is no efficient, optimal algorithm, even for a very simple benefit function, a heuristic must be used.

Parachute uses a simple greedy algorithm. Since deadlock can only be introduced by messages in synchronizing mode, all messages start out in receiver-buffered mode, then messages are individually changed to synchronizing mode, being careful on each change not to introduce a deadlock. Messages are first sorted into decreasing order of the benefit function. Then each message is considered in turn, from largest to smallest benefit, changing a message to synchronizing mode if it has positive benefit and if making the change does not introduce a deadlock.

5.2.4 Buffering and Performance

The remaining problem in buffering analysis is how to calculate the weighting function that expresses the benefit of making a particular message synchronizing rather than receiver-buffered. This is done by a heuristic, taking two factors into account:

- the cost of the extra message handling overhead in the synchronizing case, which is treated as a constant, and

- the (time) cost of buffering a message, which is treated as linear in the size of the message. This is consistent with the assumption that this cost arises mostly from data copying.

Thus, the benefit of choosing synchronizing over receiver-buffered, for a message of length L, is $cL - r$, for some constants c and r. This cost function does not take

into account the cost of the extra waiting in the synchronizing case. (Or, since the constant L is chosen empirically, the cost of waiting is treated as constant.) Dividing by c, we can further normalize the benefit function to be $L - L_0$ for some constant L_0. On the iPSC/860, L_0 is chosen as 8000 bytes. Thus, unless there is a danger of exceeding the limit on buffer space, Parachute will not attempt to treat a message as synchronizing unless it is longer than 8000 bytes.

Ideally, the choice of buffering modes would be made by doing a detailed timing analysis of the block being analyzed. The event-ordering graph would be augmented by labelling each node with the expected time at which it is executed. The analysis would then look for messages in which the receiver is ready before the sender, and change these messages to synchronizing mode. In addition, the analysis could consider both possibilities (synchronizing and receiver-buffered) for some messages that were particularly large or were on the critical path.

In practice, such analysis is usually impossible, because detailed timing information is not available at compile-time. In particular, the time taken by sequential computation phases is not known and cannot be predicted accurately. However, we could potentially gather such information by tracing the program.

5.2.5 Summary

There are several message-passing modes that can be used to transmit data from one processor to another. The modes offer different tradeoffs between protocol overhead, buffer space requirements, and the amount of compile-time analysis needed to guarantee correctness.

Parachute uses three modes. It uses blast mode whenever it is safe. For the majority of messages, it uses either receiver-buffered mode or synchronizing mode.

Parachute's choice between receiver-buffered mode and synchronizing mode is based on issues of correctness and performance. Handling a message in synchronizing mode introduces a time-ordering between events which may cause the program to deadlock; before choosing synchronizing mode for a message, Parachute must ensure that this deadlock cannot occur. Performance issues include the extra buffer space required by receiver-buffered mode, as well as the tradeoff between waiting in synchronizing mode and data copying in receiver-buffered mode.

The combinatorial optimization problem that arises from the choice between

receiver-buffered mode and synchronizing mode is NP-complete. Parachute uses a simple, greedy heuristic to address it.

5.3 Buffer Allocation

The last of our three major issues is buffer allocation. Certain message modes require the use of memory on some processor to temporarily buffer a message until the receiving process is ready for it. In order to use these modes, the protocol compiler must allocate buffer space and see that it is managed correctly.

There are two classes of buffer allocation schemes: static and dynamic. Static schemes allocate buffers for messages at compile-time, while dynamic schemes allocate buffers at run-time. Each scheme has its advantages and disadvantages. Standard message-passing packages, like Intel's NX, use dynamic buffer allocation[3].

Dynamic buffer allocation is more flexible and does not require any compile-time analysis. On the other hand, dynamic allocation requires a heavier-weight protocol such as those in section 2.4.2, since a processor can never be certain that some other processor will have buffer space available. In addition, dynamic allocation schemes have poor failure properties. Although a good dynamic scheme works for most reasonable programs, it may cause some programs to deadlock, and the conditions under which this deadlock might take place are difficult to specify precisely.

The main advantage of static allocation is that it allows a faster protocol, since the sender is assured that the receiver is prepared to store the message. In addition, when static schemes fail, they do so in a helpful way, reporting an error at compile-time, or perhaps even reconsidering some of the compiler's earlier decisions in an attempt to reduce buffering requirements. The drawbacks of static allocation are that it requires compile-time analysis, and this analysis must necessarily be conservative.

In practice, the protocol compiler does not have complete knowledge of what the application program will do, so it cannot use static analysis alone. Therefore it uses some mixture of static and dynamic allocation. Some message-passing modes require static buffer allocation. In particular, the receiver-buffered mode depends crucially on the availability of a buffer on the receiving processor. The only way to guarantee this is to allocate a buffer statically.

---

[3] Some systems support only synchronizing mode, and hence do no buffering at all. They are effectively forcing the programmer to do static buffer allocation by hand.

5.3.1 Static Allocation of Buffers

In general, static buffer allocation takes place in two stages. First, an upper bound is put on the amount of buffering the program requires. Second, buffer locations are allocated to messages in a way that re-uses buffers as much as possible in order to save space.

Bounding a Program's Buffer Requirements

Process P_0	Process P_1
$n \leftarrow 0$	$y \leftarrow 0$
while(someCondition()){	while(someCondition()){
$n \leftarrow n + 1$	receive x tag 0
send $f(n)$ to P_1 tag 0	$y \leftarrow g(y, x)$
}	}
	print y

Figure 5.18: A program that might use unbounded buffers.

Consider the program of figure 5.18. The two processes interact in a producer-consumer fashion. If the producer can send messages faster than the consumer can receive them, the amount of buffer space in use can grow. If nothing stops this process, the program may overflow any bounded amount of buffer space[4].

The solution is simple: we must transform the program to an equivalent form, adding sufficient synchronization, perhaps in the form of additional messages, to ensure that the producer cannot get ahead of the consumer by more than some bounded number of iterations. For example, we could make the program execute a barrier synchronization before every kth iteration of the loop, or we could require the consumer to send a flow-control message to the producer every k iterations.

Now consider the program in figure 5.19. P_0 sends a message for each iteration of its loop, and all of these messages must be buffered until P_0 exits the loop and

[4] It is tempting to classify such a program as erroneous. However, programmers write programs like this and expect them to work, so a reasonable implementation should allow them.

Process P_0	Process P_1
$n \leftarrow 0$	receive n tag 1
while(someCondition()){	$y \leftarrow 0$
$n \leftarrow n + 1$	for $i \leftarrow 1$ to n
send $f(n)$ to P_1 tag 0	receive x tag 0
}	$y \leftarrow g(y, x)$
send n to P_1 tag 1	}
	print y

Figure 5.19: A program that might require unbounded buffers.

sends the message with tag 1. Since we cannot put an upper bound on the number of iterations of P_0's loop, we cannot bound the amount of buffering required.

Note the difference between this program and the one of figure 5.18. In this case, the buffering requirement is an inherent property of the program, whereas in figure 5.18 the buffering requirement could be reduced without changing the result of the program.

It is unrealistic to expect the protocol compiler to execute *every* program, regardless of its buffering requirements. Indeed, no possible implementation can run a program whose buffering requirements exceed the available storage. The only choice is to guess optimistically that the actual execution will not use too much buffer space, and generate code under that assumption. If the guess turns out to be wrong, the run-time system can signal an error. Although the protocol compiler cannot run every possible program, it can at least hope to run every program that a conventional implementation could run.

Allocating Buffers to Messages

Once the buffering requirements of a program have been bounded, the protocol compiler can proceed to allocate buffers to the program's messages. The buffer allocation problem is roughly like the register allocation problem that compilers face. In both cases, a set of values must be stored in memory, these values have certain lifetimes,

and the allocation must avoid sharing storage between values that might be alive at the same time.

In the buffer allocation problem, space is allocated on a *per-send* basis, that is, a buffer is allocated for each `send` operation in each process's program.

There are two differences that make buffer allocation harder than the standard register allocation problem. First, the values being assigned storage may vary widely in size, depending on the maximum sizes of various messages. Second, in an asynchronous parallel program, a sending process may get ahead of a receiver; if the program contains loops, multiple messages from a single send-call might be alive at the same time.

The second difference is the more important of the two. The solution is to allocate extra space for a send-call, enough to store all the messages from that send-call that might be alive at any time. Since we know the total buffering requirements are bounded, there is some upper bound on the number of instances of each send-call that can coexist. Program analysis may enable a tighter bound to be found.

As in the case of register allocation, buffer allocation proceeds by first building a *conflict* graph that summarizes which send-calls can share storage and which cannot. The conflict graph is then used to produce an allocation of space.

The conflict graph contains one vertex for each send-call, and an (undirected) edge between two vertices if those two send-calls might possibly require storage in the same processor's memory at the same time. The protocol compiler must assume that two send-calls might conflict unless it can prove that they do not conflict.

Once the conflict graph has been determined, the protocol compiler must solve a kind of extended graph-coloring problem. Formally, this problem can be stated as follows:

> Given an undirected graph $G = (V, E)$, with nonnegative integer labels s_i on the vertices V, find integer values b_i for each vertex such that:
>
> 1. $b_i \geq 0$ for all $i \in V$, and
> 2. for all $(i, j) \in E$, either:
>   - $b_i + s_i \leq b_j$, or
>   - $b_j + s_j \leq b_i$,
>
>   and

3. $\max_{i \in V}(b_i + s_i)$ is minimized.

Here G represents the conflict graph, s_i is the size of the buffer required by send-call i, and b_i is the buffer offset assigned to send-call i. This problem is NP-complete — when all $s_i = 1$ it is exactly the standard graph coloring problem.

5.3.2 Buffer Allocation in Parachute

As in the other phases of analysis, Parachute takes advantage of the special structure of the programs it is analyzing to simplify its buffer allocation task.

Parachute's runtime library executes a barrier synchronization at the beginning of each communication pattern. This ensures that no process enters a pattern until all processes have finished previous patterns, which makes buffer allocation easier in two ways. First, it ensures that at most one message from each send-call can be active at a given time. Second, it provides a guarantee that messages from two send-calls in different communication patterns can never be active at the same time. Messages in separate communication patterns can always share storage, and therefore Parachute can solve the buffer allocation problem separately for each communication pattern.

Parachute builds a conflict graph for each communication pattern by examining the event-ordering graph that was produced by earlier phases of the analysis. If two messages A and B must be buffered at the same processor, there is a conflict between A and B unless either A's `endRecv` event happens before B's `beginSend` event, or B's `endRecv` event happens before A's `beginSend` event.

Parachute uses a nonoptimal, but fast, greedy algorithm to solve the buffer allocation problem. First, the messages are sorted in order of decreasing size s_i. Then proceeding from largest to smallest s_i, each message is assigned the smallest buffer offset b_i which does not cause a conflict with the messages that have been assigned so far.

Figure 5.20 shows an example. The conflict graph on the left side of the figure shows which pairs of messages can share buffers. The solution found by Parachute's greedy algorithm is shown on the right side of the figure.

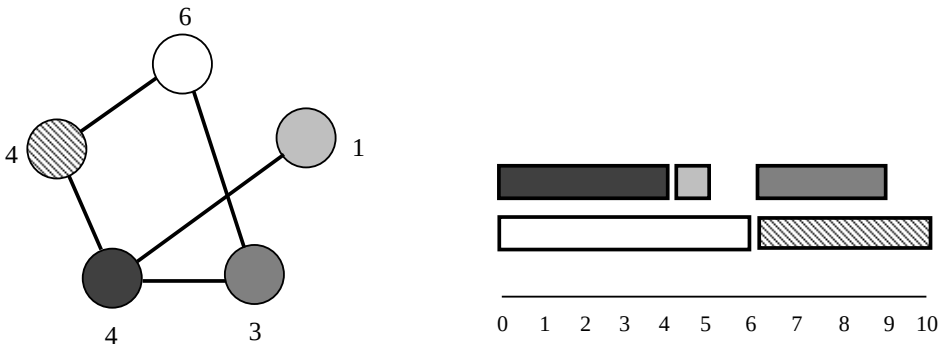

Figure 5.20: An example buffer allocation problem and its solution. The conflict graph on the left shows a set of five messages that must be buffered on some processor in some communication pattern. The graph has one node for each message; each node is labelled with the maximum size of the corresponding message. An edge between two nodes means that those two messages cannot share buffer space. The diagram on the right shows the solution, which satisfies the 18 words of total buffer requirements in 10 words of buffer space.

5.3.3 Meeting the Space Limit

The analysis described so far is satisfactory, except that it may sometimes use too much buffer space. In practice, there is a limit on the amount of buffer space that the tailored protocol may use. This limit can be stated explicitly by the programmer, or it can be inferred from the size of the generated program, or a default value can be used.

If the analysis as described so far yields a solution that meets the space limit, that solution is used. If, on the other hand, the result exceeds the space limit, corrective measures must be taken. Decisions made in earlier phases of the analysis can be reconsidered, with the new goal of reducing the amount of buffering required.

