Compiler Directed Architecture-Dependent Communication Optimizations

Susan Karen Hinrichs

June 1995

CMU-CS-95-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Thomas Gross, Chair
David R. O'Hallaron
Peter Steenkiste
Robert Schreiber, RIACS, NASA Ames

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, and in part by the Air Force Office of Scientific Research under Contract F49620-92-J-0131.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, the Air Force, or the U.S. Government.

Keywords: Data parallel compilation, communication optimizations, communication resource management, distributed memory systems.

Abstract

Communication required for distributed data structures is one of the major overheads of parallelization. Poor communication performance limits the the scalability of a given program, i.e. the number of processors that can effectively work on a problem.

Differences in the architectures of different parallel systems affect how communication models perform on these systems. To minimize communication overheads, the parallel application should use the communication model that performs best on the target machine. For many systems, a static communication resource reservation model enables more efficient resource usage and improved communication performance.

Data parallel compilers are well-suited to the task of selecting the appropriate communication model for the target architecture. With communication pattern analysis and architecture information, the compiler can optimize communication within each communication step and between communication steps. Knowledge of the target communication resources and all communication patterns enables the compiler to schedule use of limited resources and eliminate redundant control information.

This thesis describes the implementation of a communication generation and optimization phase in the Fx compiler. The effect of these communication optimizations are evaluated on a suite of programs. Targeting alternative communication models can reduce communication time up to 50% and total execution time by 10 to 20%.

Acknowledgements

Many people have helped me along the way making it possible for me to finish this thesis. My family and in particular my mother have been very supportive and made me believe that I am capable of doing anything I set my mind to. Teachers in high school and college have also influenced me to look at the possibilities beyond central Illinois.

Thomas Gross has been a very good advisor by keeping me directed but also reminding me to keep an eye on the big picture. I have been very fortunate to work with interesting groups in the iWarp and Fx projects. The people and work performed in these groups have been essential to my thesis work and technical development in general. In particular, discussions and collaborations with Tom Stricker, Jim Stichnoth, Tom Warfel, and Michael Hemy have been very fruitful. This thesis work also benefited from discussions with the iWarp group at Intel and the SUNMOS development team at Sandia National Labs.

Comments of my committee members have greatly improved the presentation of this document. Thanks to Rob Schreiber for reading the document very carefully and presenting views from outside the Carnegie Mellon environment. Peter Steenkiste provided early comments and made good suggestions for basic structural improvements in the document. Discussions with Jaspal Subhlok improved the presentation of the synchronization elimination algorithm. Dave O'Hallaron suggested the image analysis evaluation program.

The people in the Carnegie Mellon environment have been important for me growing technically and keeping my sanity these past six years. Thanks to the dinner co-op members for maintaining some semblance of a social life, my cultural husband Bob Doorenbos for taking me to the opera and the Y concerts, and my officemates: Peter Stout, Hugo Patterson, Tom Warfel, and John Hampshire for the random bull sessions in the office.

Finally, thanks to my husband Alan for being there and making me remember that there is more to life than this thesis and the academic world. His unconditional love and support is very important.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Communication is the price of parallelization. Poor communication performance adds to the parallelization overhead and reduces scalability. Therefore, good communication performance is important for good overall performance and scalability.

Though the basics of communication are the same on all distributed memory machines, there are many architectural differences, such as routing support and network interface design that dramatically affect how communication models perform on different machines. For good performance, the communication in a parallel application must be optimized for the target architecture.

It is unreasonable to assume that a human programmer will keep track of the details of different architectures to reoptimize the communication in an explicit message passing program for each new parallel architecture. In contrast, parallelizing compilers are better suited to keeping track of the details of different target architectures. The programmer has a global view of data movement required in the parallel program, but with communication analysis algorithms, the parallelizing compiler can also derive this global view of communication from data parallel programs.

For ease of retargeting to different platforms, most data parallel compilers generate communication code for standard message passing interfaces such as PVM[Sun90] or MPI[MPI93]. However, using a common representation for communication limits communication optimizations much the same way that generating common P-code limits uniprocessor optimizations. While the communication library can be optimized for each node of the target system, such message passing libraries do not have global information about the communication patterns, limiting the possibilities for optimization. Some message passing standards do define interfaces for common collective communication steps (e.g. MPI), but the library approach does not have the information about the sequence of communication patterns in the program that is available to the parallel compiler. With this sequence information, the compiler has more information to perform resource reservations and other communication optimizations.

Most distributed memory machines can support other communication models in addition to the standard message passing interfaces. In particular, this thesis examines communication models that differ in the method of communication resource reservation. Many communication models (including the standard message passing interfaces) rely on dynamic resource reservation for flexibility and portability. Other communication systems allow static reservation of hardware and/or software communication resources. With application-specific communication pattern information, a system that supports static communication resource reservation can generate more efficient communication code than the general message passing system. By using information about the communication patterns, the compiler has greater freedom for routing and communication resource reservation. In the dynamic case, the system must avoid deadlock in the face of unknown communication patterns, so it must follow a deadlock-free resource allocation strategy.

While alternative communication models enable improved communication performance on most parallel machines, these models are not appropriate for the human programmer. Unlike the general message passing models, the alternative models may not be supported on all distributed memory machines, so the optimized applications may not be easily moved to new machines. Also, some alternative models require compile-time or global runtime information for correct operation. If the programmer makes an error, a static resource reservation communication model does not degrade gracefully. For example, errors will occur on iWarp if there are not sufficient resources to open all requested connections. By contrast, the dynamic, general purpose message passing system is designed to rely only on local, runtime information, so the communication will succeed even with bad compile-time assumptions.

These issues are not problems for a data parallel compiler. The automatic code generator (once properly specified and debugged) will not make improper compile-time assumptions, so the robustness in the face of runtime errors is not necessary. The generated communication code cannot be directly retargeted to other parallel machines, but once the compiler is retargeted, all code it generates is also retargeted.

1.1 Thesis statement

The data parallel compiler can use information about the target architecture in addition to information about the required communication patterns to generate and optimize significantly more efficient communication code for alternative communication models.

1.2 Compiler structure

Other work has concentrated on developing algorithms to use or manage communication in a *connection-based* static resource reservation communication model[FSW93, Hin95, Gre93, War95]. This thesis is more interested in how the choice of the resource reservation strategy affects communication code generation in the data parallel compiler as a whole.

I explored these effects with a systems approach by building an alternative communication code generation path. The requirements of the compiler phase guided the problems and issues I addressed. I implemented the prototype communication generation phases in the Fx compiler[SSOG93], which operates on a variant of High Performance Fortran (HPF). I concentrated on optimizing dense linear algebra problems.

The Fx programmer creates a single-threaded program that operates over a logically global address space. Figure 1.1 shows an example Fx array statement. This statement stores the sum of elements `A(i)` and `A(i+1)` into element `C(i)`. The Fx compiler is responsible for incorporating the necessary distribution and communication code for execution on a distributed memory machine.

```
C(1:n-1) = A(1:n-1) + A(2:n)
```

Figure 1.1: An example Fx array statement.

Figure 1.2 shows the major phases of this compiler. The thesis work is concentrated in the communication phases from data placement through to dynamic control elimination. For data parallel programs, the parallelism in the program is guided by the placement of data arrays over the processing nodes. In the data placement phase, the compiler uses a combination of user directives and program analysis to find an assignment of data to processing nodes with a reasonable tradeoff between parallelism and communication. The communication map phase uses the data placement information to derive a concise representation of all communication patterns in the program.

For example, after data placement, the compiler knows that the array statement in Figure 1.1 requires communication before it can execute (assuming that the arrays are distributed). After the communication map phase, the compiler knows that a nearest-neighbor shift communication pattern is required.

After the communication map phase, the program intermediate representation (IR) is annotated with communication maps at the locations where communication is necessary. This IR is input to the simple communication code selection phase along with information about the target communication architecture. The compiler uses the communication maps and architecture information to select the best communication method for each communication step. The IR is annotated with this information for use in the code generation phase.

In the array statement in Figure 1.1, the communication code selection phase decides what is the best method for performing the nearest-neighbor shift on the target architecture. Depending on the type of hardware support, different communication models or different communication algorithms may be most appropriate.

For parallel architectures that enable software control of hardware communication resources, more sophisticated algorithms are better able to trade between communication implementations that rely on static resource reservation and implementations that rely on dynamic resource reservation. In this case the resource management communication code selection phase can find a better assignment of communication implementations. This resource management approach relies on information about the control flow of the program to manage the use of limited communication resources. For example, the array statement in Figure 1.1 may be part of a larger program, and another statement may require a more expensive all-to-all communication that is executed more frequently. If there are not enough communication resources to implement the faster, static resource reservation implementations for both patterns, the resource management algorithm selects the faster implementation for the more expensive all-to-all communication to optimize the program's overall performance.

Optimizations can also be performed on communication implementations in the dynamic resource reservation model. Communication in this dynamic model relies on synchronization and/or control messages to ensure that data does not arrive at the destination buffer too soon. Sometimes explicit control messages are not necessary to maintain this ordering. Instead data exchanges from previous communication steps can implicitly carry this ordering information. The dynamic control elimination phase uses control flow and communication pattern information in the program to determine where explicit synchronization is unnecessary.

The computation code optimizations can take place before or after the communication optimizations. Examples of important computation code optimizations include loop fusing, loop interchanging, and other standard loop transformations. Loop nests that result from array assignment statements tend to have the shortest distributed loop on the innermost iteration, so these loops particularly benefit from loop interchanging.

After communication and computation optimizations have been performed, the code generation phase translates the IR into a single Fortran 77 node program with communication calls in the single program multiple data (SPMD) style.

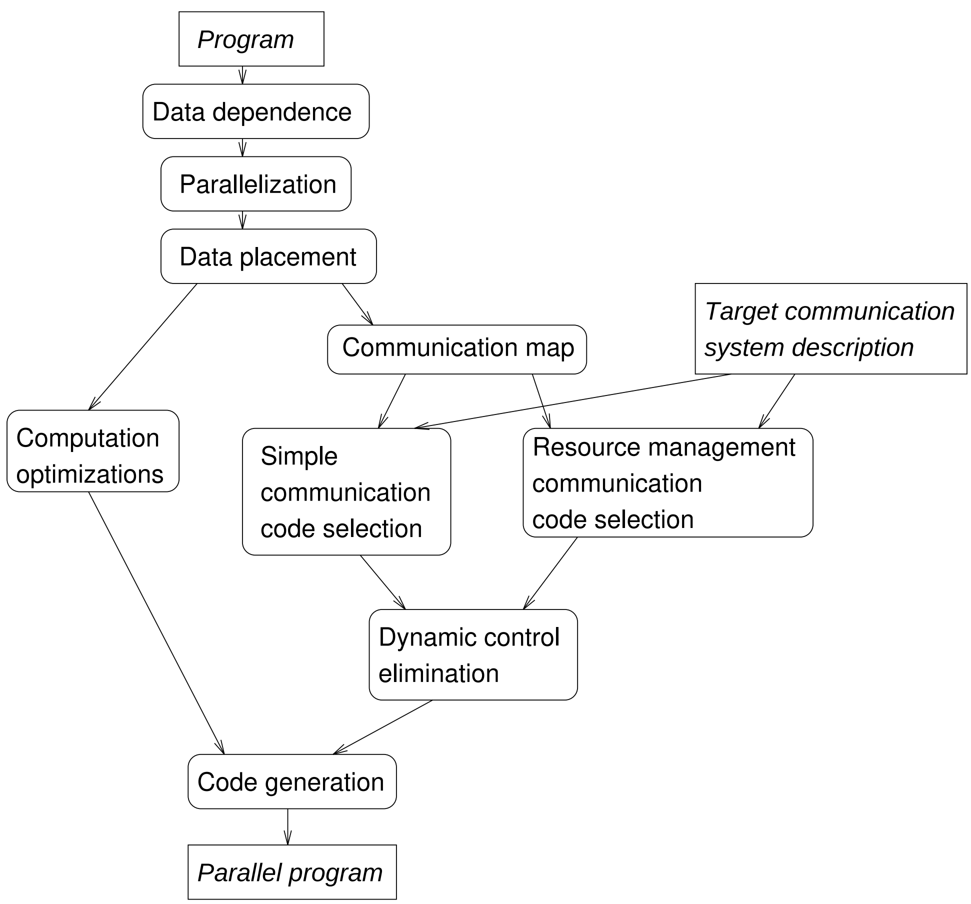


Figure 1.2: Flow chart of phases in a data parallel compiler. The work in the thesis is concentrated in the phases from data placement through to dynamic control elimination.

1.3 Thesis contributions

The contributions of this thesis can be divided into three main categories: implementation and experience, algorithm development, and evaluation. The most tangible contribution of this thesis is the implementation of communication analysis, code selection, and optimization phases in the Fx compiler. By implementing these phases, I gained insight about what previously proposed algorithms work, what additional algorithms are needed, and what optimizations are truly useful.

I implemented the communication analysis phase utilizing recent work in automatic data placement algorithms. While there has been much theoretical work in this area, there has been little empirical evaluation. By implementing a communication analysis phase, I found which techniques are useful for communication pattern extraction. Based on my practical experience, I developed a hybrid data placement. This approach enables the programmer to indicate how key array occurrences should be distributed, and the analysis routines then determine how the remaining array occurrences should be placed.

The communication patterns that result from the communication analysis phase are distilled into a concise linear mapping format I developed. Later phases use the structure of these maps to simplify matching predetermined communication pattern templates and to simplify optimizing sequences of communication patterns.

Using the results of the communication analysis phase, I developed two communication selections phases that target communication models relying on either *static* or *dynamic* communication resource reservation using information about the target architecture. One code selection algorithm is quite simple. The other code selection algorithm is more sophisticated and is able to make more intelligent tradeoffs between resource use and total program performance. These phases currently generate code for two different distributed memory machines: iWarp[B+88] and Paragon[Div91].

The resource management code selection algorithm addresses issues of targeting the static communication resource reservation model. The dynamic control elimination phase addresses optimizations in the dynamic resource reservation model. The dynamic model requires additional synchronization in some cases. In other cases, the synchronization is implicitly preserved from previous communication steps. To improve the performance of the dynamic model, I developed a data flow algorithm that determines when additional synchronization is redundant.

To evaluate the benefits of these communication selection and optimization phases, I measured the performance of a set of dense linear algebra programs executing on iWarp and Paragon. While the set of evaluation programs is not exhaustive, these programs exhibit communication patterns that are frequently required in this class of programs. These measurements show that the model selection decisions indeed differ between the two machines, and that the optimizations do make a substantial difference in communication time and total execution time in most cases. However, these measurements also show cases where communication optimizations do not improve performance.

1.4 Other communication optimization approaches

Communication optimizations either reduce the amount of required communication or make required communication steps take less execution time. The communication reduction optimizations try to maximize parallelism while minimizing the amount of required communication. Several examples of these algorithms are described in greater detail in Chapter 3.

Once the amount of communication has been reduced to only the required communication steps, the remaining optimizations attempt to optimize the performance of these "necessary" data exchanges. Again, these optimizations can be divided into two camps. Since most data parallel compilers target standard communication libraries, most work on communication optimization deals with *architecture and pattern independent* optimizations. Blocking or vectorizing messages is one common example of such an optimization[Ger90, C^+92]. By blocking messages, several smaller messages headed for the same destination are combined into one larger message. This improves performance particularly when the communication start-up time is very high. However, if communication start-up time is low, blocking messages may be detrimental to communication in performance (as explained in Section 2.3.3).

For previous generations of parallel systems, the startup overheads of the communication hardware were so high that optimizations beyond blocking messages made little difference. More recent parallel systems require that these communication assumptions be changed. To take advantage of the speed of these new systems, the programmer or compiler must be willing to use global information about communication patterns and look at alternative communication models.

Several groups have started looking at pattern-dependent optimizations. Li and Chen[LC91a] have described a method to recognize a variety of communication patterns. They propose matching the communication patterns discovered in a program with optimal implementations from a target architecture library. Similarly, Gupta and Banerjee [GB92a] describe analysis techniques to recognize a wider range of communication patterns. In both cases, the work does not discuss the communication models used in architecture-specific implementations or describe how the sequence of communication patterns can interact. This thesis integrates those architecture-specific aspects.

Other groups have also looked at architecture-dependent issues. The stencil compiler for the Connection Machine is one early example[BHMS91]. The stencil compiler recognizes simple shift communication patterns and takes advantage of the fact that the CM-2 architecture can simultaneously send to and receive from the four nearest neighbors.

With his protocol compiler, Felten looks at communication optimization by creating specialized message passing protocols for specific communication patterns[Fel93]. He also limits his domain to data parallel programs but describes his communication protocols in a bottom up fashion as a set of source/destination pairs. The protocol compiler looks at the interactions of different communication patterns to create safe, specialized message passing protocols. By contrast, this thesis uses communication maps to present a top down, structured view of communication that makes it easier for the compiler to directly control communication resources. Instead of working from the standard message passing interface, my communication generation algorithm attempts to bypass the standard message passing library entirely.

Similarly, Islam's communication tool set helps programmers create message passing protocols that are specialized for particular communication patterns[Isl94]. This tool set also concentrates on communication using a traditional message passing protocol. Instead of using information from the compiler, the tool set requires information directly from the programmer about the required communication patterns.

1.5 Dissertation road-map

Chapter 2 begins by presenting background information and thesis assumptions. This chapter presents an overview of the data parallel compiler model and describes how communication is incorporated in this model. The high level communication patterns derived in the compiler model must be mapped to the target architecture, so this chapter also examines the interaction of communication performance and architecture options. Finally, the background chapter introduces two communication models: the static and dynamic communication resource reservation models. This chapter defines variations on the two models and describes how these models perform given different performance characteristics. The performance of these models are evaluated on two target systems: iWarp and Paragon.

Chapter 3 describes the communication analysis phase of the Fx compiler (the Data placement and Communication map phases of Figure 1.2). This phase uses automatic data alignment and distribution algorithms to determine how data arrays should be placed on the target array to maximize parallelism and minimize communication. After the data placement phase, the communication pattern information is distilled into communication maps. When the communication analysis phase completes, the program graph is annotated with the required communication maps.

Chapter 4 presents the simple communication code selection phase of the compiler (the Simple communication code selection phase of Figure 1.2). This phase uses information about the target system to chose the best communication model for code generation for each communication pattern. This chapter presents a code selection algorithm and examines the benefits of this algorithm by presenting program measurements on iWarp and Paragon.

For systems that allow software control of a limited set of hardware communication resources (such as iWarp, ATM switches, and Transputers), a resource management approach to code selection is preferable. Chapter 5 describes limited communication resources on several systems. This chapter formalizes the limited communication resource problem and presents several simple algorithms that trade off resource limitations and performance improvements. The compiler can use these algorithms to make reasonable decisions about using communication implementations that statically reserve limited communication resources (shown as the Resource management communication code selection phase in Figure 1.2). I evaluate the effectiveness of these algorithms on iWarp, a system that allows software control of hardware communication resources.

The dynamic resource model can require additional synchronization to ensure correct operation, but in some cases the synchronization is implicitly maintained by previous communication patterns. Chapter 6 presents a data flow algorithm that determines when additional synchronization is redundant. This data flow algorithm is implemented in the Dynamic control elimination phase shown in Figure 1.2. The effectiveness of synchronization elimination is evaluated on

Paragon, a system with relatively costly barrier synchronization overheads.

Finally, Chapter 7 summarizes the dissertation. I describe how this work fits in the current state of research and suggest ways to extend this work.

Chapter 2

Model and architecture background

Before describing the compiler structure and communication optimizations, this chapter presents background information on the models used and the architectural issues discussed in this work. First, the chapter describes the basic data parallel compiler model on which the communication optimizations are built. Then it presents the main features of communication architectures and discusses how architectural options affect communication performance. The chapter finishes by describing two communication models, showing how each model relies on different aspects of the communication architecture.

2.1 Compiler model

Over the last decade many data parallel languages have been developed and studied, including Connection Machine C*[RS87] and Fortran[AKLS88], AL[Tse89], Data Parallel C[HQL+91], DINO[RSW91], NESL[BHS+94], and High Performance Fortran (HPF)[For93].

Programs in the data parallel model exploit parallelism by performing independent operations over collections of data elements. The degree of parallelism in such element-wise operations is limited by the size of the data set. For most data parallel languages, the parallel collection is an array or a subsection of an array. In HPF, programs operate with subsections of arrays called *slices*. An array slice is described by a lower bound, an upper bound, and a stride for each array dimension. For example, the array slice `A(2:n:2)` describes the even elements of array `A` from 2 to `n`. If slice arguments are missing, defaults of the array range are assumed for the lower and upper bounds, and 1 is assumed for the stride. Thus, `A(:)` describes all elements of array `A`.

Data parallel languages were originally developed for single instruction multiple data (SIMD) architectures, since one statement in the data parallel program can affect data on many different processors. Data parallel languages have been developed for SIMD architectures from the Illiac to Thinking Machine's CM-2. More recently data parallel programs have been targeted to multiple instruction multiple data (MIMD) architectures. The data parallel compiler translates the single data parallel program into a node program that is executed in multiple processes in the single program multiple data (SPMD) style.

The data parallel programming model eases the task of communication analysis for a compiler by providing a global name space. The data parallel compiler translates each data parallel statement into a communication step and a computation step, so the compiler views the program as a series of alternating communication and computation steps. With the proper analysis tools [CGST93, KLS90, Who91] or user directives[For93], a data parallel compiler can discover exactly what communication must take place in each step and describe these communication requirements as sequences of *communication patterns*.