In the case of Parachute, the analysis of the offending communication pattern is done again, with the L_0 parameter of the buffering analysis reduced. (Recall that L_0 is the "break-even" message size; messages L_0 bytes in length or larger are put in synchronizing mode if possible.) The algorithm tries new values of L_0, trying to find the largest value which allows the space bound to be met. This is done via a binary search between zero and the original value of L_0.

If the space bound cannot be met, even for $L_0 = 0$, Parachute gives up, outputs the minimum-space solution it has found, and issues a warning to the user.

5.3.4 Summary

Allocating buffer space for messages is the third major task of the protocol compiler. In general, the protocol compiler must put an upper bound on the amount of buffer space a program requires, and then must perform the buffer allocation.

Buffer allocation is similar to the standard register allocation problem, except that in buffer allocation, storage may be allocated for objects whose sizes differ widely.

Parachute's buffer allocation algorithm first builds a conflict graph that summarizes all the available information about which messages can share buffers. The algorithm then uses this conflict graph to assign buffer addresses to messages, using a greedy heuristic.

5.4 Complexity of Parachute's Algorithms

This section calculates the time complexity of Parachute's algorithms, as a function of two parameters: the number of processes P, and the number of (static) messages M. The complexity is calculated for a single communication pattern; multiplying by the number of patterns in the program gives the total complexity of the algorithms running on an entire application program. Since each process participating in a communication pattern must handle at least one message, we know that $P \leq M$. Also, since all statements refer to some message, and there are at most four statements referring to any message, we know there are $O(M)$ statements executed in the pattern.

The following subsections consider each phase of Parachute's execution. Adding up these contributions, we will see that the protocol compiler's overall running time is $O(M^2)$.

5.4.1 Message Matching

If implemented carefully, the message matching phase requires $O(M)$ time. This can be demonstrated by "charging" some message for every operation performed by the algorithm, and then arguing that no message can be charged for more than some constant amount of work.

Two implementation techniques are needed to bound the complexity of this phase. First, the algorithm must maintain a "ready-list" of processes that are possibly ready to execute a statement. When a process reaches the end of its program, or tries to receive a message that hasn't yet been sent, that process is removed from the ready-list. Sending a message to a process that is not on the ready-list causes that process to be returned to the ready-list. The ready-list ensures that a process cannot try and fail to execute a particular statement too many times.

The second technique is to store the list of pending messages carefully. By doing this, we can ensure that the algorithm can always find the first pending message that matches some `endRecv` statement in $O(1)$ time. Using FIFO linked lists is a simple way to guarantee that messages from a given sender to a given receiver will be delivered in order. Each pending message is a member of two doubly-linked lists:

1. a list of all pending messages with the same destination process, and

2. a list of all pending messages with the same destination process and message tag.

Since a receiving process can either accept any message tag or specify a particular tag, these two lists correspond to the two ways in which an `endRecv` operation can attempt to match a message. The algorithm maintains a header for each list; these headers are stored in a hashtable.

If the hashtable is implemented properly, finding the header for any list requires $O(1)$ expected time. Thus, a message can be added to the pending message list, or the first message matching some `receive` can be found and removed, in $O(1)$ expected time.

Operations are charged to messages as follows: The time to execute an operation is charged to the message that operation applies to. The time to remove a process from the ready-list as a result of a blocked `endReceive` is charged to the message eventually received by that `endReceive` call. The time to return a process to the ready-list is charged to the message whose sending causes that process to be returned. Per-process start-up and shut-down costs are charged to the first message the process sends or receives. Since each message can be charged at most a constant number of times, and each charge adds only a constant to that message's account, the total charge for each message is $O(1)$, and hence the total execution time is $O(M)$.

5.4.2 Buffering Analysis

This phase performs a series of steps. First, it builds an event-ordering graph and checks it for cycles. Second, it generates a list of messages, sorted by size. Third, it considers each message in turn, deciding whether or not to make each one use synchronizing mode. Making a message synchronizing causes the event-ordering graph to change.

The event-ordering graph can be built in $O(M)$ time, since the outdegree of each node in the graph is at most two: one for a program-order edge to the next statement, and one for a possible message-order or rendezvous-order edge.

Since each node has $O(1)$ outdegree, we can check for a path between some pair of nodes a and b in $O(M)$ time by depth-first search. Thus, the analysis can check for a cycle in the newly-constructed graph in $O(M^2)$ time by checking for a (nontrivial) path from each node to itself.

Sorting the list of messages into order of decreasing size requires $O(M \log M)$ time.

In its final step, the buffering analysis phase considers each message in turn, and decides whether or not to make it synchronizing. Determining whether or not a message can be synchronizing requires testing for the existence of a path between two specific nodes in the event-ordering graph, which takes $O(M)$ time. If the message is made synchronizing, the event-ordering graph is updated, which takes $O(1)$ time. Since this entire procedure might be repeated for each message, this step could require $O(M^2)$ time.

Adding up the times required by these three steps, we see that the buffering analysis phase requires $O(M^2)$ time.

5.4.3 Buffer Allocation

The buffer allocation phase builds a conflict graph for each process, and then assigns a buffer address to each message. In the worst case, all the messages sent in the communication pattern are destined for the same process.

Building the conflict graph requires $O(M^2)$ time. First, the algorithm calculates the transitive closure of the event-ordering graph. This is done by doing a depth-first search from each node in the event-ordering graph. Since there are $O(M)$ nodes, and the search takes $O(M)$ time, this calculation takes $O(M^2)$ time.

Assigning buffer addresses requires a doubly-nested loop over the nodes in the conflict graph, and hence requires $O(M^2)$ time.

5.4.4 Cycling to Satisfy the Space Bound

Recall that if the buffer allocation step is unable to buffer the required messages in the available buffer space, the algorithm cycles back to the buffer analysis phase, while changing the benefit function to discourage buffering. This cycle might repeat several times before the space bound is met, or the protocol compiler determines that it cannot meet the space bound. At first glance, it appears that this would increase the complexity of the protocol compiler.

In reality, the worst-case asymptotic complexity is not increased by this cycling. Recall that when cycling, the protocol compiler tries to find the largest value of the L_0 parameter which allows the space bound to be met. Since the first, failed, attempt

used a fixed value of L_0 (call this value L_{orig}), and using $L_0 < 0$ doesn't make sense, the protocol compiler can do a binary search between the bounds $0 \leq L_0 < L_{orig}$. This binary search requires $O(\log L_{orig})$ iterations. But since L_{orig} is a constant, the number of iterations is $O(1)$. Therefore, the binary search multiplies the total complexity by $O(1)$, which leaves the total complexity as $O(M^2)$.

5.5 Summary

A protocol compiler's analysis must deal with three main issues: nondeterminism, buffering modes, and buffer allocation.

Nondeterminism in message-passing programs arises from tag-matching. The protocol compiler can reduce or eliminate nondeterminism by retagging: changing the tags used by the application's communication calls. Reducing nondeterminism in this way simplifies the tasks of later phases of analysis.

A message can be transmitted from sender to receiver via one of several modes. Different modes offer different tradeoffs between performance and the amount of compile-time analysis required to guarantee correctness. A protocol compiler must choose between modes on the basis of correctness and performance; making the right choice might depend on having an accurate model of the target architecture.

Some buffering modes require that buffer space be allocated for messages at compile-time. This leads to the buffer allocation problem. Allocating buffers for messages is similar to the standard compiler problem of allocating registers for variables. The main difference is that messages come in many different sizes. The buffer allocation problem leads to a modified graph coloring problem.

Parachute handles these three issues by taking advantage of the special structure of the application programs it accepts. Parachute uses symbolic execution to remove all nondeterminism from the source program. Both buffering-mode analysis and buffer allocation lead to NP-complete problems, on which Parachute uses greedy heuristics. For a communication pattern with M messages, Parachute's analysis requires $O(M^2)$ worst-case time.

Chapter 6

PARACHUTE: THE PROTOTYPE

Pandora's Pox: Nothing ever works the first time, but there is always hope that this time is different.

— *Gerald M. Weinberg*

This chapter describes the remaining details of Parachute, the prototype protocol compiler. Parachute generates tailored protocols for communication patterns. These protocols execute on the Intel iPSC/860 multicomputer, which runs the NX operating system. The next chapter will present experiments that evaluate the performance, for real application programs on real machines, of tailored protocols generated by Parachute.

Parachute consists of about 2500 lines of C code, plus about 300 lines of `lex` and `yacc` source. Parachute's protocol compiler reads a communication description file, which summarizes the communication patterns in the application program. Parachute produces as output a tailored protocol. This output is actually C source code, consisting of declarations of a set of data structures specifying the tailored protocol; it is compiled and linked with a run-time library. The run-time library, which is described in detail in section 6.3, uses these data structures to carry out the tailored protocol.

6.1 Using NX

Parachute runs on top of the native NX operating system. Building on top of NX introduces some overhead into the protocols, since NX does not conveniently provide all the hooks that tailored protocols need. However, the decision to build on NX had two important benefits. First, it allowed me to use all the NX features not having to do with communication. This made it easier to port existing applications to Parachute, and also enabled the use of the standard NX tools for debugging and performance monitoring. In addition, this choice allowed ordinary NX message-passing to co-exist

with optimized communication. Thus, it allowed the protocol compiler to be used for only part of a program if that was desired.

Second, by comparing NX to Parachute running on top of NX, I was able to isolate the effect of improving the communication protocol. When Parachute was used, all the details of data movement and protection-boundary crossings were handled by NX. This means that any differences in performance between Parachute and NX could not have been caused by differences in the efficiency of data movement or crossing protection boundaries. Differences in performance must be due to differences in protocol overhead. Using NX as a basis allowed me to cleanly measure the speedup caused by Parachute.

NX has a mechanism called "force-type messages" that allows applications to move data without NX imposing a protocol. Parachute uses force-type messages to move data, thus avoiding any protocol overhead within NX. Parachute also uses NX's hand-coded `gsync` procedure to perform barrier synchronizations.

6.2 User's View of Parachute

Figure 6.1 shows how Parachute works from the user's point of view. To adapt an existing message-passing program to use Parachute, the programmer or compiler must do two things: add annotations to the program to mark the beginning and end of communication patterns, and make a communication description file. The protocol compiler reads the communication description file and generates code for a tailored protocol. This is compiled and linked with the application program and a runtime library to get a runnable image.

The programmer or compiler adds annotations to the application program to mark the beginning and end of each communication pattern. Patterns are denoted by integer identifiers; the statement `beginPattern(6)` marks an entry to pattern number 6, and `endPattern(6)` marks an exit from pattern 6.

6.2.1 The Communication Description File

The communication description file encapsulates all the information the protocol compiler needs to know to make its decisions. In principle, this information could be gleaned from an analysis of the source code of the application program, but for

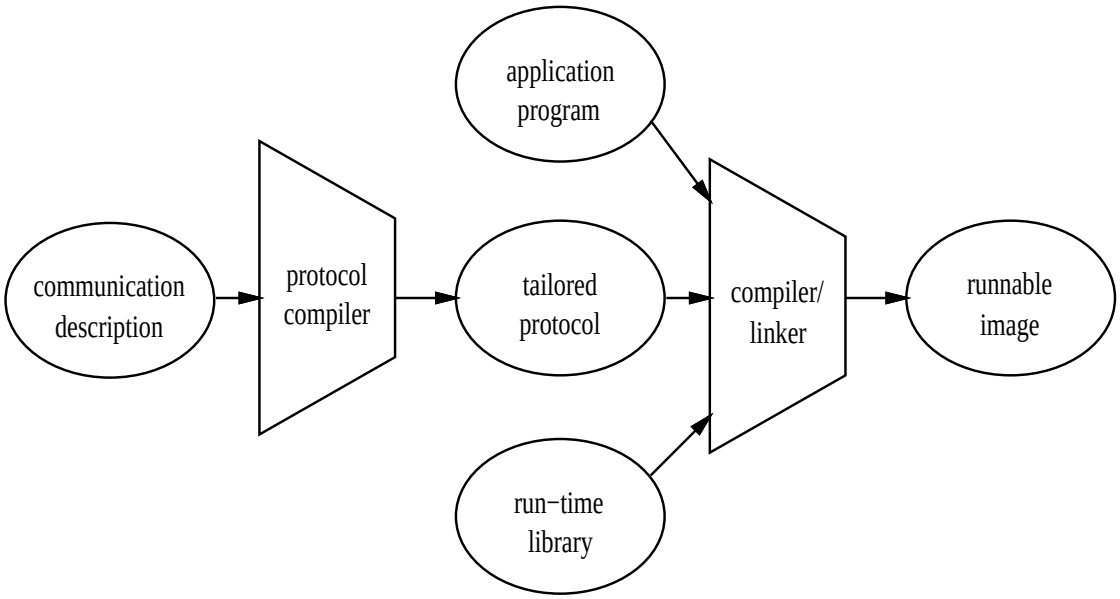

Figure 6.1: The user's view of how Parachute works. The communication description file is fed through the protocol compiler to produce C code for the tailored protocol. This is then compiled and linked with the application program and a runtime library to make a runnable image.

Parachute I chose the simpler alternative of requiring a separate description file.

Appendix A contains a formal grammar for the PDL language used in the communication description file; this section will give an informal treatment. The communication description file has two parts: directives, and pattern descriptions. The directives section says how many processes will execute the program, and how much memory space the tailored protocol may use. The pattern descriptions section lists all the patterns, and the behavior of each process within each pattern, as described in section 4.3.3.

There are three ways the communication description file can be generated. First, the programmer can write it. This is straightforward but tedious.

Second, the communication description file can be generated automatically from a trace of the program. Once the source program has been annotated to mark the pattern boundaries, the program can be run in a mode that generates a trace file describing the exact sequence of communication calls (and annotations) executed within each pattern. This trace file can then be post-processed to generate the communication description file. This is the approach I used in my experiments; it requires almost no work beyond annotating the source program. Once the annotations have been added, the rest of the procedure can be automated using `make`.

Note that, by assumption, the trace of each communication pattern generated by this procedure is data-independent. This is because the programmer or compiler, by annotating the communication patterns in the program, asserted that the communication within those patterns was not data-dependent. If this assertion was in error, that fact will be caught at run-time and signalled as an error.

Finally, the third method of generating the communication description file is to rely on the compiler. If the source message-passing program is generated by such a compiler, this compiler could be induced to create the communication description file. The information in the communication description file is probably known to the compiler at one time or another, so this is merely a matter of writing the information out to a file in the correct format. Alternatively, the compiler could merely mark pattern boundaries in the generated message-passing code; then the tracing method could be used to generate the communication description file.

6.3 Parachute at Runtime

The final component of Parachute is its runtime library. The library does all the bookkeeping necessary for running the tailored protocol, and provides the low-level communication services that tailored protocols need.

The library contains all the code that executes in every tailored protocol. The protocol compiler does not actually generate executable code, but merely creates C code that declares a set of data structures that are used by the code in the library. The contents of these data structures are interpreted by the library code to carry out the appropriate actions for each message.

When considering how the library works, some confusion may arise between the `send` and `receive` calls made by the application program, and the NX `send` and `receive` operations performed by the library to accomplish the tailored protocol. Throughout this section, I will refer to calls made by the application program as "user-level" calls. Any calls not designated as user-level should be assumed to be NX calls made by the library.

6.3.1 Synchronization

Parachute's runtime library executes a barrier synchronization at the beginning of each communication pattern. This synchronization has two purposes. First, it simplifies the buffer allocation task by effectively isolating the patterns from each other so that they can be analyzed separately. Second, it enables a process to execute per-pattern initialization, where necessary, without any danger of other processes entering the pattern prematurely.

These barrier synchronizations add some overhead at run-time. However, this is a small effect, since patterns tend to be large, so the cost of synchronization can be amortized over many messages. Section 8.2.5 discusses methods for reducing the number of synchronizations.

6.3.2 The State Machine

The application program uses a user-level interface that does not mark each message specially. Yet the tailored protocol must determine which message each communication call pertains to, so it can treat that message in the special way that the protocol

compiler decided on. The mapping between these levels is done by using a state machine managed by the library. The state machine keeps track of which communication pattern is currently being executed (if any), and exactly where in the execution of that pattern the program is.

In each state of the state machine, some subset of the possible user-level communication calls is legal. In other words, the application program may only make calls that conform to the communication pattern that it is in. The state machine allows the library to detect when the application program violates these constraints. If the program makes an "illegal" user-level communication call, a run-time error is reported to the user. Checking for this error costs only a few instructions per communication call.

The state machine has one state for each user-level communication call made by each process in each communication pattern. There is another, distinguished state that denotes that the program is not in any communication pattern. When a process enters a communication pattern, it looks up in a table which state to enter. (States are identified by integers.)

During the execution of the pattern, the state-identifier of a process is incremented each time the process makes a user-level communication call, thus "counting off" the required user-level communication calls within that communication pattern. When the end of the communication pattern is reached, the state machine reverts to its "no-pattern" state.

6.3.3 The Library's Data Structures

The protocol compiler outputs a set of data structures that are used by the library. These structures include the following:

- a *message descriptor* for each message sent or received within any pattern. A message descriptor identifies the message's sender, receiver, tag, maximum size, buffer location, and which message-passing mode it uses. In addition, the message descriptor has a few slots used at runtime to keep track of the status of the message.

- a *state descriptor* for each state of the state machine. This descriptor says which communication call the application program is allowed to make in that state,

and which message that call pertains to.

- a *pattern descriptor* for each communication pattern the program executes. This descriptor contains the initial state-identifier each process should use when entering that communication pattern, as well as information about which states and which messages are part of the pattern.

- *mapping tables* that translate back and forth between the integer pattern-identifiers used by the application program, and those used internally by the library.

6.3.4 Moving Data Without a Protocol

The library uses an NX feature called "force-type messages" to move data between processors without a communication protocol. An NX message is designated as force-type by giving it a message tag in a certain range. The data of a force-type message is sent immediately through the network by the sending processor. When the message data arrives at the destination processor, if some process on the destination has issued a matching `receive` call, the data is given directly to that process. If, on the other hand, no matching `receive` has been issued, then NX silently drops the message.

In order to use force-type messages, we must ensure that the receiving process has issued its `receive` call before the sender issues its `send`. Appendix C explains in detail how this is done. Separate mechanisms are used to implement synchronizing and receiver-buffered messages on top of force-type messages.

In order to uniquely identify each message in transit, the library uses a unique message tag for each message the protocol compiler knows about. This tag is assigned by the protocol compiler.

6.3.5 Handling Synchronizing-Mode Messages

Synchronizing messages involve a rendezvous between sender and receiver. This rendezvous is initiated by the receiver. When the receiving process issues its user-level `receive` operation, the library code on the receiver's node issues a non-blocking NX `receive` call to accept the message data, then transmits a small packet to the sender using the NX `csend` call, which uses a protocol optimized for small messages. This

packet causes a bit to be set which informs the sender of the receiver's readiness. Once the sender has issued its user-level `send` operation, and the ready-bit is set, the sender clears the ready-bit and then uses an NX force-type message to transmit the data to the receiver.

6.3.6 Handling Receiver-Buffered Messages

A receiver-buffered message is somewhat simpler. Recall that the protocol compiler allocated a buffer for each receiver-buffered message. At the beginning of the program's execution, each process issues a set of non-blocking NX `receive` operations, one for each receiver-buffered message that process will receive. After issuing these `receive`s, the processes perform a barrier synchronization, to ensure that all processes have had a chance to issue their `receive`s before any messages are sent to them. Whenever one of these NX `receive`s completes, the process re-issues it.

When a process issues a user-level `send` operation on a receiver-buffered message, the sender simply issues an NX forced-type `send` operation to transmit the data to the receiver. This is safe since the receiver is known to have previously issued a matching NX `receive`.

Appendix C shows pseudocode for the runtime library's implementation of the MP0 communication calls.

Chapter 7

PERFORMANCE

First get your facts; and then you can distort them at your leisure.
— Mark Twain

This chapter uses experiments to evaluate the performance consequences of using a tailored protocol rather than standard data-movement techniques. In addition to simply measuring the performance of Parachute, I will attempt to address the more general issue of how valuable the protocol-compiler approach is.

To determine the benefit of using a protocol compiler, we must answer three questions.

- What fraction of their time do applications spend in communication?
- What fraction of the communication time is protocol overhead?
- What fraction of the protocol overhead can be eliminated by using the protocol compiler?

Combining these three factors yields the decrease in running time due to the protocol compiler.

To address these questions, I performed experiments on six programs, using Parachute. Of the six programs, one is a micro-benchmark: a small, "toy" program designed to cleanly measure a particular feature of the implementation. The other five are macro-benchmarks: real computational-science applications.

All timing results reported for the benchmarks were measured on a 16-processor iPSC/860, using the PGI `icc` and `if77` compilers with maximum optimization level. Times were measured using a hardware timer with one-microsecond granularity. All times were measured separately on all processors, and then averaged.

The micro-benchmark is called `pong`. The `pong` application consists of a single communication pattern that repeatedly "ping-pongs" a message between a pair of processors. The communication operations are repeated many times to make up a communication pattern; this ensures that per-pattern costs have minimal effect on the results.

There are three versions of the `pong` micro-benchmark, one for each of three message-passing modes. The `native-pong` version uses the native NX communication library directly. The `buf-pong` version uses receiver-buffered mode for all messages. The `sync-pong` version uses synchronizing mode for all messages.

The micro-benchmark experiments consisted of running all three versions of `pong`, for various message sizes.

The macro-benchmarks consisted of these five application programs:

- *Matmult:* This program multiplies two matrices, using the Fox/Hey/Otto algorithm [Fox et al. 88, Fox et al. 87]. This algorithm lays out the program's data, and orchestrates its communication, very carefully. Most of this application's communication consists of broadcasts of submatrices across rows of the machine, and vertical pipelined "rolls" of data. This program was written by David Walker at Caltech.

- *Nbody:* This program simulates the evolution of a set of stars, interacting through gravity. It uses the simple n^2 algorithm, in which each timestep requires all-to-all communication. All-to-all communication is implemented with a "bucket brigade" technique [Fox et al. 88]; the program uses double-buffering to overlap communication and local computation. I wrote this program.

- *Olfactory:* This application simulates the olfactory cortex of a cat's brain. The program does a biologically accurate simulation of the behavior of each neuron, and the connections between them. This program was originally written by Jim Bower of the Caltech Biology Department and colleagues [Bower et al. 88], and has been updated by Wojtek Furmanski and Roberto Battiti of the Caltech Concurrent Computing Program.

- *Md:* This program carries out a molecular dynamics simulation. It simulates a set of molecules interacting via a Lennard-Jones force law. This program was

written by Steve Plimpton at Sandia National Laboratory [Plimpton 90].

- *Eigen:* This program implements a Jacobi eigensolver. This forms one important part of a larger quantum chemistry code. This program was written by Rik Littlefield and colleagues at Battelle Pacific Northwest Laboratory [Littlefield & Maschhoff 93].

Each application was run using the problem size suggested by the person or organization that supplied the benchmark.

Table 7.1: Characteristics of the macro-benchmarks.

application	lines	language	data size	code size	running time [sec]
matmult	690	C	205k	218k	0.74
nbody	242	C	74k	218k	1.59
olfactory	1161	C	4k	256k	0.56
md	1288	Fortran	3k	319k	0.28
eigen	4799	Fortran	20k	572k	3.17

Table 7.1 shows some static characteristics of the benchmark programs. They vary in size from 250 lines to almost 5000 lines, and are written in both C and Fortran.

Table 7.2: Communication statistics for the macro-benchmarks.

application	messages sent	avg message size	smallest	largest
matmult	187	7668	4	12800
nbody	240	2304	2304	2304
olfactory	7147	452	16	1024
md	1280	720	196	1540
eigen	2619	251	1	10000

Table 7.2 shows some dynamic properties of the application programs. Note that the average message size varies by more than a factor of 30; there is further variation within each benchmark.

7.1 Fraction of Time Spent Communicating

The first task is to evaluate what percentage of running time is spent in communication. This was determined by instrumenting the macro-benchmark programs to measure the amount of time spent in communication calls.

Table 7.3: Percentage of time spent in communication.

application	percent communication
matmult	13.3%
nbody	13.0%
olfactory	78.3%
md	24.4%
eigen	89.8%
average	43.8%

Table 7.3 shows the percentage of time spent in communication for the five macro-benchmarks. On average, these applications spend nearly half of their time communicating.

One caveat is required in interpreting this data. If a program suffers from load imbalance, processors waiting for others to reach a synchronization point will spend their extra time in the NX `msgwait` call, waiting for a message to arrive. This waiting time will be reported as time spent in communication, thus inflating the communication time. There is no simple quantitative way to separate this waiting from the communication time. Fortunately, examining trace files reveals that all of the benchmark programs exhibit nearly perfect load balance, with the exception of `eigen`. The traces indicate that about 70% of `eigen`'s time is spent in load imbalance. Still, even if `eigen` spent *no* time in communication, the average of the five benchmarks would show 27% of the time spent in communication.

7.2 Magnitude of Protocol Overhead

The next major task is to determine how much protocol overhead real programs experience. It is impossible to remove all protocol overhead from a program; unfor-

tunately, it is also impossible in practice to isolate protocol overhead from essential communication. Since protocol overhead cannot be measured directly, I will use a performance model to predict how large it is.

The parameters of the performance model are based on the micro-benchmark experiments. The model assigns a cost to each message, and simply sums these costs to estimate the total communication cost of a program.

This model ignores two important effects: load imbalance and communication overlap. Load imbalance arises when a processor becomes idle waiting for communication; the model ignores the effect of communication time on the amount of load imbalance. The model also ignores the possibility that communication cost can be hidden by overlapping it with computation, or with other communication.

Despite the model's shortcomings, it will prove useful in predicting the amount of protocol overhead that programs experience. Constructing a highly accurate performance model for a modern multicomputer would require a dissertation in itself!