A *communication pattern* is a set of node-to-node communications that form a system-wide communication step.[1] These communication patterns are either *regular* or *irregular*. The structure of a regular communication pattern is only dependent on the structure of the program. Many scientific and signal processing programs need only regular communication patterns[LG94].

A shift is a simple example of a regular communication pattern. Assume the nodes have a linear ordering. In a "shift right" pattern, each node sends data to its right neighbor, so processor p sends to processor $p + 1$. Other examples of regular communication patterns are reductions, transposes, all-to-all communications, scatters, and gathers. If the communication pattern is regular, each node performs the same set of communications and requires the same resources (in the form of bandwidth, buffers, etc.).

The structure of an irregular communication step relies on the particular data set in addition to the program structure. For example, programs that use finite element methods often encode the finite element mesh in a separate data array and rely on array indirection to implement the mesh division. Without pre-running the program or using external knowledge about the contents of the mesh division array, it is difficult for the compiler to analyze the communication patterns required in such a program. Also, refinement or load balancing techniques may change the communication pattern during the course of execution. In some cases with domain information, even irregular communication patterns may be known at compile time[S^+92], and many adaptive programs have communication patterns that do not change frequently[GLS93].

The assignment of data to nodes defines the amount of parallelism that is available and the amount of communication that is required in the program. A good data parallel compiler must carefully trade available parallelism against unnecessary communication overhead. Section 3.1 describes the problem of data placement in greater detail.

For an example of compiler-derived communication patterns, consider a two dimensional fast Fourier transform (2D FFT). Figure 2.1(a) outlines the data parallel code for this program. First the program performs 1D FFTs over the columns. Then it performs a transpose and calculates the 1D FFTs over the new columns. Finally, it performs another transpose to return the data to its original array. If the columns of arrays `A` and `B` are distributed over the processor array, the parallel loops can be executed in parallel, and the transpose operation requires an all-to-all communication pattern shown in Figure 2.1(b).

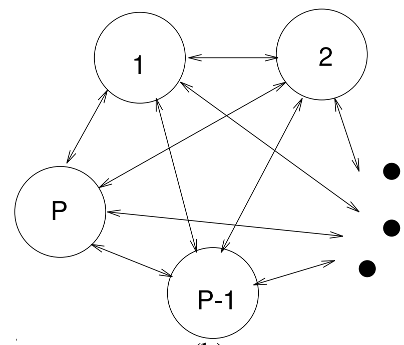
2.1.1 Thesis assumptions

To limit the scope of the thesis problem, I make several assumptions about the compiler. Most of the issues I omit have been addressed elsewhere or are natural extensions of the thesis results.

[1] The communication pattern can be defined over virtual or physical processors. Eventually, the virtual processors are mapped to physical processors, so for the sake of simplicity, this thesis considers only the real communication between physical processors.

```
parallel do i = 1, n
   fft(A(:,i))
enddo
transpose(A,B)
parallel do i = 1, n
   fft(B(:,i))
enddo
transpose(B,A)
```

(a)



(b)

Figure 2.1: Basic operations of the 2D FFT program.

This thesis work augments the Fx compiler[SSOG93] which compiles a variant of High Performance Fortran (HPF)[For93]. The input language supports array statements, parallel loops, and directives for data alignment and distribution. The data alignment directives support the full range of block, cyclic, and block-cyclic distributions. For implementation simplicity, this work concentrates on block distributions. The optimizations described by this thesis should naturally extend to cyclic and block-cyclic distributions. The cyclic and block-cyclic distributions differ from the block distribution in the endpoint data scattering and gathering, but the communication maps calculated in the communication analysis phase can be directly augmented to describe the communication patterns for the cyclic and block-cyclic distributions (Section 3.2.4 describes some of these extensions for cyclic and block-cyclic distributions). The distribution patterns will differ in the order of packing and unpacking data, but the same methods should be applicable for calculating and manipulating the communication maps.

This thesis does not address the issues of separate compilation or inter-procedural analysis. While these are both necessary features for a production compiler, these problems have been addressed elsewhere[C$^+$87, A$^+$88, HHKT92, PS92, RG89] and are orthogonal to the main issues of this thesis.

This thesis work concentrates on optimizations in the "pure" data parallel model. Combining task and data parallelism has generated a lot of interest recently[SSOG93, Fos94]. Data parallelism alone requires scaling a problem to increase parallelism. However, in the real world, it is not always practical to scale the problem size. By dividing the machine between a set a cooperating parallel tasks, tasking can add another dimension of parallelism.

Finally, this thesis concentrates on optimizations for regular communication patterns. Many important problems require communication patterns that cannot be analyzed at compile time, but compiler techniques for addressing irregular communication patterns at runtime (much less compile-time) are still an active research topic[GLS93, Sti94, SAS92, BW93]. Once we understand how to retarget the simpler regular communication patterns to other communication models, perhaps similar techniques can be used at runtime for irregular communication patterns.

2.2 Communication architecture

Details of the communication architecture affect communication performance. Before we can hope to optimize communication, we must have a good understanding of the interactions between communication architecture design and communication performance. This section presents a generic, distributed memory machine model and defines several key communication performance characteristics. Then it describes implementation options for several key communication architecture choices and discusses how the implementation specifics affect the communication performance characteristics.

2.2.1 Distributed memory architecture

We begin the discussion of communication architecture choices by defining a model of a generic, distributed memory computer. The major units of this model are shown in Figure 2.2(a). It models both distributed shared memory systems and private memory systems.

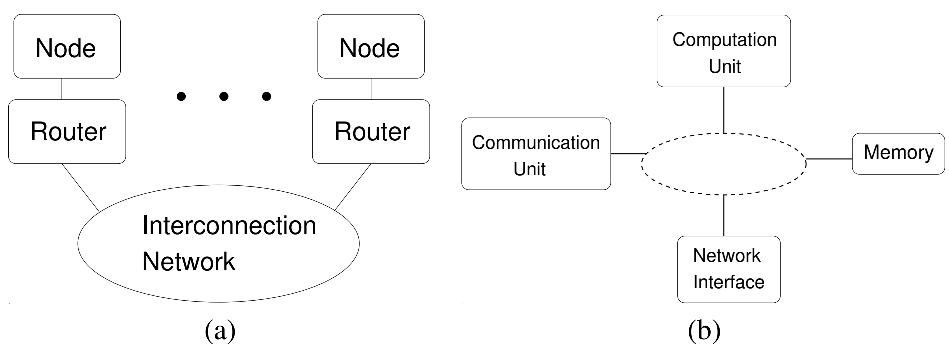


Figure 2.2: An abstract view of a distributed memory machine. (a) Major units of the generic machine. (b) Elements of a generic node.

All distributed memory systems are made of sets of *nodes*, *routers*, and *communication links*. Figure 2.2(b) shows the generic components of a node. The node contains the *computation unit* and the memory of a normal uniprocessor system. The node also contains support for communication in the *communication unit*, which is responsible for injecting and extracting messages.

The routing units and the communication links form the interconnection fabric of the distributed memory system. The links are reliable, high speed networks. The routing units interpret message headers to forward the messages along the appropriate link to their final destinations.

With some variations, today's distributed memory systems contain these basic components. In some systems, the computation and communication may be handled in the same unit (e.g. Intel iPSC), or communication and routing may be implemented by the same unit (e.g. iWarp). Each node can also contain multiple computation units (e.g. Cedar[K^+93], Paragon-MP3 (a three processor version of Paragon)), so each node can be thought of as a bus-based, shared

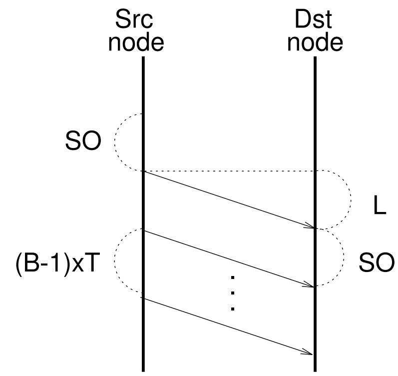


Figure 2.3: A sketch of a message transfer divided by the different performance issues. Time runs from top to bottom.

memory architecture. The number of routing units per computation node varies between different topologies. There may be one routing unit for each node as in the Paragon; there may be many routing units for each node as is the case for multi-stage networks; or there may be many nodes for each routing unit as is the case for small cross-bar networks.

The particular topology of the distributed memory system affects the degree of network congestion and message latency. For this discussion, we ignore the specific system topology, because the same qualitative performance issues arise regardless of specific topology.

2.2.2 Communication performance characteristics

Given an algorithm and any distributed memory system, there are several performance characteristics that must be identified before the algorithm can be implemented efficiently. By abstracting away from the machine to these performance issues, several research groups have tried to create simple and realistic execution models for developing parallel algorithms (e.g. PRAM[FW78], bulk synchronous[Val90], and LogP models[CKP+92]).

Using this previous work as a base, we examine four communication performance characteristics: latency (L), communication start-up overhead (SO), bandwidth (BW), and overlap loss (OL). Figure 2.3 labels the time line of a message exchange with these parameters. Latency (L) is the time from when the first word of a message leaves the source node to when it reaches the destination node. Startup overhead (SO) is the time from the start of communication to the time the first word of the message is injected to or extracted from the network. This includes software overheads, such as interrupt handler or message polling, buffer selection, and buffer copying delays. Bandwidth (BW) is a measure of the rate communication between two nodes. We also discuss the inverse of bandwidth, the per byte transfer time (T). The total time to send or receive a B byte message is $2 \times SO + L + (B-1) \times T$.

The computation unit may be involved in the communication even after the first word of the message is injected (extracted). For example on iWarp, the computation unit can directly inject data at the full network bandwidth. Sometimes direct computation unit control is preferable to using a direct memory access (DMA) controller or communication co-processor, but if the computation unit is injecting (extracting) data, it cannot be performing other computations; the program loses the opportunity to overlap communication and computation. The overlap loss parameter (OL) is a measure of the time the computation unit is performing communication and quantifies this loss of opportunity for overlap. In the example shown in Figure 2.3, the OL measure for the sending node is either SO if the computation unit is not injecting data, or $SO + B \times T$ if the computation unit is injecting data.[2]

These four parameters adequately describe endpoint costs and transmission costs assuming there is no contention for network resources. A message traveling through the network can temporarily delay other messages, causing congestion and reducing the effective bandwidth. To be complete, the abstract communication model must also include a measure of network congestion, but network congestion is dynamic and quite hard to predict. Therefore, no static performance model adequately describes network congestion, and most understanding of network congestion comes from empirical simulations and experiments.

2.2.3 Implementation options

There are many implementation choices to be made when moving from the generic machine model to a real system. This section outlines several of the key decisions and describes how the implementation choices affect the four performance characteristics identified in the previous section.

Transfer granularity The communication subsystem must make choices about what limits to place on the amount of data to transfer as one unit. In the simplest case, the communication subsystem can transfer the same amount of data in each message, usually a small number of words. This is the approach used by ATM switches, the CM-5[L$^+$93], and most distributed shared memory systems. While this approach is simple, many programs want to send larger messages, so packetization software or hardware must be added. This adds to the complexity of the system and potentially adds to SO and/or OL.