7.2.1 Modeling Protocol Overhead

To build the model, I measured the running time of all three versions of the `pong` program, as a function of message size. The results of these experiments are shown in figures 7.1 and 7.2. Latency is roughly linear in message size, except for the discontinuity in the `native` version at a message size of 100 bytes. This jump in latency is due to NX/2 switching protocols; NX/2 uses a sliding-window protocol for messages less than 100 bytes in length, and a pre-reservation protocol for longer messages.

I then created a linear model for communication time in each mode, by doing a least-squares fit to the measured data. (The `native` data was modeled by separately fitting lines to the data from each of the two regimes.) The resulting formulae give the message-passing latency in microseconds as a function of the message size, n:

$$T_{native}(n) = \begin{cases} 108.39 + 0.326n & \text{if } n \leq 100 \\ 233.96 + 0.358n & \text{otherwise} \end{cases}$$

$$T_{buf}(n) = 40.45 + 0.407n$$

$$T_{sync}(n) = 224.64 + 0.357n$$

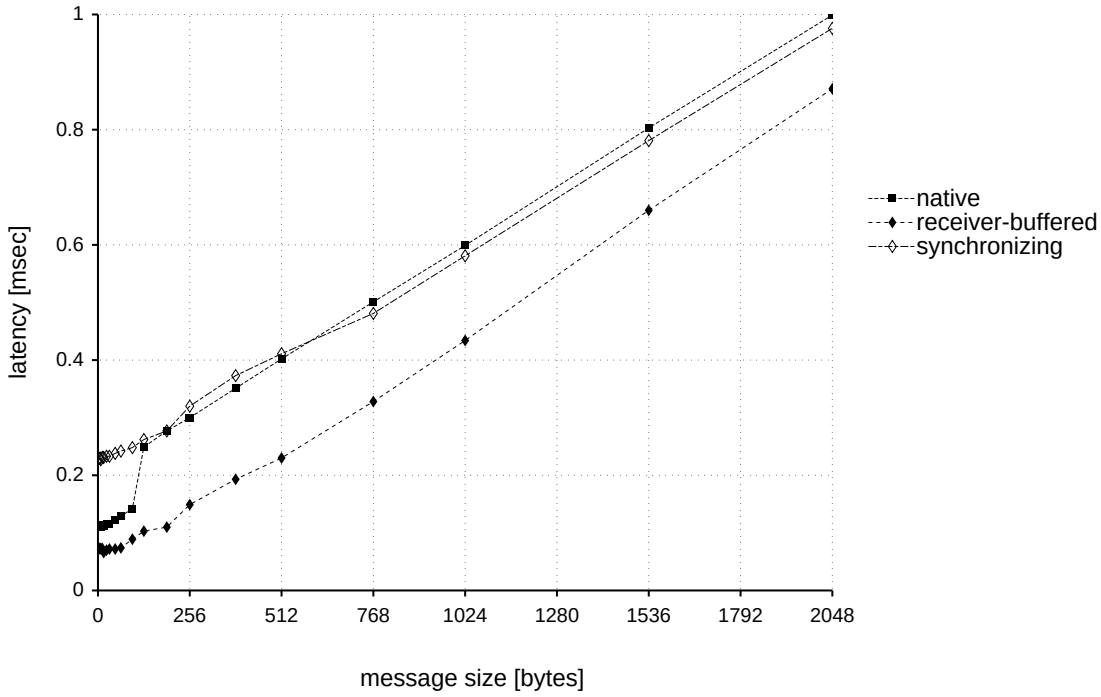

Figure 7.1: Message latency for three versions of the `pong` program.

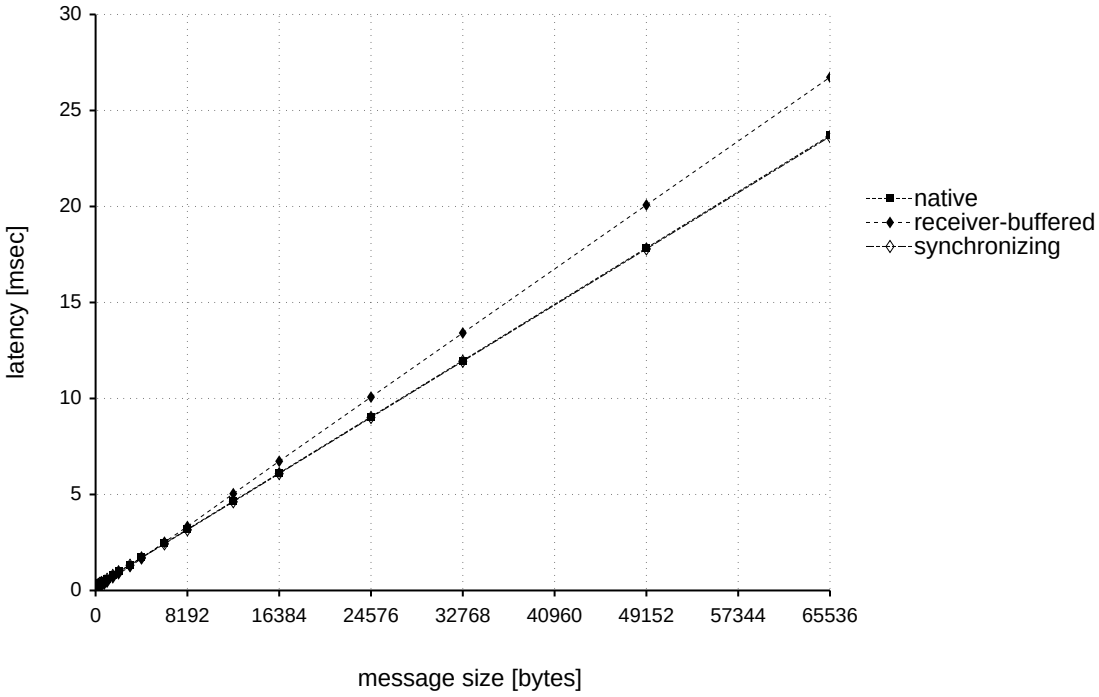

Figure 7.2: Message latency for three versions of the `pong` program.

To estimate the essential cost of communication, we can use the known parameters of the iPSC/860 hardware: the latency L is 25 microseconds, and the peak bandwidth B is 2.8 megabytes per second.

$$T_{essential}(n) = L + \frac{n}{B} = 25 + 0.357n$$

Subtracting the essential time from the native time, we can find the amount of protocol overhead in NX as a function of message size.

$$T_{protocol}(n) = T_{native}(n) - T_{essential}(n)$$

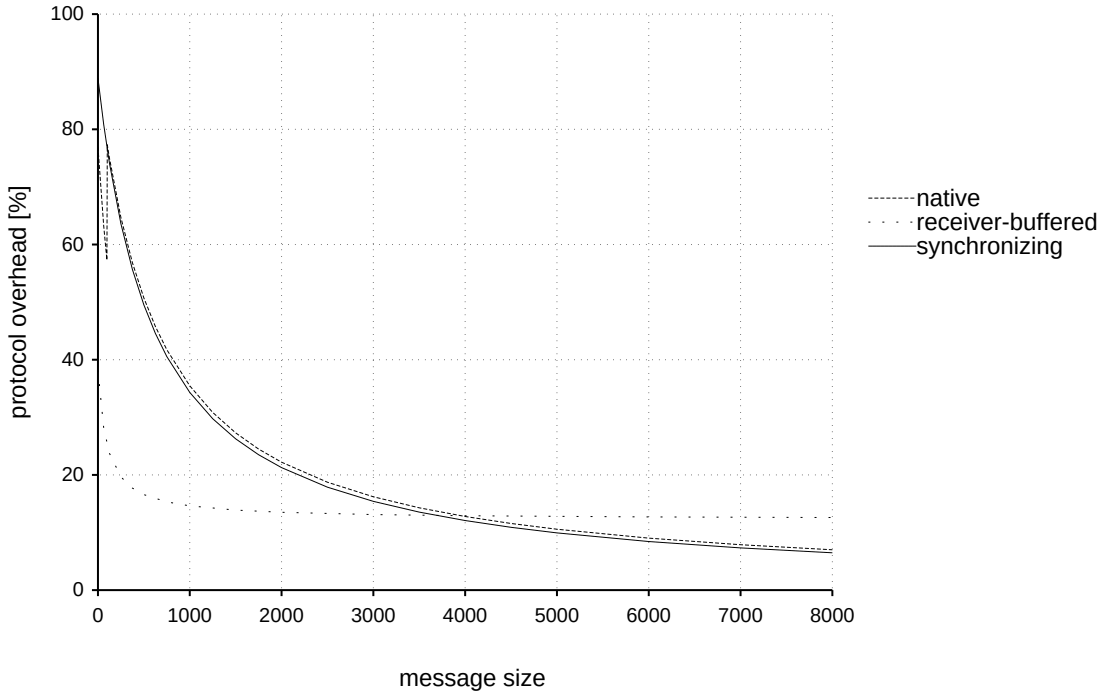

Figure 7.3: Percentage of communication time spent in protocol overhead.

Figure 7.3 shows the protocol overhead as a fraction of communication cost. Protocol overhead is a significant factor, except in the case of very large messages, where

communication time is dominated by physical bandwidth requirements.

Table 7.4: Predicted protocol overhead, as a percentage of communication time, for the five macro-benchmarks.

application	percent overhead
matmult	5.7%
nbody	19.9%
olfactory	52.3%
md	42.6%
eigen	44.3%
average	33.0%

Table 7.4 shows the model's prediction for the five macro-benchmarks. The model's predictions were calculated by instrumenting the message-passing library to add up the predicted cost under the native communication model, and the predicted amount of protocol overhead, as the program ran. Protocol overhead accounts for about one-third of the communication cost.

7.3 Reduction in Protocol Overhead

The third major question is how much protocol overhead can be reduced by using a protocol compiler. In general, we know that it is impossible to eliminate protocol overhead entirely. On the other hand, using a protocol compiler will not increase the protocol overhead, since the protocol compiler is always free to use the previously existing protocol if that is the best choice.

We can predict the reduction in protocol overhead by using the performance model developed in the previous section. Assuming that the protocol compiler handles each message in the optimal way, the communication cost incurred by the tailored protocol is the *minimum* of the costs under the three basic modes.

$$T_{pc}(n) = \min(T_{native}(n), T_{buf}(n), T_{sync}(n)).$$

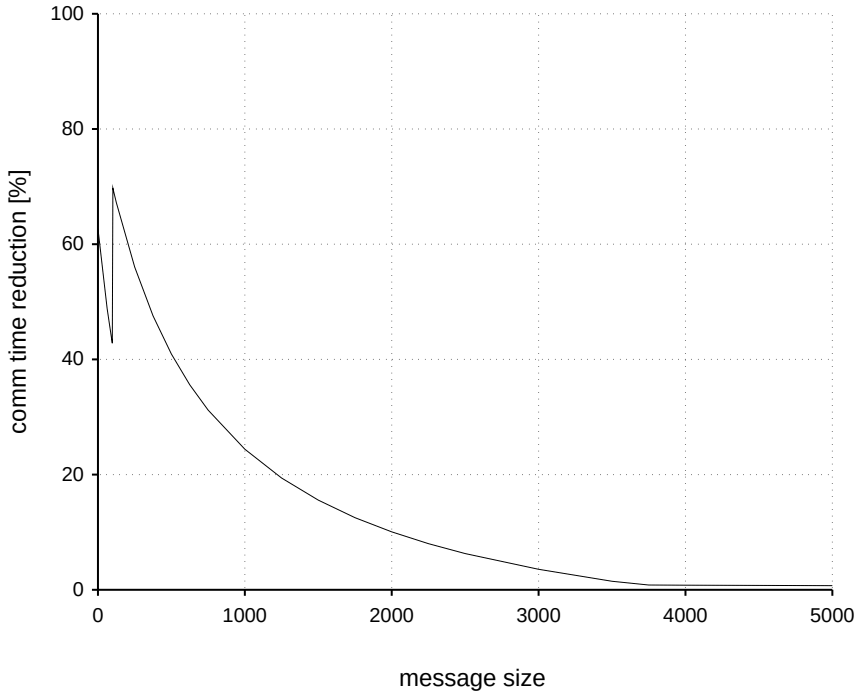

Figure 7.4: Predicted reduction in communication time due to using an ideal protocol compiler rather than the native communication system, as a function of message size.

The percentage reduction in protocol overhead is then

$$f_{overhead}(n) = \frac{T_{native}(n) - T_{pc}(n)}{T_{protocol}(n)}$$

We can now put all the pieces together to get an estimate of the improvement in communication performance due to using the protocol compiler. Assume we have an application program that sends M messages, with the ith message having length n_i. Using an ideal protocol compiler then reduces the total communication time by the fraction

$$f_{time} = \frac{\sum_{i=0}^{M-1} T_{native}(n_i) - \sum_{i=0}^{M-1} \min(T_{buf}(n_i), T_{sync}(n_i), T_{native}(n_i))}{\sum_{i=0}^{M-1} T_{native}(n_i)}.$$

For an application whose messages are all the same size (all $n_i = n$), we have

$$f_{time}(n) = \frac{T_{native}(n) - \min(T_{buf}(n), T_{sync}(n), T_{native}(n))}{T_{native}(n)}$$

Figure 7.4 shows the value of $f_{time}(n)$ as a function of n for the iPSC/860. The reduction in communication time is considerable for small messages, but the improvement due to the protocol compiler becomes negligible for messages larger than about 4000 bytes.

7.3.1 Applying the Model to the Macro-Benchmarks

The next step is to apply the performance model to predict the effect of the protocol compiler on the communication performance of the five macro-benchmarks. I did this by instrumenting the runtime library so that it adds up the sums in the formula for f_{time} as the application program runs.

Table 7.5 shows the results of this modeling. On the average, the protocol compiler is predicted to reduce communication time by about 24%.

7.4 Performance of Parachute

In addition to evaluating the general properties of protocol compilers, I performed experiments to measure the actual performance of Parachute and the protocols it

Table 7.5: Predicted effect of the protocol compiler on communication performance for the five macro-benchmarks. This table shows the predicted reduction in communication time under an ideal protocol compiler, compared to time under NX.

application	predicted comm time reduction
matmult	1.3%
nbody	7.6%
olfactory	42.5%
md	32.2%
eigen	34.2%
average	23.6%

generates. Inevitably, the performance of a first prototype will underestimate the potential of the protocol compiler approach, because of engineering compromises that were made in designing and constructing the prototype.

Several factors combine to make performance of Parachute protocols worse than could be achieved by an aggressive, "from scratch" implementation. The main degradation is due to the fact that Parachute is implemented on top of NX rather than manipulating the hardware directly. The semantic gap between the functionality of NX and the hardware means that Parachute protocols must sometimes use multiple kernel calls to do something that could potentially be done with a single call. (Section 6.1 discusses the reasons for building Parachute on top of NX.)

We can view the prototype results as a lower bound on the value of using protocol compilers. The only way to accurately determine the benefit of using a protocol compiler would be to measure the performance of a re-implementation of Parachute which takes advantage of the lessons learned in building the first version, and which is implemented more aggressively and on faster hardware.

Table 7.6 shows the measured performance improvement for the five macro-benchmarks. The improvement in communication time ranges from 2% to 45%, and the reduction in overall running time range from 0% to 20%. On the average, communication time is cut by about 17%, and overall running time by 7.5%.

The data show one surprising feature. For all five programs, the improvement in

Table 7.6: Performance of the macro-benchmarks, running on a 16-processor iPSC/860. The NX version of each program uses the native communication system, and the Parachute version uses a communication protocol generated by Parachute. All times are measured in seconds. The table shows the percentage of time that the NX version spent in communication, the improvement in communication time due to using Parachute rather than NX, and the reduction in total running time due to Parachute.

application	percent comm	comm time improvement	overall improvement
matmult	13.3%	1.7%	0.3%
nbody	13.0%	44.6%	8.6%
olfactory	78.3%	19.3%	19.8%
md	24.4%	17.2%	5.0%
eigen	89.8%	4.1%	3.8%
average	43.8%	17.4%	7.5%

total running time is larger than can be accounted for by the reduction in communication alone. Despite the fact that both versions (NX and Parachute) of the program execute exactly the same local calculation code, measurements show that the local calculation runs faster under Parachute. There are two ways in which communication code could be affecting the performance of local calculation. Some communication processing is done at interrupt-time; if the interrupt happens during local calculation, the interrupt-handler executes while the local calculation stopwatch is running. In addition, the communication code affects the contents of the cache, which can cause local calculation speed to change. Since the gain in local calculation time is observed for all five programs, we can assume that it would be common for other applications as well.

We can now compare the measured values of the reduction in communication time, to the values predicted by the performance model. Doing this makes the limitations of the model even more clear. The two agree only in the aggregate — the model predicts the average reduction as 24.4%, while measurement yields 17.3%. The small discrepancy can be explained by engineering compromises made in the design of

Parachute. Despite its shortcomings, the model is apparently adequate when averaged over several benchmarks.

The values for the individual benchmarks do not agree so well. Particularly large discrepancies are exhibited by `nbody`, for which the predicted improvement is too low, and `eigen`, for which it is too high. In the case of `nbody`, the model fails to account for the fact that the tailored protocol allows two messages, one outgoing and one incoming at each processor, to be overlapped, thus gaining nearly a factor of two in communication time. (This overlap is inherent in the source program, but NX fails to exploit it.) In the case of `eigen`, the model fails to account for the fact that about 70% of the nominal communication time is really load imbalance, which cannot be reduced by any protocol.

As stated above, a more accurate model, which takes into account these and other factors, would be useful. However, constructing such a model would be extremely difficult.

Table 7.7: Running time of Parachute (running on a DecStation 5000), generating tailored protocols for 16-processor runs of the macro-benchmarks, and statistics about the generated protocols. Statistics include the number of communication patterns in the program, the number of distinct send-calls encountered in these patterns, the number of states in the protocol's state machine, and the number of bytes of buffer space used by the tailored protocol. Except for the protocol compiler's running time, these are all static properties of the protocol.

application	prot comp time [sec]	patterns	send-calls	states	buffer space
matmult	0.2	3	187	748	51200
nbody	0.3	1	240	960	147456
olfactory	0.8	5	446	1784	2048
md	0.4	1	256	1024	3852
eigen	29.5	2	3219	12876	10000

Table 7.8: Running time of Parachute (running on a DecStation 5000), generating tailored protocols for 64-processor runs of the macro-benchmarks, and statistics about the generated protocols. Statistics include the number of communication patterns in the program, the number of distinct send-calls encountered in these patterns, the number of states in the protocol's state machine, and the number of bytes of buffer space used by the tailored protocol. Except for the protocol compiler's running time, these are all static properties of the protocol.

application	prot comp time [sec]	patterns	send-calls	states	buffer space
matmult	6.3	3	1275	5100	102400
nbody	33.4	1	4032	16128	589824
olfactory	21.2	5	1490	5960	9472
md	14.3	1	1536	6144	3852
eigen	465.9	2	9963	39852	10000

7.4.1 Protocol Compiler Statistics

Table 7.7 shows statistics about the execution of Parachute, and the protocols it generates, when generating 16-process protocols for the macro-benchmarks. Table 7.8 shows the same statistics for generating 64-process protocols. The running time of the protocol compiler is satisfactory, except for `eigen` in the 64-process case. Here we see the beginning of the blow-up in the protocol compiler's running time as the number of messages gets large. Clearly, scaling to a larger number of processes will require the development of techniques that break up large communication patterns into smaller pieces.

The other statistics show that the size of the generated protocols, in terms of the number of send-calls, number of states, and total buffer space requirements, is quite reasonable. The only possible exception is the buffer space requirement of `nbody`; for 64 processes the protocol requires almost 600K bytes. Of course, if this were unacceptable, adding a single line to the communication description file would cause a smaller protocol to be generated.

Table 7.9 shows how the tailored protocols use message-passing modes. Four of the benchmarks use only receiver-buffered mode. Matmult uses a mix of receiver-buffered

Table 7.9: Distribution of message modes in the five macro-benchmarks. The table shows the number of (static) send-calls that use each message mode.

Problem	receiver-buffered	synchronizing	blast
matmult	145	48	0
nbody	240	0	0
olfactory	446	0	0
md	256	0	0
eigen	3219	0	0

mode and synchronizing mode. In matmult, synchronizing mode accounts for 25% of the messages, and 43% of the bytes transmitted. There are no opportunities to use blast mode in the macro-benchmarks.

7.4.2 Effectiveness of the Protocol Compiler's Algorithms

Two phases of the protocol compiler's analysis, buffering-mode analysis and buffer allocation, face NP-complete problems. In both cases, Parachute uses a simple greedy algorithm. This section evaluates the effectiveness of these greedy algorithms.

Ideally, we could simply compare the solution generated by the greedy algorithm to the optimal solution to the problem. However, it is not feasible to find the optimal solution, so I had to be satisfied with a very good solution. I implemented an alternative solver for each of the two problems, using simulated annealing [Kirkpatrick et al. 83]. Simulated annealing is a simple, robust approach to combinatorial optimization problems; it has the advantage that longer running times yield better results. When searching for a near-optimal solution to a problem, I ran the simulated annealing solver for at least one hour on a DECstation 5000.

Simulated annealing works by starting with some feasible solution to the optimization problem, and then considering a series of "moves" from that solution. The moves are chosen randomly from some simple set of changes to the solution. If a move improves the objective function, it is accepted; if it damages the objective function by an amount Δ, then it is accepted with probability $e^{-\frac{\Delta}{T}}$, where T is a parameter called the "temperature". Accepting a move leads to a new current solution, while

rejecting a move leaves the current solution unchanged. The temperature parameter is slowly lowered from a large value, which causes the system to move randomly, to a temperature of zero, which makes the system accept only profitable moves.

For each of the two problems, I chose several problem instances at random from those generated by the benchmark programs discussed earlier in this chapter. I then compared the values of the objective function generated by Parachute's greedy algorithm, to the best solution I found in two one-hour runs of simulated annealing.

The following subsections describe the simulated annealing solvers, and what they revealed about the effectiveness of the protocol compiler's algorithms.

Buffering-Mode Analysis

The buffering-mode analysis problem is stated as follows: given a directed graph $G = (V, A)$ and a set of "optional" arcs A', with weights on the arcs in A', find the set $B \subseteq A'$ such that $(V, A \cup B)$ is acyclic, and the total weight of the arcs in B is maximized.

The simulated annealing process starts with $B = \emptyset$. A move is generated by randomly choosing an arc $a \in A'$. If $a \in B$, then the proposed move is to remove a from B. If $a \notin B$ and $(V, A \cup B \cup \{a\})$ is acyclic, then the proposed move is to add a to B. Otherwise, another move is chosen.

Table 7.10 shows the effectiveness of Parachute's buffering-mode analysis algorithm. The table compares the solution found by Parachute to the best solution found by simulated annealing, for buffering-mode analysis problems chosen randomly from those faced by Parachute when generating tailored protocols for the macrobenchmarks. As the table shows, Parachute apparently finds the optimal solution in all cases. Surprisingly, of about additional twenty buffering analysis problems, also chosen randomly from those faced by Parachute when processing the application programs, I was unable to find a single one where the greedy solution appeared to be nonoptimal. (A solution "appears to be optimal" when several long runs of simulated annealing all lock onto that solution in the first few seconds, and never find a better one.) It appears that the structure of these applications leads to a particularly easy set of buffering analysis problems.

Table 7.10: Effectiveness of Parachute's buffering-mode analysis algorithm. The table shows two solutions to a set of buffering-mode analysis problems: the first solution was generated by Parachute's greedy algorithm, and the second is the best solution found in two one-hour runs of simulated annealing. The last column shows the percentage difference in benefit function between the two solutions. The problems were chosen by picking randomly, for each application program, one of the (non-trivial) buffering-mode analysis problems solved by Parachute when generating protocols for the iPSC/860.

Problem	Parachute solution	best solution	difference
nbody	4147200	4147200	0.0%
matmult	307200	307200	0.0%
olfactory	30720	30720	0.0%
md	92672	92672	0.0%
eigen	240	240	0.0%

Buffer Allocation

The buffer allocation problem is stated as follows: given an undirected graph $G = (V, E)$ with labels $s_i \geq 0$ on each vertex $i \in V$, assign a quantity b_i to each vertex $i \in V$, such that:

- all $b_i \geq 0$, and
- for all $(i, j) \in E$, either:

  $$b_i + s_i \leq b_j, \text{ or}$$
  $$b_j + s_j \leq b_i,$$

  and
- $\max_{i \in V}(b_i + s_i)$ is minimized.

Here G is the conflict graph; each vertex represents a message, and an edge between two vertices means that those two messages cannot share buffer space. The label s_i

denotes the amount of buffer space message i uses, and b_i is the base address of the buffer assigned to message i.

The simulated annealing algorithm represents each solution as a totally ordered list of the vertices. From such an ordered list, a unique solution can be generated by considering the vertices in that order, and assigning each one the lowest buffer address b_i that does not conflict with any decision made so far.

Not all feasible assignments can be represented in this ordered-list form, but we are guaranteed that there is an optimal solution that can be represented in this form[1]. The advantage of the ordered-list representation is that it allows the simulated annealing solver to easily manipulate possible solutions to the problem. A simulated annealing move consists simply of interchanging the positions of two randomly-chosen vertices in the ordered list.

Table 7.11: Effectiveness of Parachute's buffer allocation algorithm. The table shows two solutions to a set of buffer allocation problems: the first solution was generated by Parachute's greedy algorithm, and the second is the best solution found in two one-hour runs of simulated annealing. The last column shows the percentage difference in cost function between the two solutions. The problems were chosen by picking randomly, for each application program, one of the (non-trivial) buffer allocation problems solved by Parachute when generating code for the iPSC/860.