Other systems allow large packets by defining a large upper bound on packet size. Messages smaller than the bound are sent directly, and larger messages are divided into several packets. In this approach not all messages are the same size, but there is a bound on the message size. Under OSF1, the Paragon takes this approach by setting a system-wide packet size limit. Assuming a packet size of P, the time to send a B byte message is $\lceil B/P \rceil (SO + L + min(P, B) \times T)$.

To minimize the impact of SO, the system designer wants a large packet size, but in congested communication patterns, small packets are better able to use the system's bandwidth (BW). While a packet uses router resources it prevents other packets from making progress through the network. Smaller packets release router resources more frequently, so other packets have a chance to make progress.

Unbounded, variable length packets are *streams*. Most systems that use streams actually have hardware to interleave multiple streams over a single physical bus on a word by word basis, i.e. support message packetization in hardware. Logical channels of the iWarp, virtual channels implemented in the T9000[MTW93], and virtual connections implemented for ATM

[2] This measure of OL is a simplification. Even if a separate unit (e.g. a DMA unit) is responsible for injecting the message, the computation unit will not be able to run at full speed in most architectures, so the true OL measure should be greater than SO. The injecting unit will steal memory bandwidth or bus bandwidth in most systems.

networks are examples of these interleaved streams. This option requires additional hardware support, but combines the best of the low overhead of large messages and the good congestion performance of small messages.

Transfer method Once the transfer granularity is selected, there are several options for transferring data over the network.

Store and forward is the simplest option. The entire packet is buffered at each router which increases latency. This approach is used in the earliest distributed memory systems and in systems that use small, fixed size packets. For small, fixed-size packets, the entire message can be stored in hardware buffers at the intermediate routers, but large packets require more memory at each hop. Unless the memory buffering speeds can match network bandwidth, store and forward will reduce effective bandwidth.

Wormhole routing is used by most current systems that transfer large or variable length packets. In wormhole routing, portions of the packet may be stored in buffers on several routers, so the message stretches out like a worm on the network. This reduces the overhead of waiting for the entire message at each router. One packet is capable of blocking many messages in this scheme, because it can consume resources over several routers. This increased blocking can increase network congestion and reduce effective bandwidth.

Virtual cut-through[KK79] and *segment* routers[Kon94] have been proposed as a compromise between the low overhead wormhole routing and the low congestion store and forward. In virtual cut-through routing, the packet follows a wormhole routing scheme while it can make forward progress. When it cannot reserve resources on the next router, the packet is gathered into the current router, releasing resources on all but one router. This works well for short messages but requires large buffers on the routers for large messages. Segment routers divide message traffic into short and long messages and use separate routing methods and resource pools for each class. It uses wormhole routing for long messages and a form of store and forward routing for short messages.

Synchronization During communication there must be some degree of synchronization between nodes and routers from the source node to the destination node. Nodes and routers must negotiate to ensure that there is space in the next router for the data. For most high-performance interconnection networks, this synchronization or flow control is handled in hardware. For example on iWarp, two wires are added to the data path to signal when data is queued and dequeued on the the neighboring node.

For more distant nodes or less expensive systems, additional control wires can be prohibitively expensive. In this case, control messages and communication protocols are used to provide reliable service. The Transputer T9000 and C104 routing chips use end-to-end flow control for each virtual channel and avoid the need for additional control wires. No additional words are sent from the virtual channel until the currently outstanding word is acknowledged.

These alternatives for link-level flow control affect the network efficiency. Hardware-based flow control does not consume any network bandwidth but requires additional wires and circuits. Software-based flow control uses some of the data bandwidth for flow control.

At the application level, inter-node synchronization may also be required. Communicating nodes must negotiate to ensure that data is not sent before it is ready to be received, either by the application or by system software. This synchronization is dependent on the communication protocol, so the issues of application-level synchronization are discussed in greater detail in Section 2.3.

Injection/extraction point Decisions in the design of the interface between the computation unit and the network are very important to system performance. One of these interface issues involves where to insert or extract data from the network into the memory system. It is feasible to inject data into the network from any point in the memory hierarchy, and there are examples of designs and systems that use each of these levels.

The simplest option is to move data between main memory and the network. Since the network interface is memory-mapped, communication looks like loads and stores to the computation unit. Therefore, commodity processors or DMA's can be used as the communication unit with no modifications. Memory-mapped communication is simple, but it can add to the communication overhead (SO). If the computation unit directs communication, it must use load and store operations with the same overheads as loading and storing to main memory.

At the other extreme, the injection/extraction point can be mapped into the register space. This requires a major change to the design of commodity processors, but it enables very low overhead access to the network. iWarp uses register-mapped communication queues to inject data produced by the computation unit. Such low overhead access is required for systolic algorithms, and low overhead access gives the communication unit more options for data extraction and injection.

Placing the injection/extraction point at the second-level cache, trades off hardware modifications against efficiency[HJ92]. Many distributed shared memory designs use a cache-level network interface (e.g. Flash[K^{+}94] and Typhoon[RLW94]).

Injection/extraction agent In addition to deciding where in the memory hierarchy to inject and extract data, the system designer must decide what type of agent to use to actually inject and extract data from the network.

First generation systems used a single processor for both data injection/extraction and computation. While sharing a single processor reduces hardware requirements, the cost of switching between the communication and computation threads adds to the startup overheads (SO). Also, using a single processor for both communication and computation eliminates the possibility of overlapping communication and computation.

A DMA controller can act as a simple communication agent. After the computation unit sets it up, the DMA can move data between the network and memory without consuming additional resources of the computation unit. However, DMAs can only move contiguous blocks of data or single, strided sections of memory. This is often sufficient for explicit message passing programs, because the human programmer is normally cognizant of data layout in the program and will restructure the problem to simplify data transfers. However, many programs generated by parallelizing compilers must communicate data scattered over memory. Communicating this scattered data either requires first packing it into a contiguous piece of memory (requiring an additional copy) or setting up the DMA for each small contiguous piece of memory (paying the startup overhead of the DMA many times). Either option adds to SO.

Another option is to use a second commodity processor as the communication agent. This approach is used by Paragon; a second i860 is included in each node to take care of communication. However, the second commodity processor is not close to the network interface, so it must still rely on DMA controllers to move data at the full network bandwidth. The second processor can take care of the address calculation, buffering, and DMA startup overheads, decreasing the OL penalty on the main processor.

Machines that require fine-grained, low-overhead communication such as iWarp and Transputer systems use specialized communication co-processors. Designs for distributed shared memory systems either use hardwired engines (e.g. Dash[L^+92]) or specialized programmable engines (e.g. Flash and Typhoon) to control remote accesses.

Routing control Routing decisions determine the amount of effective system bandwidth. Systems differ in how routing is controlled. Many systems have a single routing strategy hardwired in the routing unit, so routing decisions can be made quickly. However, to place the routing strategy in hardware, it must be simple and general, and the "best" routing strategy may change based on the current traffic characteristics affecting the amount of usable network bandwidth.

Some systems allow semi-programmable routing strategies. For example, iWarp and ATM networks allow source controlled routing. Simple routing information is stored in the message header, so the source node can control which route the message should take. For this approach to avoid deadlock conditions possible in a general message passing pattern, the source nodes must follow a system routing policy.

The Transputer C104 routing chip also allows some programmability in the routing strategy. The routing chip uses a routing table which can be loaded by the application. The C104 routing table is discussed in further detail in Section 5.1.2.

Routing strategies Regardless of how flexible the routing control is, the system must choose at least one routing strategy. Most systems use simple, *oblivious* routing strategies. For example, the mesh-connected Paragon systems uses e-cube or row then column routing. In a oblivious routing strategy, the same route is always used between a particular pair of nodes.

Oblivious routes are repeatable and relatively easy to implement, but *adaptive* routing strategies may provide better use of network bandwidth. With an adaptive routing strategy, a number of routes may be used between a particular pair of nodes depending on local traffic conditions. There has been much study and simulation work on adaptive routing algorithms[GPBS94], but few current machines actually implement adaptive routing.

While adaptive routing can improve bandwidth usage, it destroys any assurance that messages sent between a pair of nodes will arrive in the order sent. This out of order message arrival increases endpoint overheads (SO) for re-ordering messages. If the program exchanges logically larger chunks of data than the system message size, either additional addressing information in the message headers or message re-ordering is required.

2.2.4 Thesis assumptions

By distributed memory machine, I mean a machine where local memory can be accessed substantially faster than non-local memory, regardless of the method of remote access. I also assume that the system is either dedicated to running a single parallel application, or the system uses gang-scheduling to time-share the system.

By concentrating on dedicated systems (dedicated either permanently or on a time-sliced basis), this thesis can avoid many of the issues brought up by operating system interactions. If the parallel system is shared between multiple applications either by space-sharing or time-sharing nodes, the operating system must manage shared system resources to insure the applications execute safely and fairly. On such systems, the network interface is a shared resource. The operating system must guarantee that messages coming in from the network reach the correct process. The operating system must also guarantee availability by ensuring that one process does not consume all available bandwidth at the cost of other processes.

These shared network resource management services can slow network communication by adding to communication startup overhead and hiding the details of the communication architecture. There has been some work to reduce the operating system communication overhead by shared user/kernel buffers (e.g. Fbufs[DP93]) and application-specific libraries (e.g. [MB93]).

The specific topology of the target machine is not important, but it is important that this topology is known at compile-time, so the compiler can use the topology information in its optimization decisions.

The thesis assumes the nodes are connected by an interconnection network with the following characteristics:

- The routers use wormhole or virtual cut-through routing not simple store and forward.
- The network is dedicated and reliable.
- The communication bandwidth in and out of a node and a node's computation speed are roughly balanced.

These network assumptions are what separates multi-computers from networks of workstations. Some claim that the coming of ATM switches and gigabit networks will make this distinction less clear over time.

The thesis also assumes that routing decisions are oblivious. Adaptive behavior limits the amount of compile-time analysis and reduces the strength of the assumptions that can be made about the communication patterns.

2.3 Communication models

The design of the communication system affects the performance of different styles or models of communication. These communication models can be described and separated along many dimensions. Differences in communication resource reservation can have a great effect on communication performance, so this thesis concentrates on differentiating communication models by their means of reserving communication resources. In particular, this section discusses the extreme models of *dynamic* and *static* communication resource reservation.

Supports?	Static	Dynamic
Communication resource reservation	Yes	No
Program controlled routing	Yes	No
Arbitrary runtime communication patterns	No	Yes

Table 2.1: Summary of the major differences between the features supported by dynamic and static resource reservation.

Table 2.1 highlights the major differences between these models. With static resource reservation, the communication system can take advantage of knowledge about communication patterns to reduce network contention and communication startup overheads. While the dynamic resource reservation has the flexibility to handle any arbitrary communication patterns. Each approach has its advantages, and different situations are best suited to different communication models.

Of course, this discussion is addressing extreme cases of managing communication resources. Many communication models or protocols rely on dynamic techniques at one level and static techniques at another. For example, the TCP is a connection-based protocol. It statically reserves buffers for each open connection, but TCP is frequently implemented on packed-based hardware that relies on dynamic techniques to reserve and release hardware resources for each packet of data exchanged. Section 2.3.3 discusses a hybrid implementation on the Paragon in greater detail.

2.3.1 Dynamic resource reservation

Most distributed memory systems support a message passing package that performs general communication such as the PVM library[Sun90] or Intel's NX message passing library[PR94]. These message passing systems are examples of communication that rely on dynamic resource reservation. No resources (e.g. bandwidth, software and hardware buffers) are reserved between message exchanges. Here we introduce several dynamic communication models. See [SSO$^+$95] for a more detailed discussion and comparison of these models.

In the *general* message passing model, both source and destination nodes explicitly make communication requests. A send statement on one node logically matches a receive statement on another node. By matching explicit send and receive statements, the message passing model guarantees a partial order; before one receive can complete, the matching send must occur. In addition to data transfer, the general message passing model provides application-level node synchronization.

Figure 2.4 depicts the control flow and buffer requirements for a general message passing data transfer; the vertical axis represents time. An incoming message must be buffered in a preallocated system buffer at the receiver node if the receive operation has not been invoked to accept the data.

The message passing library may also buffer messages on the sending side. With sender side buffering, the send call can return before the data has actually been sent, and the sending program can safely change data in the array that was just sent. For simplicity, we concentrate our discussion of general message passing on receiver-only buffering.

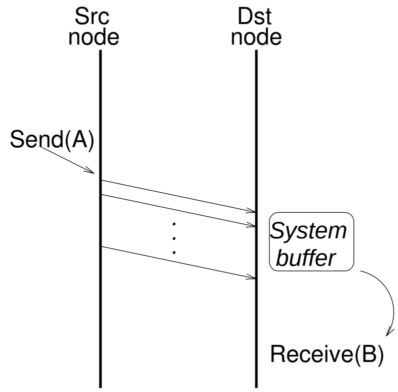


Figure 2.4: A timeline of a general message passing exchange. Time moves from top to bottom.

The *deposit* and *fetch* message passing models separate the data transfer issues from the application synchronization issues. Logically, the fetch and deposit models are duals of each other. In the fetch model, the destination controls the data transfer. The destination node sends a fetch request to another node, containing the addresses of the data elements to fetch. The source nodes replies to fetch requests by sending the requested contents of its memory unconditionally. Figure 2.5(a) shows the relevant steps in a fetch model data transfer. The destination node sends a request for items to the source node. The source node replies with the requested data. When the data arrives, the destination node stores the incoming data in its final location.

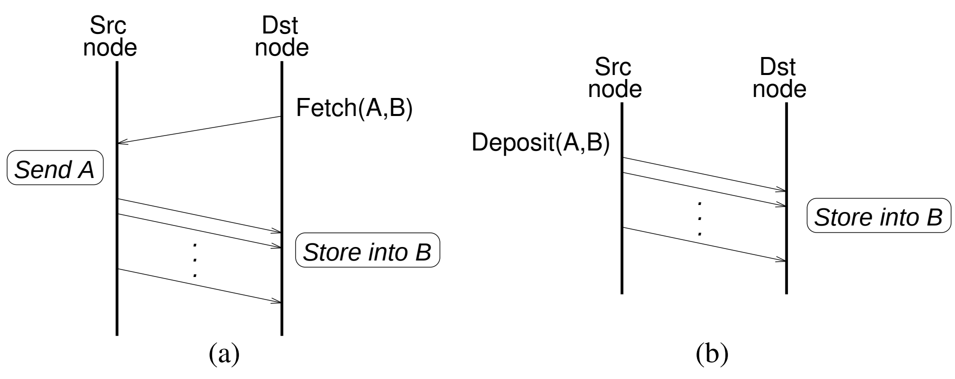


Figure 2.5: Basic operations of the (a) fetch and (b) deposit models.

In the deposit model, the source node controls the transfer. Only one message is required to transfer the data together with its destination address(es). The destination node then deposits the contents of the message based on the addresses provided. Figure 2.5(b) depicts the relevant steps in a deposit model data transfer. The source node sends the data together with the remote address(es). Then the destination node accepts the transfer and deposits the data directly into its final destination, without further involvement or computation on the destination node. The destination node requires a mechanism to determine when the data has been updated. Generally, a predetermined semaphore or update variable is also updated by the deposit message handler to indicate that the target buffer has been updated.

The fetch and deposit models only define the data transfer mechanism; they do not define the sequencing mechanism. Without an additional sequencing mechanism, data may be fetched before it has been computed, or data may be deposited before the destination is finished with the old data values. This sequencing information can be communicated through additional mechanisms such as control messages, barrier synchronization, or a tree of messages.

Depending on the sophistication of the communication unit, dynamic communication may require additional packing and unpacking steps. If DMA controllers are used for the communication unit, only contiguous or single-strided memory patterns can be transferred. Any more complex memory access patterns must first be packed into contiguous locations in memory adding to the startup overhead. If the communication unit has direct access to the network, it can use an index array or local address calculations (i.e. data chaining[SG95]) to directly inject or extract data in irregular memory access patterns.

2.3.2 Static resource reservation

Static resource reservation is an alternative on many distributed memory systems[FSW93]. A static resource reservation model has the potential for more efficient communication by allowing resources to be reserved for several messages.

Generally, resources are reserved along a *long-lived connection*. The resources are reserved once when the connection is initiated, so the cost of the resource reservation can be amortized over many data exchanges. Once the connection is set up, data movement over that connection requires no software intervention by any of the intermediate nodes. Wormhole and virtual cut-through routing also give these advantages to the dynamic resource reservation model, but the connection must be created and destroyed for each message. Long-lived connections can be used for multiple messages, so the start-up overhead (SO) is better amortized. Figure 2.6(a) shows messages sent on two long-lived connections. The resources for these messages are reserved initially at time T0. Then the data (shown as the shaded region can be sent over the connections for many time steps. By contrast, Figure 2.6(b) shows the same data exchanges in the dynamic resource reservation model. The resources must be reserved before each data exchange and released after each data exchange.

The program can communicate over the long-lived connection using send and receive functions similar to those used in the general message passing model. The deposit and fetch protocols can also be implemented over a static, connection-based model, but the connection-based model does not require buffering, so the benefits of the deposit and fetch protocols are not as apparent.

Regardless of the sending and receiving protocols, the overheads of the connection-based communication are far lower, since the communication resources have been statically assigned to the connection and do not need to be assigned with each data exchange. This lower overhead makes it reasonable to send smaller messages.

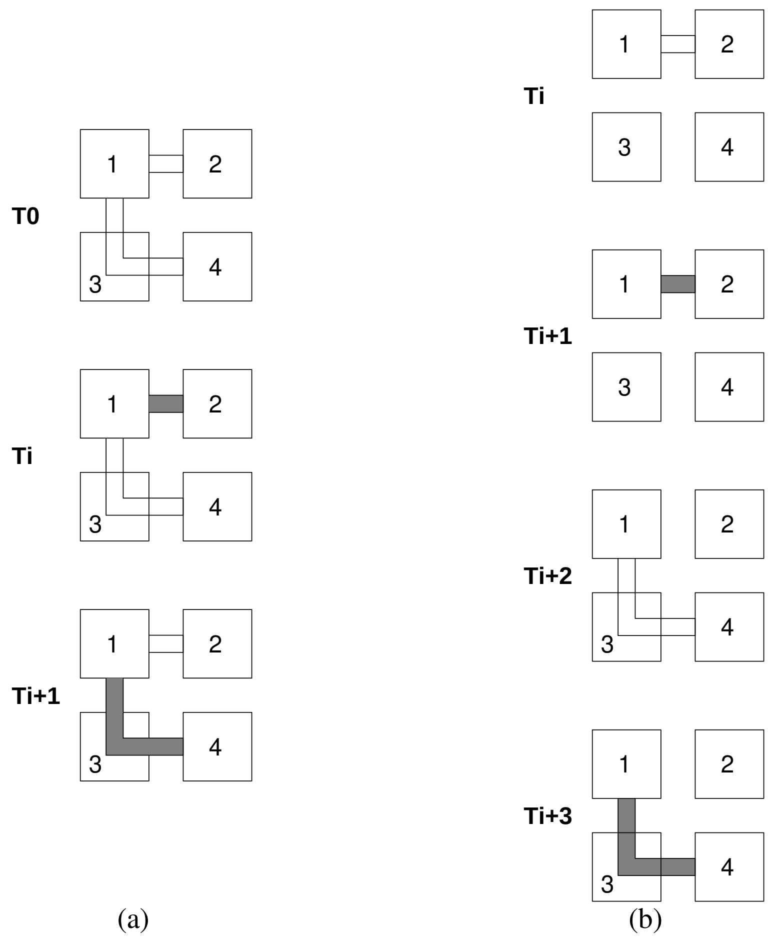


Figure 2.6: Messages sent in the static (a) and dynamic (b) resource reservation models. In the static, connection-based model, resources are reserved at time T0 for potentially many data exchanges. In the dynamic model, resources are reserved and released for each data exchange. Data is shown as the shaded regions.

Figure 2.7 shows the communication steps in one data exchange over a statically reserved connection. The connection-based model enforces a strict synchronization between the sending and receiving nodes. No data is sent until the corresponding receive is made. Depending on the implementation, the sending node keeps track of when the receiving node requests data through link level hardware flow control or software control messages. Section 2.3.3 describes the details of a hardware-based and a software-based connection implementation. Since the connection model enforces this strict synchronization, data communicated over connections never needs to be buffered in system buffers.

This strict synchronization between sender and receiver places a greater burden on the

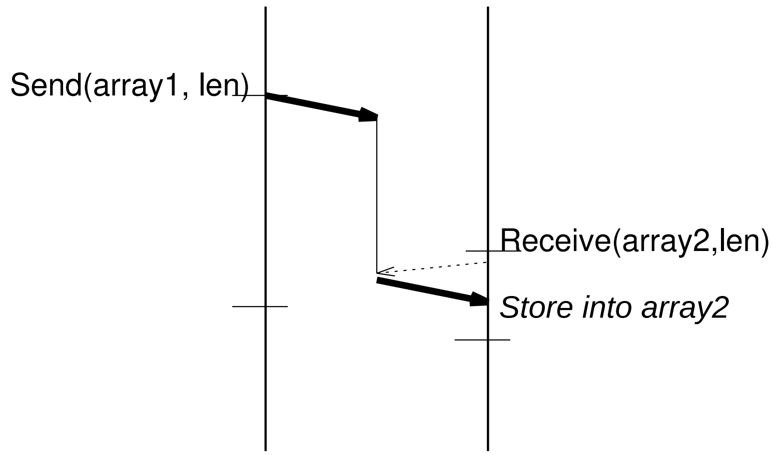


Figure 2.7: Schematic operation of connection-based communication.