Problem	Parachute solution	best solution	difference
nbody	55296	55296	0.0%
matmult	22400	22400	0.0%
olfactory	2816	2816	0.0%
md	5396	3852	40.1%
eigen	10000	10000	0.0%

[1] Proof sketch: Starting with any feasible solution, we can generate an ordered list of vertices by sorting the vertices in increasing order of b_i in that solution. One can show that the solution represented by this ordered list has a cost less than or equal to the cost of the original solution. Thus, if the original solution was optimal, then the resulting solution must also be optimal.

Table 7.11 shows an evaluation of the effectiveness of the greedy algorithm for buffer allocation. For each application program, I randomly chose one of the buffer allocation problems faced by Parachute when generating a 4-, 16-, or 64-process protocol for that application. I then compared the greedy solution for these problems with the best solution generated by two one-hour runs of simulated annealing.

Of the five randomly-chosen problems, the greedy algorithm appears to find the optimal solution for four, but does 40% worse than optimal for the other one. Apparently the greedy algorithm is capable of generating solutions that are far from optimal.

7.5 Summary

Performance models, and experiments with real applications, show that using a protocol compiler leads to a significant reduction in the running time of application programs.

Experiments on a set of typical applications show that the programs spend about half of their time in communication. Performance models predict that using an ideal protocol compiler reduces communication time by about 24%, and experiments show that Parachute cuts communication time by 17%. As a result, the reduction in total running time of the applications ranges from 0% to 20%, with an average of 7.5%. This improvement is seen in spite of the engineering compromises made in building Parachute; a more aggressive implementation on faster hardware would yield a larger speedup over traditional approaches.

Chapter 8

CONCLUSIONS

In the end, veracity and rectitude always triumph.
— Batman

We have seen that compile-time analysis of communication can effectively improve the performance of message-passing programs. By taking advantage of known communication patterns in an application, the protocol compiler can generate tailored protocols that reduce protocol overhead.

This chapter concludes by discussing first related work, and then future work.

8.1 Related Work

There are several classes of related work to consider.

8.1.1 Data Movement

Research on data movement has shown that low-level data transport can be provided efficiently in a variety of ways. Spector used custom microcode to implement fast remote-access instructions on 1980-vintage networked workstations [Spector 82]. Thekkath updated this idea by re-implementing it on today's fast workstations and networks [Thekkath et al. 93]. Delp and co-workers devised MemNet, which provides a set of workstations with a region of physical memory that is kept coherent by transmitting all writes around a ring interconnection. von Eicken, Culler, and their co-workers developed the Active Messages system, which provides fast data transport on multicomputers by decorating each packet with the address of a "handler" procedure to be executed on the receiving processor.

All these systems provide methods for moving data quickly between processors, but they do not in themselves implement the protocols needed to make this data transport useful. Protocols must be built on top of these data movement systems.

8.1.2 Compiling to Message-Passing Code

Many researchers have studied how to compile languages into message-passing code. These languages include Dino [Rosing 91], Dataparallel C [Hatcher & Quinn 91], C* [Rose & Steele 87, TMC 90], CM-Fortran [TMC 89], Fortran D [Fox et al. 91, Tseng 93], Vienna Fortran [Chapman et al. 92], NESL [Blelloch et al. 93], and many others.

While several such compilers generate efficient message-passing code, they all treat `send` and `receive` as indivisible primitives. The code generated by these systems could still be sped up by using tailored protocols rather than generic message-passing libraries.

8.1.3 Optimizing Transformations for Message-Passing Code

Another related area is compiler optimization and transformation of message-passing programs [Rogers & Pingali 89, Hatcher & Quinn 91, Amarasinghe & Lam 93]. Optimizations take several forms: aggregating several messages into one, lifting invariant messages out of loops, moving sends earlier and receives later to hide message latency, or recognizing collective communication operations like broadcast or parallel-prefix.

Like the language compilation work, the optimization research treats `send` and `receive` or higher-level operations as indivisible building blocks. These optimizations are valuable, but the resulting code could be improved further by using tailored protocols.

8.1.4 Efficient Protocol Implementation Techniques

The *x*-kernel [Hutchinson & Peterson 91] and Morpheus [Abbott & Peterson 92] are two systems that facilitate the construction of efficient protocols. The x-kernel provides a software substrate through which small pieces of protocol code can be assembled into a single protocol. Morpheus goes further by defining a programming language in which to write protocol code, and then compiling protocols written in that language into efficient implementations.

These systems are very useful for building fast and flexible implementations of protocols that have been designed by a person. However, unlike my work, these systems do not *design* communication protocols automatically.

8.1.5 Protocol Verification

A great deal of research has been done on formal verification of protocols. This work typically starts with a formal description of the behavior of some implementation, and then proves that it meets a specification. Like protocol implementation research, this work treats the protocol as a given. Although the machine analyzes the protocol, unlike in my work the machine does not design the protocol.

8.2 Future Work

This work could be extended in several interesting ways. First, Parachute could be re-implemented on other architectures, such as the CM-5 or a set of workstations connected by a high-speed network. Second, it could be extended to apply to a wider class of programs, for example, by allowing optimized communication patterns to overlap with each other asynchronously, or by allowing incompletely specified patterns. Third, runtime compilation could be incorporated, using the same analysis techniques described in earlier chapters. Fourth, this work could be extended to allow collective communication operations, such as multicast or parallel-prefix, to be treated as first-class operations on a par with `send` and `receive`. Finally, the protocol compiler's analysis could be extended to do more aggressive optimization.

The following subsections discuss these extensions, touching briefly on the issues involved in each.

8.2.1 Porting to Other Architectures

The most likely target architectures are the CM-5, and workstations connected by an ATM network[1]. The CM-5 and ATM networks share an important characteristic not present on the Intel machines: they use small, fixed-size packets (20 bytes on the CM-5, 48 bytes on the ATM). Each network adds another twist — the CM-5 does not guarantee in-order delivery of message packets, and the ATM network occasionally drops packets.

Re-ordering of packets should not be a major issue. Each packet can simply be labelled with a sequence number, identifying its position within the user-level

[1] Note that I am discussing the "dedicated system in the closet" model of workstation-network computing, and not the more difficult model based on capturing idle cycles in desktop machines.

message. The receiver can simply count the packets when they come in, enabling it to determine when the entire message has been delivered. Since messages arrive into predetermined buffers, message reassembly can happen in-place on arrival.

Packet loss, which occurs on the ATM network, is a more serious problem. ATM networks drop packets for two reasons: data corruption and congestion. Data corruption errors are so rare that they can be ignored — mistaking data corruption for congestion will not affect correctness and will have only a tiny effect on performance.

Brustoloni and Bershad present a promising message-passing protocol for ATM networks [Brustoloni & Bershad 92]. Their protocol uses per-message pre-reservation to allocate buffer space for arriving messages. (They use a separate protocol for small messages.) Dropped packets cause the sender to retry with exponential backoff.

The protocol compiler technique could be used to improve the performance of this protocol. In this approach, buffer allocation would be done at compile-time, but the congestion control mechanisms of Brustoloni and Bershad would not be changed — timeout, retry, and exponential backoff would still be used.

From the protocol compiler's point of view, the only new feature of this situation is the fact that the sending processor must keep a copy of transmitted message data available until an acknowledgement has arrived. This problem can be handled by simply blocking the sender and accepting a (perhaps small) performance degradation. Alternatively, the protocol compiler could sometimes choose to use sender buffering — copying message data into a buffer on the sending end, and discarding this buffer only when the message has been fully acknowledged.

Small-packet networks like the CM-5 and ATM offer an additional problem. Because packets are so small, and interrupting a processor is relatively expensive, we cannot afford to interrupt a processor whenever a packet arrives. The protocol must take care, and the hardware must allow, the use of one interrupt for each group of packets rather than for individual packets only. Thekkath and Levy offer an analysis of these architectural issues [Thekkath & Levy 93].

Remote-Access Architectures

Remote-access systems are another important class of network architectures. These systems associate each memory location with one processor, but allow processors to directly read or write the memory of other processors. Systems in this class include

the BBN Butterfly and Cray T3D, in addition to workstations connected by devices like MemNet.

For building tailored protocols, these systems provide some advantages over traditional message-passing networks. In particular, they allow one processor to put data into the communication buffers of another processor without interrupting the destination processor. Porting Parachute to one of these systems is simply a matter of rewriting the run-time library to address the new architecture.

8.2.2 Extending to Wider Classes of Programs

One important limitation of Parachute is that it applies only to fixed communication patterns. The system would be more useful if it could support a larger class of programs.

There are two main difficulties in extending Parachute. First, we must define a formal language that can specify the behavior of programs in the expanded class; this language plays the role that PDL plays in Parachute. Second, we must determine how the protocol compiler can exploit the information in this language to generate efficient protocols.

An Example

Consider the problem of extending Parachute to allow data-parallel conditional statements inside a communication pattern. This allows some piece of code, including communication, to either be executed on all processors, or not be executed at all, during a communication pattern.

First, we must adjust the communication description language to account for conditional statements by adding the ability to conditionally execute a pattern inside of another.

Second, the protocol compiler must be able to analyze the extended class of programs. The main issue raised by conditionals is that they preclude the availability of exact "happens-before" relationships between events. In other words, whether or not a conditional is executed could affect the existence of causal relationships between statements in the surrounding code. Thus, the protocol compiler's analysis would have to be carefully extended, using conservative estimates of event orderings where necessary. It is beyond the scope of this example to describe exactly how this might

be done, but it is clearly possible, and this extension is, in fact, likely to appear in a future version of Parachute.

8.2.3 Runtime Compilation

Runtime compilation applies the protocol compiler to communication patterns that are executed several times, but are not known until runtime. For example, in a finite-element simulation, the specific communication patterns are not known until the exact set of elements being simulated is known. This information is typically read in at the beginning of a run, and not changed thereafter.

Runtime compilation would use the inspector/executor model [Saltz et al. 91]. In this model, the first execution of a pattern occurs slowly, as general-purpose mechanisms are used; during this execution, each process records its sequence of communication operations. Once the first execution is finished, the protocol compiler is run to analyze the pattern that was recorded. This allows subsequent executions of the pattern to use the optimized protocol generated by this analysis.

Runtime compilation would require only a small extension to the message-passing interface. Calls would be added to declare a new pattern, and to invalidate an existing pattern. In addition, the protocol compiler's analysis phase would be integrated into the runtime library, so it could be executed as necessary.

Executing the protocol compiler at runtime introduces one additional issue: the time required for the analysis. This time affects the usefulness of runtime compilation by determining how many times a pattern must be executed before runtime compilation is profitable. There are two ways to improve this crossover point. First, we could speed up the execution of the analyzer, perhaps by parallelizing it. Second, we could develop quick-and-dirty algorithms that provide a satisfactory, but not ideal, result quickly.

8.2.4 Collective Communications as First-Class Operations

One limitation of the current implementation is that it supports only `send` and `receive` as first-class operations. The main consequence of this is that only `send` and `receive` can be used as split-phase operations, with separate `begin` and `end` operations that allow them to be overlapped with local calculation or other communication.

There are no split-phase versions of important collective communication operations like `broadcast` or `parallel-prefix`. These collective operations must currently be broken down into their constituent `send` and `receive` operations.

We would like to allow split-phase versions of collective communication operations. There are two ways to do this: either supporting a fixed set of collective communication operations, or allowing the programmer or compiler freedom to specify new collective communication operations. The second, user-extensible, approach is not much harder than the first, so it is preferred.

Logically, a split-phase collective operation is like a separate thread of control within each process; this thread sends and receives messages asynchronously. For instance, in a broadcast operation this thread may receive one message and then send several messages. Logically, this thread is spawned by the main thread at the beginning of the split-phase operation; at the end of the split-phase operation the main thread waits for the subordinate thread to complete.

Note that this model allows split-phase operations to be nested within other split-phase operations. In its full generality, any properly nested structure is possible.

Although the execution of these programs has been described using separate threads of control, the use of threads only expresses the semantics — it does not necessarily imply an implementation based on general-purpose threads. In particular, the implementation may choose to statically interleave the operations of the threads, or it may embed the desired behavior of a subordinate thread into an interrupt-handler associated with the arrival of a message. Implementation details remain to be worked out.

8.2.5 Improved Optimization

Although the current version of Parachute generates efficient code, there are steps remaining to be taken that afford even greater possibilities for improving the quality of the generated code.

Reducing Synchronization

Parachute uses barrier synchronizations at run-time, on entry to each communication pattern, to ensure that storage is not re-used until the previous use is finished. Since each communication pattern is analyzed separately, storage used by one pattern may

be re-used by the next pattern executed. Since the use of storage is triggered by the arrival of messages sent by other processes, the protocol cannot send a buffered message from process A to process B until B is in the same communication pattern as A. This property is ensured by making all processes perform a barrier synchronization at the beginning of each pattern.