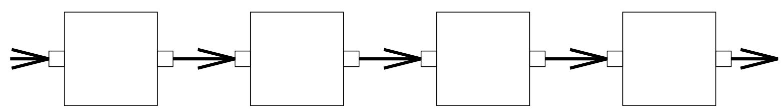
programmer to ensure that each node's communication schedule is deadlock free. For example, four nodes with connections shown in Figure 2.8(a) executing the code in Figure 2.8(b) will deadlock, because all nodes will stall waiting for the corresponding receive to be made. The code in Figure 2.8(c) will work since one node makes the receive call first.

Some systems allow non-blocking send and receive operations. Requests to send and receive data are queued, and the communication function returns before the communication has completed. In this case, several communication requests can be outstanding. Non-blocking calls can be implemented in the dynamic and static models, and non-blocking operations enables greater freedom in the scheduling problem. The code in Figure 2.8(d) shows a deadlock free program that uses non-blocking receives.

Each node of a system has a limited number of resources that can be statically assigned to connections, so only a limited number of connections can efficiently exist simultaneously. The specific communication resources are machine dependent. Some examples of communication resources are hardware buffers, software buffer space, or router space. Some systems may support an effectively infinite number of connections, but it is likely that the machine will support only a limited number of connections well. For example, ATM networks may only make quality of service guarantees to some subset of active connections.

One important observation is that programs go through different *phases* of communication patterns. When a particular communication pattern is not in use, its resources can be used for other patterns. Thus, the communication resources can be time-shared. This network phase switching is analogous to reusing registers via a context switch on a multiprogrammed uniprocessor. For example, Figure 2.9 shows a program that alternates between a phase that communicates using a hypercube pattern and a phase that communicates over a grid pattern.

Therefore, it makes sense to define sets of connections that operate in the same portion of the program in the same phase[Hin95]. At runtime, the program performs a *phase switch* to move between the phases. In a phase switch, the system performs a system-wide synchronization, reclaims the communication resources, and then instantiates a set of pre-computed connections for the new phase.



(a)

```
send_block(out_port, block)
receive_block(in_port, block)
```

(b)

```
if (my_node_id == 0)
   receive_block(in_port, block)
send_block(out_port, block)
if (my_node_id != 0)
   receive_block(in_port, block)
```

(c)

```
receive_block_nb(in_port, block, &status)
send_block(out_port, block)
while (status == 0)
```

(d)

Figure 2.8: Three node programs that exchange data over connections shown in part (a). Code (b) will deadlock because all nodes block on the send. Code (c) breaks this deadlock by introducing asymmetry on node 0. Code (d) breaks the deadlock by using non-blocking, background receive operations.

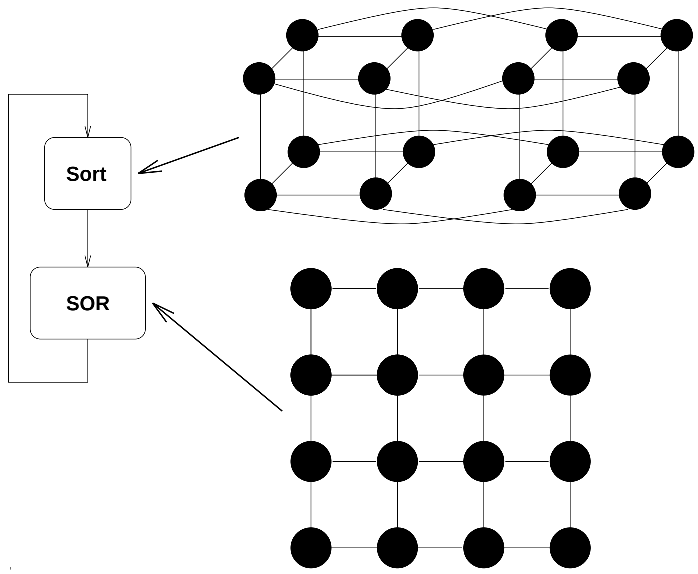


Figure 2.9: Program that alternates between two communication phases.

Switching between communication patterns is conceptually simpler than general connection creation, because all nodes open and close connections at a common point in the program. The compiler can statically assign resources to all connections of the phase in one step. Alternatively, connections can be created and destroyed individually, avoiding the distinct phase switches, but in this case, the compiler must ensure deadlock free routing by verifying that all possible sets of connections can co-exist

With dynamic resource reservation, routes must be limited to avoid deadlock[Str91]. With phases, connections are routed off-line with global knowledge, so they are not subject to these routing limitations. Therefore, the statically reserved connection model allows greater freedom for routing connections.

With static resource reservation, a single node can have several connections open at the same time. With several connections, a program can selectively read from different connections (and therefore different source nodes). In the dynamic model, the destination must service all data as it arrives, but in the connection-based model, the destination controls the communication, so it can chose to only service data from one connection and ignore data on the other connections until a later point in the program. For example in Figure 2.10, node 1 can chose to first process the data from node 0, before working on the data from node 2.

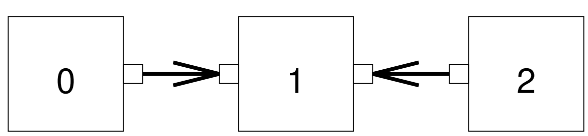


Figure 2.10: Node 1 can control whether to process data from node 0 or node 2 first.

iWarp

	Connection	Connection and DMA	Deposit	General
Start-up overhead (SO)	0	5 μs	10 μs + *sync*	20 μs
Per byte transfer (T)	0.025 μs	0.025 μs	0.025 μs	0.025 or 0.05 μs
Latency	$T \times D$	$T \times D$	$T \times D$	$T \times D$

(a)

Paragon

	Connection	Deposit	NX (general)
Start-up overhead (SO)	50μs	50μs + *sync*	50μs
Per byte transfer (T)	0.00625 μs	0.00625 μs	0.00625 or 0.05 μs
Latency	$T \times D$	$T \times D$	$T \times D$

(b)

Table 2.2: Performance characteristics of communication on iWarp(a) and Paragon(b), where D is the number of links from the source to the destination node. The per byte transfer time for the general message passing case depends on whether the message must be buffered. The deposit startup overhead includes synchronization to ensure that messages do not arrive too soon.

2.3.3 Model implementations

To make the differences between these communication models more concrete, this section describes implementations of the dynamic and static resource reservation models on iWarp[B$^+$88] and Paragon[Div91].

iWarp and Paragon are two substantially different parallel architectures. From the view of communication models, these systems differ on two main points:

- Software can control resource reservation on iWarp. Resource reservation strategies are hardwired on Paragon.
- iWarp supports program-controlled routing. Paragon routing control is hardwired in the routing chip.

Models on iWarp

iWarp is a distributed memory, parallel computer developed by Intel and CMU. Table 2.2(a) summarizes the performance characteristics of the static (connection-based) and dynamic (deposit and general) resource reservation implementations on iWarp.

The nodes in the system are connected in a two-dimensional torus by 40 MB/s busses. Each node of an iWarp system consists of two units: a computation and a communication agent. The two units are tightly integrated since they are implemented on the same chip. The computation agent is a loads/store architecture that operates at 20 MHz. It can execute long instructions that keep all three functional units busy.

For this work, the communication agent is the more interesting of the two agents. The communication agent was designed to support program-directed, systolic communication and message passing communication[B+90]. The low latency systolic-style communication is made possible by hardware support for *logical channels*. The logical channels are in effect register-mapped network queues, so data from the network can be read and written directly by the computation agent by reading and writing registers.

The program can control how the logical channels are configured, so the static, connection-based model maps naturally to the logical channel hardware. Logical channels can be chained together to form reconfigurable direct connections between nodes not physically adjacent in the system. Once the logical channel connection is set up, the connected nodes can communicate using a zero-overhead protocol. The network queue can be written or read once every two cycles. Logical channels reserve bandwidth. When data is traveling over the channel, it need not wait for bandwidth, but when the channel is inactive, its bandwidth is available for traffic on the other channels.

Since the node has direct access to the network, the overhead of accessing the network is very low. Therefore, the program can effectively send data without first packing it into contiguous memory locations. For some communication patterns, the connection-based implementation requires no intermediate data buffering.

For cyclic data distributions, this low-overhead communication can nullify the benefits of some common architecture-independent optimizations such as blocking or vectorizing communication. This optimization combines many small messages into fewer, larger messages (generally by pulling communication out of loops) which reduces the number of communication startups.

If communication startup is zero as is the case for communication over statically reserved connections on iWarp, it may be better to leave communication in the loop. With communication in the loop, data can *stream* through a node, and the program can use data directly from the network without storing it in memory. After moving communication out of the loop, the program must store the data in memory possibly performing more memory accesses.

Consider the sequential loop in Figure 2.11(a). Assume `A(i)` and `B(i)` are allocated to the same node, and the arrays are cyclically distributed. Blocking up all the communication before the loop results in code for each node as shown in Figure 2.11(b). Leaving the communication in the loop results in code for each node as depicted in Figure 2.11(c).

The blocked version reads or writes $2n$ words from the network and $4n$ words from memory. The streaming version (with communication in the loop) also accesses $2n$ words from the network but only $2n$ words from memory. Figure 2.12 shows total execution time for blocked and unblocked versions of SOR and LU decomposition. These figures show that there is some benefit to leaving communication in the loop for programs like SOR. Since LU decomposition spends proportionately more time in computation than communication, the streaming versus blocking choice does not effect its performance as much. Vectorizing or blocking communication is still beneficial for many cases where the program eventually stores the communicated data to memory.

iWarp also supports a deposit message passing implementation[SSO+95]. The iWarp communication agent has eight DMA-like communication engines that are capable of moving data between memory and the network. These DMA engines can communicate contiguous

```
do i = 1,N
   B(i) = f(A(i+1))
```
(a)

```
send_block(left, A, n)
recv_block(right, ashift, n)
do i = 1,n
   B(i) = f(ashift(i))
```
(b)

```
do i = 1,n
   send(left, A(i))
   B(i) = f(receive(right))
```
(c)

Figure 2.11: Comparison of blocked (vectorized) and unblocked communication. Original sequential code in (a). Blocked communication code in (b). Unblocked communication code in (c).

blocks of memory or do simple strided accesses. The message passing system uses two of these DMA engines: one for receiving messages and another for sending messages. The iWarp deposit library uses oblivious, e-cube routing. This library can use a torus router to utilize all the links of the network or a surface router to lower the resource requirements.

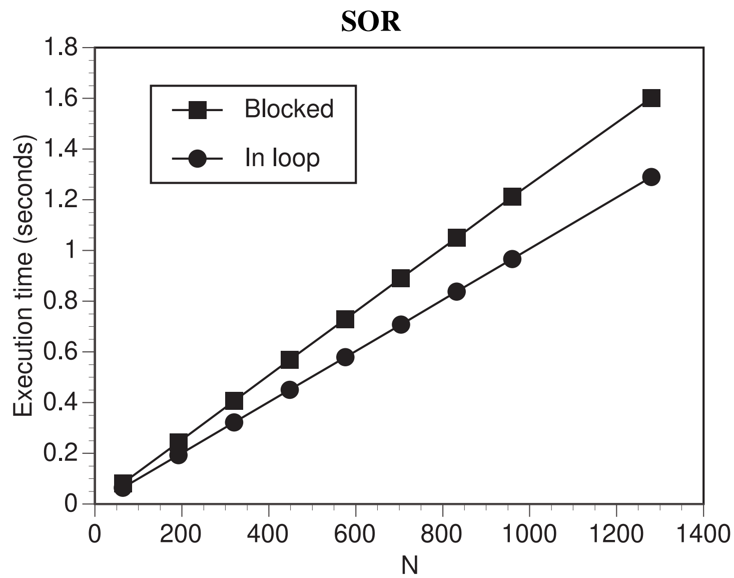
The deposit call starts a sending DMA operation. When a deposit message arrives, an interrupt handler is called that sets up the corresponding receive DMA operation. The destination node expects a certain number of messages in each communication step, so the deposit interrupt handler also increments a predetermined message counter.

The static resource reservation communication model yields performance benefits as a results of several factors. The most obvious performance improvement is a result of improved resource reservation. Figure 2.13(a) shows the benefits of resource reservation by comparing the bandwidth of deposit message passing and connections between two nodes on iWarp. The line labeled deposit measures the cost of the deposit transfer and the barrier synchronization on the 64 node system that is required to ensure data is ready to be received.

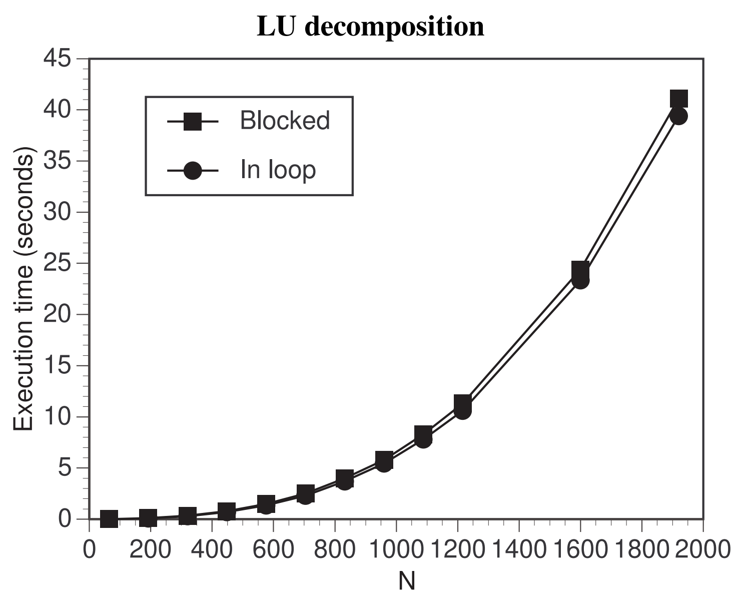
With static resource reservation, the connection-based model can also take advantage of program-controlled routing to better schedule use of the interconnection network. The routing control is particularly important when executing dense communication patterns such as all-to-all communication. With routing control and the static resource reservation, different all-to-all communication algorithms can be used for superior performance. The all-to-all message passing implementation on a 64 node system reaches 500 MB/s aggregate bandwidth, while the fastest connection-based implementation reaches 2.5 GB/s[HKO+94] (as shown in Figure 2.13(b) phased and 2 stage are two connection-based alternative all-to-all implementations).

Models on Paragon

The Paragon is another distributed memory system developed by Intel. Table 2.2(b) summarizes the performance characteristics of three communication libraries on the Paragon using the

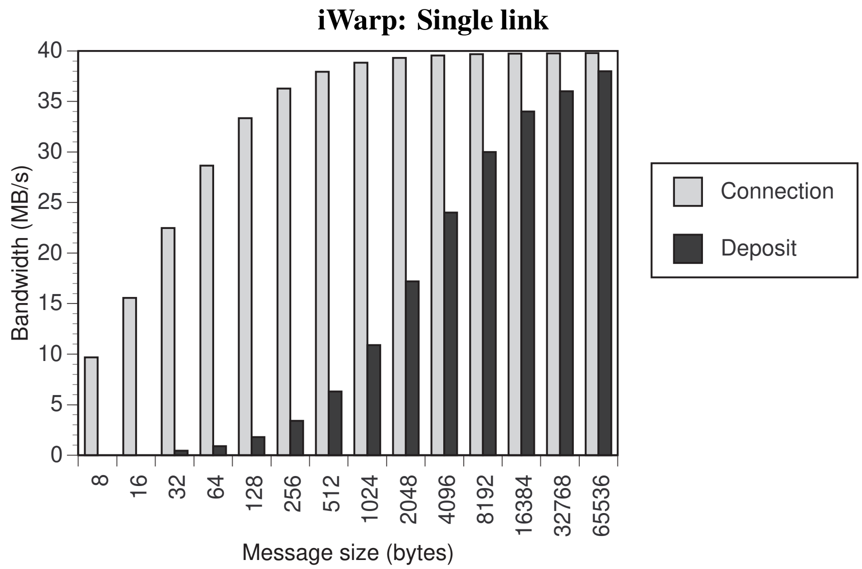


(a)

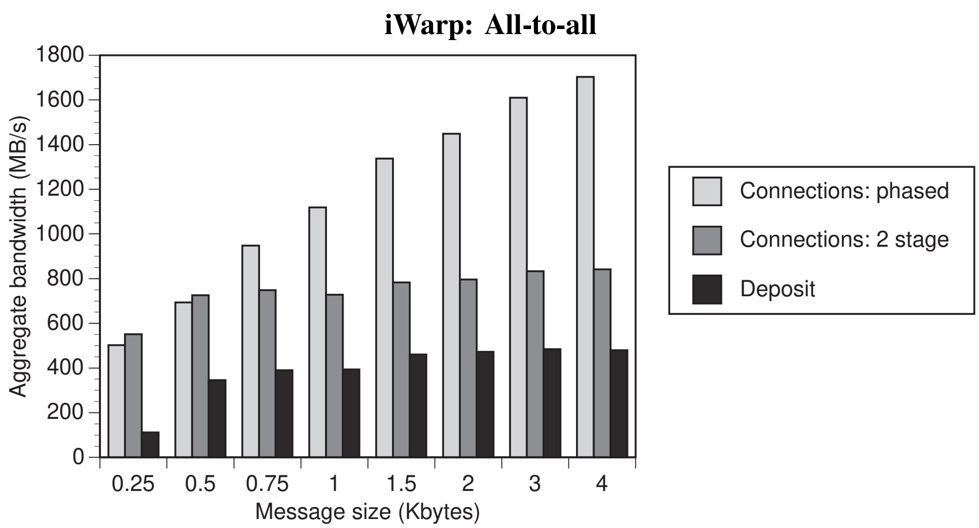


(b)

Figure 2.12: Total execution time comparing blocked communication with communication left in the loop (i.e. streamed) for SOR (a) and LU decomposition (b). The programs operate on cyclically distributed data.



(a)



(b)

Figure 2.13: Average communication performance on iWarp for statically and dynamically reserved implementations for (a) single link bandwidth and (b) all-to-all aggregate bandwidth on 64 nodes.

SUNMOS runtime system[MMRW94]. Under SUNMOS, all three communication libraries use the same basic communication transfer functions. The differences between the libraries are due to differences in the system calls and the message handler routines.

The nodes of a Paragon system are connected in a two-dimensional mesh by 200 MB/s busses. Each node of a Paragon system consists of two i860 processors and a network interface chip (NIC)[3]. One i860 is designated for communication protocol calculations. The routing hardware implements a packet-based communication protocol and oblivious e-cube (row then column) routing.

Software cannot directly control resource reservation on the Paragon; a dynamic strategy for hardware resource reservation is implemented in the system. However, connection-based communication protocols can be used to implement a static resource reservation strategy at the software level. Intel implemented such a software-based connection library in the Virtual Channel Facility (VCF)[MPS93]. This implementation has no routing control, and at the hardware level it must reserve communication resources for each packet of data exchanged. However, data is associated with the connection once it is opened. This data includes the source and destination nodes, the number of requested bytes outstanding, and a queue of outstanding send requests. While hardware resources are not statically associated with the connection, buffers and connection information are stored with the connection. By storing information with the connection, we simplify the logic for performing data transfers.

The connection library guarantees that data need never be buffered in system buffers by following a protocol similar to a simplified version of the fetch model described in Section 2.3.1. In response to a receive operation, the destination node sends a control message of the form $<$ *connection ID*, *number of bytes* $>$ to the source node requesting more data. The address calculation in this connection model is simpler than the general fetch model. The receiving node only sends flow control requests instead of general sets of addresses, and the data in a send operation is queued until a request is received on that connection.

Figure 2.14 shows two examples of data exchange in the software connection protocol. In this protocol, the sending side of the connection stores all state about communication over the connection. In Figure 2.14(a) the send function is executed first. The send call checks if any data requests are outstanding on the connection by checking req_len. There are no outstanding requests, so the send call queues the send request. Later the corresponding receive request is made. The receive call sends a request control message. The request message handler uses the top of the send queue to return the requested information. Figure 2.14(b) shows the corresponding case where the receive request precedes the send. In this case the length of the receive request is stored in req_len. When the send request is made, the send call sends the data immediately because req_len is greater than zero.

The NX message passing library is an example of the general message passing model on Paragon. The SUNMOS runtime system also supports a deposit message passing model. The receiving message handler for deposit messages reads the target address from the message header and reads the remainder of the message from the network directly to the target address. The message handler also updates a predetermined semaphore, so the destination node knows that the buffer has been updated.

[3]The system I measured uses A-step NICs.