Several techniques can be used to reduce the number of barriers executed. First, if two communication patterns use totally disjoint regions of buffer space, there is no need to separate them by a barrier synchronization. If buffer space is plentiful, the protocol compiler can carefully lay out the buffers allocated to the patterns to avoid conflicts. In one simple variant of this scheme, the protocol compiler can allocate k distinct replicas of the program's buffers. At run-time, the first pattern executed would use the first replica, the second pattern would use the second replica, and so on. A barrier would be required only after every kth pattern. In exchange for increasing buffer space by a factor of k, this strategy cuts the number of barriers by a factor of k.

Another way to reduce the number of barriers is to take advantage of the "free" synchronization provided by messages within a pattern. Consider the case where three patterns A, B, and C are executed in sequence, A and B have disjoint buffers, B and C have disjoint buffers, but A and C do *not* have disjoint buffers. A barrier synchronization is apparently required at some time between the end of A and the beginning of C. However, the messages in pattern B impose some synchronization on the program. Indeed, B may impose sufficient synchronization that the barrier is not necessary at all. Even if the synchronization in B cannot completely replace the barrier, it can at least reduce the amount of "artificial" synchronization that must be added to insulate A from C.

A third way to reduce synchronization is to use trace scheduling [Fisher 81]. Originally developed to aid scheduling of instructions across basic-block boundaries in compilers, this technique works by "guessing" a sequence of patterns that will be executed, and then treating the entire sequence as if it were a single pattern. If the prediction is correct, the sequence can be executed without any barrier synchronizations, because buffers were allocated for the sequence as a whole. If, on the other hand, the execution deviates from the predicted sequence of patterns, a barrier synchronization is done, and execution continues normally.

Many of these techniques for reducing synchronization rely on information about the frequency with which certain patterns, or sequences of patterns, are executed. This information can be included as hints in the directives section of the communication description file. The programmer can generate these hints directly, or they can be derived by tracing. In some cases, runtime compilation can be used to detect the necessary information at run-time and react to it.

Using Timing Information

In addition, we could take advantage of information about the timing of events to make better choices of whether or not to buffer messages. This requires a timing model to be developed, to estimate how long various communication operations take. In addition, it requires accurate information about the running time of the local-calculation sections of the application program. This information is probably best gleaned by tracing the program. If the communication description language is extended to carry timing hints, we can hope to automatically generate these hints by using tracing.

8.3 Conclusion

In this dissertation, I have opened up the "black box" of message-passing to reveal the benefit of using carefully tailored mechanisms rather than the generic mechanisms that are used today. Together with the previous work on the protection and data-movement problems, my work offers a strategy for substantially reducing the cost of communication. I hope that this is one useful step on the road to flexible, practical parallel computing.

Bibliography

[Abbott & Peterson 92] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. In *Proceedings of ACM SIGCOMM '92 Conference*, pages 27–38, 1992.

[Agarwal et al. 91] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[Amarasinghe & Lam 93] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[Anderson & Snyder 91] R. Anderson and L. Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 79(4):480–487, 1991.

[Anderson et al. 89] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, 1989.

[Anderson et al. 91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[Anderson et al. 92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level manage-

ment of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Bershad et al. 93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *38th IEEE Computer Society Intl. Conf. (COMPCON)*, pages 524–533, February 1993.

[Blelloch et al. 93] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of Fourth SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, May 1993.

[Bokhari 90] S. Bokhari. Communication overhead on the intel iPSC-860 hypercube. Technical Report Interim Report 10, ICASE, May 1990.

[Bower et al. 88] J. M. Bower, M. E. Nelson, M. A. Wilson, G. C. Fox, and W. Furmanski. Piriform (olfactory) cortex model on the hypercube. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 977–999, 1988.

[Brustoloni & Bershad 92] J. C. Brustoloni and B. N. Bershad. Simple protocol processing for high-bandwidth low-latency networking. Technical Report CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, March 1992.

[Byrd & Delagi 88] G. Byrd and B. Delagi. A performance comparison of shared variables versus message passing. In *Proceedings of Third International Conference on Supercomputing*, volume 1, pages 1–7, 1988.

[Callahan & Kennedy 88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[Carriero & Gelernter 89] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *Computing Surveys*, 21(3):323–357, September 1989.

[Carter et al. 91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Chapman et al. 92] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran — a Fortran language extension for distributed memory systems. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*. Elsevier Press, 1992.

[Culler et al. 93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumette, T. von Eicken, and K. Yelick. Introduction to Split-C. Available by anonymous ftp from *mammoth.berkeley.edu*, 1993.

[Dally & Seitz 87] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[Dally 87] W. J. Dally. Wire-efficient VLSI multiprocessor communication networks. In P. Losleben, editor, *Proceedings of Stanford Conference on Advanced Research in VLSI*, pages 391–415. MIT Press, March 1987.

[Dally 90] W. J. Dally. The J-machine system. In P. Winston and S. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, volume 1. MIT Press, 1990.

[Dally 92] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, pages 194–205, March 1992.

[Derby et al. 90] T. M. Derby, E. Eskow, R. K. Neves, M. Rosing, R. B. Schnabel, and R. P. Weaver. The DINO user's manual. Technical Report CU-CS-501-90, University of Colorado, November 1990.

[Feeley & Levy 92] M. J. Feeley and H. M. Levy. Distributed shared memory with versioned objects. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, pages 247–262, October 1992.

[Felten & McNamee 92] E. W. Felten and D. McNamee. Newthreads 2.0 user's guide, 1992.

[Fisher 81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[Foster & Taylor 90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[Foster et al. 92] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1:51–66, 1992.

[Fox 88] G. C. Fox. What have we learnt from using real parallel machines to solve real problems. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 897–955, 1988.

[Fox et al. 87] G. C. Fox, A. J. G. Hey, and S. W. Otto. Matrix algorithms on the hypercube I: Matrix multiplication. *Parallel Computing*, 4, 1987.

[Fox et al. 88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[Fox et al. 91] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141 (revised), Dept. of Computer Science, Rice University, April 1991.

[Geist et al. 91] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL, a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, 1991.

[Gelernter & Carriero 92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Gharachorloo et al. 90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[Griswold et al. 90] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proceedings of Fifth Distributed Memory Computing Conference*, 1990.

[Hatcher & Quinn 91] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[Hatcher et al. 91] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *Proceedings of Third SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 1991.

[Heath & Etheridge 91] M. T. Heath and J. A. Etheridge. Visualizing performance of parallel programs. Technical Report TM-11813, Oak Ridge National Laboratory, May 1991.

[Hillis & Steele 86] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[Hillis 85] W. D. Hillis. *The Connection Machine*. MIT Press, 1985.

[Hoare 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HPFF 93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[Hutchinson & Peterson 91] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Intel 91a] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[Intel 91b] Intel Supercomputer Systems Division. *A Touchstone DELTA System Description*, February 1991.

[ISO 88] ISO. *Information Processing — Open Systems Interconnection — LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1988. ISO International Standard 8807.

[Karp 72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[Kirkpatrick et al. 83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671, 1983.

[Koelbel et al. 90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of Second SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, March 1990.

[Lamport 78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Larus et al. 92] J. R. Larus, B. Richards, and G. Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report 1126, University of Wisconsin, Computer Sciences Dept., November 1992.

[LeBlanc 86] T. LeBlanc. Shared-memory versus message-passing in a tightly-coupled multiprocessor: A case study. In *Proceedings of International Conference on Parallel Processing*, pages 463–466, 1986.

[Leiserson 85] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–900, October 1985.

[Leiserson et al. 92] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the connection machine CM-5. In *Proceedings of 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[Lenoski et al. 92] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[Li & Hudak 86] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *ACM Conf. on Principles of Distributed Systems*, pages 229–239, 1986.

[Lin & Snyder 90] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of International Conference on Parallel Processing*, pages II 163–180, 1990.

[Littlefield & Maschhoff 93] R. J. Littlefield and K. J. Maschhoff. Investigating the performance of parallel eigensolvers for large processor counts. *Theoretica Chimica Acta*, 84:457–473, 1993.

[Littlefield 92] R. J. Littlefield. Characterizing and tuning communications performance for real applications. In *Proceedings of the First Intel DELTA Applications Workshop*, pages 179–190, February 1992. Proceedings also available as Caltech Technical Report CCSF-14-92.

[Logrippo et al. 92] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23(5):325–342, February 1992.

[Markatos & LeBlanc 92] E. P. Markatos and T. J. LeBlanc. Shared-memory multiprocessor trends and the implications for parallel program performance. Technical Report 420, Computer Science Dept., University of Rochester, May 1992.

[Mellor-Crummey & Scott 91] J. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[Milner 80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[NCUBE 87] NCUBE Corporation. *NCUBE Users Handbook*, 1987.

[Ngo & Snyder 92] T. A. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. In *Proceedings of Scalable High Performance Computing Conference*, pages 284–291, April 1992.

[Nikhil 88] R. S. Nikhil. Id (version 88.0) reference manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, March 1988.

[Otto & Wolfe 92] S. W. Otto and M. Wolfe. The MetaMP approach to parallel programming. In *Frontiers '92: Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 1992.

[Otto 93] S. Otto, 1993. Personal communication.

[Parasoft 88] Parasoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[Pierce 88] P. Pierce. The NX/2 operating system. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[Plimpton 90] S. J. Plimpton. Molecular dynamics simulations of short-range force systems on 1024-node hypercubes. In *Proceedings of Fifth Distributed Memory Computing Conference*, pages 478–483, April 1990.

[Quinn et al. 90] M. J. Quinn, P. J. Hatcher, and B. K. Seevers. Implementing a data parallel language on a tightly coupled multiprocessor. In *Proceedings of 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, 1990.

[Rogers & Pingali 89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[Rose & Steele 87] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[Rosing 91] M. Rosing. *Efficient Language Constructs for Complex Data Parallelism on Distributed Memory Multiprocessors.* PhD dissertation, University of Colorado, 1991.

[Rosing et al. 90] M. Rosing, R. W. Schnabel, and R. P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado, April 1990.

[Saltz et al. 91] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[Skjellum & Leung 90] A. Skjellum and A. Leung. Zipcode: A portable multicomputer communication library atop the reactive kernel. In *Proceedings of Fifth Distributed Memory Computing Conference*, pages 767–776. IEEE Press, 1990.

[Snyder 86] L. Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.

[Spector 82] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):246–260, April 1982.

[Sullivan & Brashkow 77] H. Sullivan and T. R. Brashkow. A large scale homogeneous machine. In *Proceedings of 4th Annual Symposium on Computer Architecture*, pages 105–124, 1977.

[Sunderam 92] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 4:509–531, 1992.

[Thekkath & Levy 93] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[Thekkath et al. 93] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Efficient support for multicomputing on ATM networks. Technical Report 93-04-03, University of Washington, 1993.

[TMC 89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual, Version 5.2*, 1989.

[TMC 90] Thinking Machines Corporation. *C* Programming Guide (version 6.0)*, August 1990.

[TMC 91] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[TMC 92] Thinking Machines Corp. *CMMD 2.0 Reference Manual*, 1992.

[Tseng 93] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD dissertation, Rice University, January 1993. Also published as technical report COMP TR93-199.

[Tullsen & Eggers 93] D. M. Tullsen and S. J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 278–288, May 1993.

[van Eijk et al. 89] P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. North-Holland, Amsterdam, 1989.

[von Eicken et al. 92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

Appendix A

SYNTAX OF THE COMMUNICATION DESCRIPTION LANGUAGE

This appendix contains a full grammar for the PDL language, which is used in communication description files. A discussion of communication description files, and a less formal explanation of PDL, can be found in section 6.2.1.

program : *directivePart patternPart*

directivePart : *numProcsDir directivePart*
| *spaceDir directivePart*
| ϵ

numProcsDir : **numprocesses** `integer`

spaceDir : **spaceadvice** `integer`
| **spacelimit** `integer`
| **spacelimit** `integer integer`

patternPart : *pattern patternPart*
| ϵ

pattern : **pattern** `integer` { *procPatList* }

procPatList : *procPat procPatList*
| *procPat*

procPat : **process** `integer` { *statementList* }

statementList : *statement statementList*
| *statement*

statement : *beginSendStat*
| *endSendStat*
| *beginRecvStat*
| *endRecvStat*
| *sendStat*

| *recvStat*

beginSendStat : **beginSend dest** `integer` **tag** `integer` **maxsize** `integer` **name** `identifier`

beginRecvStat : **beginRecv source** `integer` **tag** `integer` **maxsize** `integer` **name** `identifier`

endSendStat : **endSend name** `identifier`

endRecvStat : **endRecv name** `identifier`

sendStat : **send dest** `integer` **tag** `integer` **maxsize** `integer`

recvStat : **recv source** `integer` **tag** `integer` **maxsize** `integer`

Appendix B

NP-COMPLETENESS PROOF FOR THE BUFFERING-MODE PROBLEM

This appendix proves that the buffering analysis problem introduced in section 5.2.3 is NP-complete. The problem may be formally stated as follows: given a deterministic communication pattern C, with a weight w_i on each message in C, assign a buffering mode (either receiver-buffered or synchronizing) to each message in C, such that deadlock is avoided, and the total weight of all synchronizing-mode messages is maximized.

This problem is trivially in NP, since verifying a solution requires simply adding up the total benefit function and checking for cycles in the event-ordering graph generated by the proposed solution.

To simplify the proof, I will work with an unweighted version of the problem. The unweighted version is the same as the original problem, except that all messages have unit weight. Thus the objective is simply to maximize the number of messages that use synchronizing mode. Since the unweighted problem is a special case of the weighted problem, if the unweighted problem is NP-hard, then the weighted problem must also be NP-hard.

In order to further simplify the proof, I will restate the unweighted problem in an equivalent form. Rather than trying to maximize the number of synchronizing-mode messages, the algorithm will try to minimize the number of messages that use receiver-buffered mode. Since all messages must use one of these two modes, this restatement does not affect the solution to the problem.

The resulting problem will be called the Minimum Buffer Problem (MBP). We can state this problem in decision form as follows:

> Given a deterministic communication pattern C that transfers M messages, and a nonnegative integer $K \leq M$, is it possible to assign buffering modes (synchronizing or receiver-buffered) to all messages in C, such that the resulting program is deadlock-free and at most K messages are

receiver-buffered?

MBP is complete for NP because the feedback arc set problem, which is known to be NP-complete [Karp 72], can be reduced to MBP. The feedback arc set problem is stated as follows: given a directed graph $G = (V, A)$ and a positive integer $K \leq |A|$, is there a subset $A' \subseteq A$ with $|A'| \leq K$ such that A' contains at least one edge from every directed cycle in G? This problem can be restated equivalently as follows: given a directed graph $G = (V, A)$ and a positive integer $K \leq |A|$, is there a subset $A' \subseteq A$ with $|A'| \leq K$ such that the graph $(V, A - A')$ is acyclic?

Suppose we are given a directed graph $G = (V, A)$. Assume that its vertices are denoted $v_1, v_2, \ldots, v_{|V|}$ and its arcs are denoted $a_1, a_2, \ldots, a_{|A|}$. From this graph we will construct a communication pattern as follows:

- The pattern will be executed by $|V|$ processes, denoted $P_1, P_2, \ldots, P_{|V|}$.
- The pattern will have $|A|$ messages, denoted $m_1, m_2, \ldots, m_{|A|}$.
- Processor P_i's program is as follows:
  - First, for each arc $A_k = (j, i) \in A$, a statement `beginSend dest` j `tag` k `maxsize 1 name` ψ_k. Call this statement bs_k or $bs_{(j,i)}$.
  - Second, for each arc $A_k = (j, i) \in A$, a statement `endSend` ψ_k, which will be called es_k or $es_{(j,i)}$.
  - Third, for each arc $A_k = (i, j) \in A$, a statement `beginRecv source` i `tag` k `maxsize 1 name` ρ_k, which will be called br_k or $br_{(i,j)}$.
  - Fourth, for each arc $A_k = (i, j) \in A$, a statement `endRecv` ρ_k, called er_k or $er_{(i,j)}$.

  The order of statements within each of the four groups does not matter.

One additional definition will simplify the proof. I define the *augmented* event-ordering graph of a communication pattern to be the result of taking ordinary event-ordering graph, and adding an edge between every pair of nodes (a, b) such there is a path from a to b containing only program-order edges. Clearly, the augmented event-ordering graph has a cycle if and only if the ordinary event-ordering graph has a cycle.

Lemma B.1 *Let $F = (G, K)$ be an instance of the feedback arc set problem, and $B = (C, K)$ be the instance of MBP generated by the construction given above. Let $H = (V, E)$ be the augmented event-ordering graph induced by any set of decisions about the buffering modes of messages in B. Then any cycle of H may contain only edges of the following forms:*

a) $(es_{(i,j)}, es_{(i,k)})$

b) $(es_{(i,j)}, br_{(k,i)})$

c) $(br_{(i,j)}, br_{(k,j)})$

d) $(br_{(i,j)}, es_{(i,j)})$.

Proof: The proof proceeds by proving that all alternative forms are impossible.

That the cycle cannot contain any *bs* nodes. This is because the only incoming edges to *bs* nodes are program-order edges from other *bs* nodes. Since program-order edges alone cannot form a cycle, a cycle containing a *bs* node must also contain at least one non-*bs* node. But there are no edges from non-*bs* nodes to *bs* nodes, so such a cycle cannot exist.

An argument similar to the previous paragraph shows that the cycle cannot contain any *er* nodes. (The only edges leaving *er* nodes are program-order edges.)

The previous two paragraphs imply that the cycle may not contain any message-order edges, since all message-order edges have the form (bs_x, er_y).

Now there are only two remaining kinds of edges allowed in the cycle. The first kind consists of program-order edges involving *es* and *br* nodes; together, forms (a), (b), and (c) describe all such edges. The second kind consists of rendezvous edges; form (d) describes all such edges. □

Now the following theorem establishes the equivalence of the two problems: G has a feedback arc set of size $\leq K$ if and only if the generated message-passing program can be executed while buffering $\leq K$ messages. This shows that the feedback arc set problem can be reduced to MBP. Thus MBP is NP-complete.

Theorem B.2 *Let $F = (G, K)$ be an instance of the feedback arc set problem, and $B = (C, K)$ be the instance of MBP generated by the construction given above. Then F and B have the same solution — they are either both true or both false.*

Proof: By contradiction. If the theorem is false, one of two cases must hold:

- *Case 1: F is true and B is false.*

  Let $G = (V, A)$ be the directed graph in F. Since F is true, there is some set $A' \subseteq A$ of arcs such that $|A'| \leq K$ and $(V, A - A')$ is acyclic. Let M' be the subset of messages of B defined by $M' = \{m_k | a_k \in A'\}$. Since B is false, the program described by B, with all messages from M' in receiver-buffered mode, must deadlock. Thus the resulting augmented event-ordering graph has a cycle.

  By lemma B.1, the cycle can be written in the form

  $$(br_{(i_1,i_2)}, es_{(i_1,i_2)}, br_{(i_2,i_3)}, es_{(i_2,i_3)}, \cdots, es_{(i_{k-1},i_k)}, br_{(i_k,i_1)}, es_{(i_k,i_1)}, br_{(i_1,i_2)}.$$

  Consider each edge of the form $(br_{(j,m)}, es_{(j,m)})$ in this cycle. Since both nodes stem from the message (j, m) they must be executed by different processes, and hence cannot be connected by a path of program-order edges. The only remaining kind of edge that connects a br node to a es node is a rendezvous edge, so the edge must correspond to a message which is in synchronizing mode. Hence the edge corresponds to an edge (j, m) in the graph $(V, A-A')$. Therefore $(V, A-A')$ contains the cycle $(i_1, i_2, i_3, \cdots, i_k, i_1)$, which is a contradiction. Thus case 1 cannot hold.

- *Case 2: F is false and B is true.*

  Since B is true, there is some set $M' \subseteq M$ of messages with $|M'| \leq K$ such that buffering the messages in M' eliminates all cycles in B's event-ordering graph. Let A' be the set of edges in F such that $a_k \in A'$ iff $m_k \in M'$. Thus A' is a set of at most K edges.

  Since F is false, the graph $(V, A - A')$ has at least one cycle. Let this cycle be

  $$(a_{i_1}, a_{i_2}, \cdots, a_{i_k}, a_{i_1}).$$

  Now consider the set of messages $M'' = \{m_{i_1}, m_{i_2}, \cdots, m_{i_k}\}$. All messages in M'' are synchronizing, since the corresponding edges in F are not in A'.

Thus, B's augmented event-ordering graph has the cycle

$$(br_{i_1}, es_{i_1}, br_{i_2}, es_{i_2}, \cdots, br_{i_k}, es_{i_k}, br_{i_1}).$$

So, receiver-buffering the messages in M does not eliminate all cycles in B's event-ordering graph, which is a contradiction. Hence case 2 cannot hold.

□

Appendix C

PSEUDOCODE FOR COMMUNICATION OPERATIONS IN THE RUNTIME LIBRARY

This appendix contains pseudocode for some of the communication operations supported by Parachute's run-time library. The goal of including this pseudocode is not to map out every detail, but simply to give the reader a general idea of how the implementation works.

In order to avoid confusion between the operations that the native operating system provides to the run-time library, and the operations that the run-time library provides to the application, a naming convention will be used. The names of calls to the native operating system will all be prefixed by "`Nat_`".

Figures C.1, C.2, C.3, C.4, C.5, C.6, and C.7 contain pseudocode for the main operations supported by the runtime library.

When the program is not in a communication pattern, each communication call maps directly onto the corresponding native call. This allows the library to be invisible outside of communication patterns.

Code for handling each message is split between the various procedures. The library uses a per-message data structure, which is generated as part of the tailored protocol, to keep information about each message.

For messages in receiver-buffered mode, the system keeps a `Nat_beginRecv` operation outstanding at all times. This causes any incoming instance of that message to be directed to the buffer that the protocol compiler allocated for it. When the library is initialized, it issues a `Nat_beginRecv` for each receiver-buffered message, then performs a barrier synchronization. This ensures that there is an outstanding `Nat_beginRecv` for each receiver-buffered message before any communication pattern can start. Whenever a `Nat_endRecv` is executed for a receiver-buffered message, the corresponding `Nat_beginRecv` is reissued immediately. The system makes sure that at most one instance of each message is "alive" at any time by placing a barrier synchronization at the beginning of each communication pattern.

For a message in synchronizing mode, the system carries out a small synchronization between sender and receiver for each instance of the message. When the receiver makes its `beginRecv` call, it makes a `Nat_beginRecv` call, then signifies its readiness to receive the message by sending a zero-length message to the sender. The sender waits for this zero-length message before transmitting the main message data.

```
int beginSend(void* buf, int nbytes, int dest, int tag)
{
  if(currentCommunicationPattern != NULL){
    message = stateMachine.getIdOfThisMessage();
    check parameters;
    advance state machine to next state;
    if(message->bufferingMode == Blast){
      return Nat_beginSend(buf, nbytes, dest, message->msgID);
    }else if(message->bufferingMode == ReceiverBuffered){
      return Nat_beginSend(buf, nbytes, dest, message->msgID);
    }else{
      save message parameters to per-message data structure
      return a magic cookie;
    }
  }else{
    return Nat_beginSend(buf, nbytes, dest, tag);
  }
}
```

Figure C.1: Pseudocode for the `beginSend` operation.

```
int beginRecv(void* buf, int maxSize, int tag)
{
  if(currentCommunicationPattern != NULL){
    message = stateMachine.getIdOfThisMessage();
    check parameters;
    advance state machine to next state;
    if(message->bufferingMode == Blast){
      return Nat_beginRecv(buf, maxSize, message->msgID);
    }else if(message->bufferingMode == ReceiverBuffered){
      store arguments to this call in message descriptor;
      return message->recvID;
    }else{
      ret = Nat_beginRecv(buf, maxSize, message->msgID);
      Nat_send(NULL, 0, message->sender, message->msgID);
      return ret;
    }
  }else{
    return Nat_beginRecv(buf, maxSize, tag);
  }
}
```

Figure C.2: Pseudocode for the `beginRecv` operation.

```
int endSend(int cookie)
{
  if(currentCommunicationPattern != NULL){
    message = stateMachine.getIdOfThisMessage();
    check parameters;
    advance state machine to next state;
    if(message->bufferingMode == Synchronizing){
      Nat_recv(NULL, 0, message->msgID);
      return Nat_send(use the saved parameters);
    }else{
      return Nat_endSend(cookie);
    }
  }else{
    return Nat_endSend(cookie);
  }
}
```

Figure C.3: Pseudocode for the `endSend` operation.

```
int endRecv(int cookie)
{
  if(currentCommunicationPattern != NULL){
    message = stateMachine.getIdOfThisMessage();
    check parameters;
    advance state machine to next state;
    if(message->bufferingMode == ReceiverBuffered){
      ret = Nat_endRecv(message->recvCookie);
      copy message body from message buffer to application buffer;
      message->recvCookie = Nat_beginRecv(message->buffer,
                                          message->maxSize,
                                          message->msgID);
      return ret;
    }else{
      return Nat_msgwait(cookie);
    }
  }else{
    return Nat_msgwait(cookie);
  }
}
```

Figure C.4: Pseudocode for the `endRecv` operation.

```
void beginPattern(int patternNum)
{
  check patternNum for legality;
  currentCommunicationPattern = descriptor[patternNum];
  initialize state machine for new pattern;
  Nat_barrierSynch();
}
```

Figure C.5: Pseudocode for the `beginPattern` operation.

```
void endPattern(int patternNum)
{
  patternNum = NULL;
}
```

Figure C.6: Pseudocode for the `endPattern` operation.

```
void initPrototype()
{
  for(all messages){
    if( (message is destined for this processor) &&
        (message->bufferingMode == ReceiverBuffered) ){
      message->recvCookie = Nat_beginRecv(message->maxSize,
                                          message->buffer,
                                          message->msgID);
    }
  }
  Nat_barrierSynch();
}
```

Figure C.7: Pseudocode for the `initPrototype` operation.