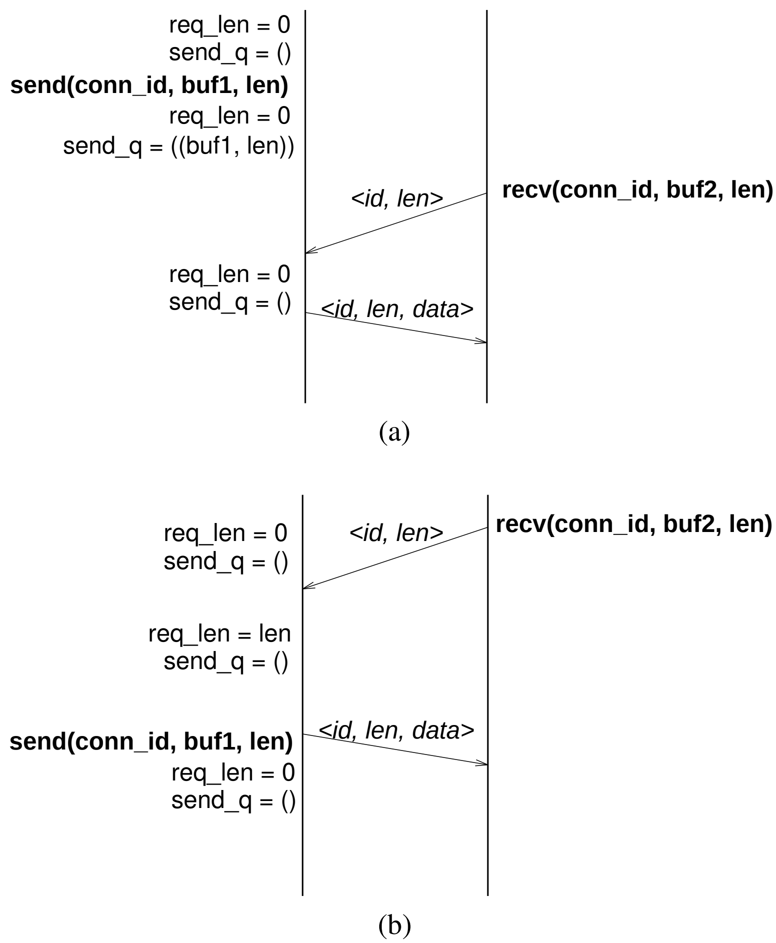


Figure 2.14: Two exchanges using a software connection-based protocol. In (a), the send precedes the receive. In (b), the receive precedes the send.

Figure 2.15(a) compares the single-link bandwidth for different message sizes using the connection library and deposit message passing under SUNMOS. The deposit message passing time includes time for a system-wide barrier synchronization (on a system of 60 nodes). The performance of NX is similar to the connection performance when the receive is posted in advance of the send. The NX performance drops to 20 MB/s if the receiving side must buffer the message[4].

Figure 2.15(b) compares the all-to-all aggregate bandwidth under SUNMOS for 60 nodes using buffered NX, connections, and deposit message passing. Performance of the dense all-to-all communication pattern will vary due to changes in network congestion, but the performance variance greatly increases for messages of 8192 bytes or larger. I am not sure what causes this performance change, but I speculate that it is related to the line transfer unit (LTU or DMA controller). The LTU cannot transfer data that crosses 16K byte boundaries, so to send large messages the system must divide the transfer into several smaller exchanges. By increasing the number of bursts of data entering the network, the variability of network congestion may be increased.

For transfer units less than 8192 bytes (which includes most instances of all-to-all communication we have observed), the deposit message passing performance is superior. The overhead of the barrier synchronization is amortized over more messages, and the connection implementation sends an extra 60×59 control messages which increases network congestion.

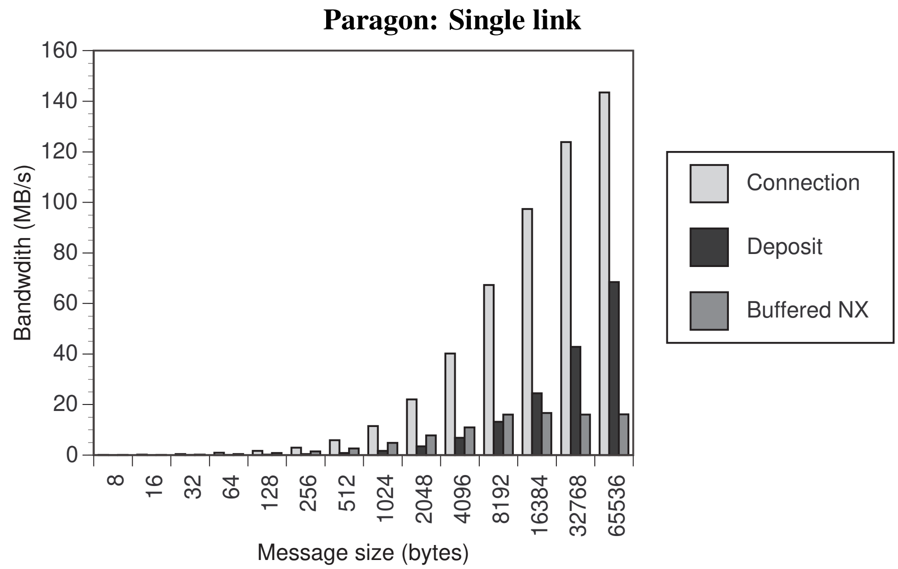
I also measured all-to-all communication performance under both models using a series of congestion-free *phases* as proposed in [Sco91]. Except for cases of extremely large messages, the performance of the phased algorithm was substantially inferior to the naive scheduling on the 60 node machine. There are several reasons for this discouraging performance including synchronization overhead, startup and packetization overheads, software deficiencies that prohibit simultaneous full speed sourcing and sinking of messages, and hardware FIFO bugs. The benefits of bandwidth scheduling are more important for larger machines. Since current Paragon systems are limited to 16 rows, the theoretical peak aggregate bandwidth stops growing for more than 256 nodes. As the size of the machine increases beyond 256 nodes, the bisection bandwidth becomes the limiting resource.

2.3.4 Thesis communication targets

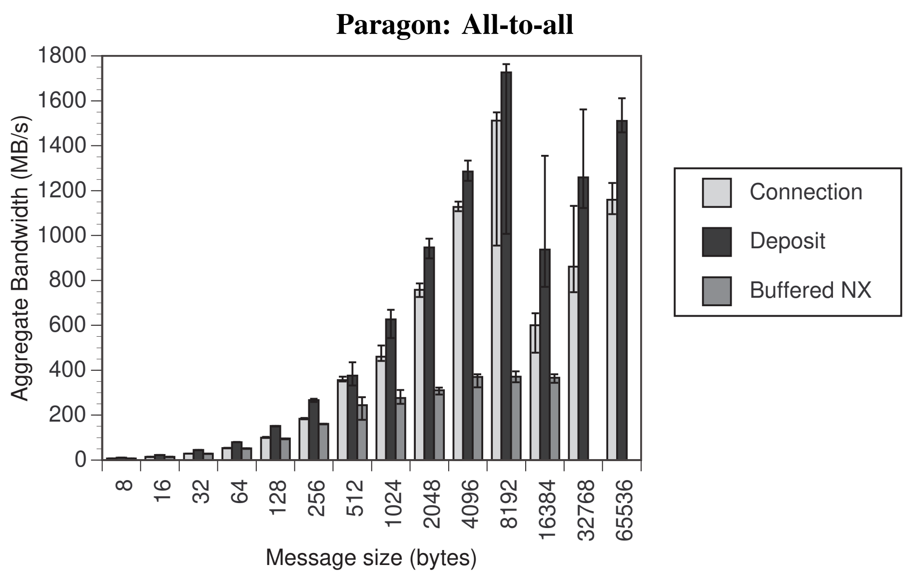
This thesis explores how differences in communication resource reservation strategies affect communication and total execution time. Therefore, the communication selection and optimization phases described in this thesis concentrate on the most efficient instances of communication models with static and dynamic resource reservation on the target machines.

For dynamic resource reservation, the deposit model has been shown to be an effective compiler target[SOG94] and more efficient than the general message passing model[SSO+95]. Therefore, I selected the deposit message passing model as the example of the dynamic resource reservation for this thesis. Since the deposit and fetch models are duals of each other and both avoid message buffering overheads, the choice of the deposit model over the fetch model is somewhat arbitrary. The fetch model requires two-way communication between the source and destination nodes (separating address and data), while the deposit model only requires

[4]Buffered performance reaches 40 MB/s under OSF1 due to a superior memory copy routine.



(a)



(b)

Figure 2.15: Communication bandwidth on Paragon for message passing and connections: (a) bandwidth on a single link and (b) aggregate bandwidth for all-to-all communication over 60 nodes. Graph (b) is also labeled with error bars that show the range between the minimum and maximum bandwidths measured.

communication from the source to the destination (combining address and data). However, for some problems the consumer-directed fetch communication may be more appropriate.

For static resource reservation, this thesis concentrates on a connection-based model with a send/receive interface. The deposit or fetch model can also be implemented over a connection-based communication model. These models avoid buffering messages with dynamic resource reservation, but buffering is unnecessary in the static connection model. Therefore, I use the simpler send/receive interface in this thesis.

2.4 Chapter summary

This chapter defines the models and architectures used by the thesis. The communication optimizations described in this thesis take place in the data parallel compiler model and are targeted to the class of distributed memory machines.

The compiler takes advantage of architecture-specific information by generating communication code for the most appropriate communication model on the target architecture. In particular, this thesis examines communication that relies on static and dynamic communication resource reservation.

Measurements of both options on iWarp and Paragon show that there are substantial performance differences between the two resource management models. Since hardware communication resources can be statically reserved on iWarp, the static connection-based communication is more efficient and schedulable, but this additional power comes at a cost of increased complexity for the programmer. In addition, there may not be sufficient hardware resources to create connections for all communication patterns in a program.

Even if hardware resources cannot be reserved, a connection-based library can statically assign buffers and data to connections. By reserving resources at a software level, the connection-based library on Paragon shows superior performance for sparse communication patterns. For dense communication patterns, the benefits of synchronization for controlling network congestion and the costs of the additional control messages for the software connections make the deposit model superior.

Chapter 3

Communication analysis

This chapter begins the discussion of the communication optimization phase of the compiler. Figure 1.2 shows the major phases and flow of information in the Fx compiler. The compiler uses the Omega test[Pug91] to find the data dependences in the program. The parallelization phase uses the parallel constructs of the input language (such as array statements and parallel loops) and the dependence information to determine which statements can be parallelized.

This chapter describes the analysis phases that discover the required communication patterns (shown as data placement and communication maps in Figure 1.2). In the data parallel compiler model, the programmer works with logical arrays that reside in a single name space. During compilation, these logical arrays must be mapped to physical arrays; potentially subsections of the logical arrays are assigned to arrays on different physical processors, i.e. the physical arrays can be "distributed". The goal of the communication analysis phase is to find a mapping between logical arrays and physical arrays that makes a good trade off between the degree of parallelism and the amount of communication.

The base Fx compiler relies on the HPF `align` and `distribute` user directives to determine data placement. The compiler can derive data placement information from these user directives, but these directives specify a constant mapping for all instances of the same array in the program. To explore more dynamic data mappings, I implemented algorithms that automatically derive the data placement.

To ease the manipulation of communication patterns in later optimization phases, I developed a linear communication map notation. These communication maps are are generated based on data placement information from either either explicit user directives or data placement algorithms.

This chapter finishes with several details of communication generation: replication and physical array assignment. In some cases, more parallelism can be revealed by replacing loop-carried communication with replication. If the physical mapping of an array changes during a program, several different physical arrays may be required to store the distributed array.

3.1 Data placement

In the data placement phase, the compiler uses analysis or user directives to calculate the mapping from data array elements to processing nodes. To simplify the problem, data placement is often divided into two sub-problems.

Data alignment: A relative mapping between elements of different arrays that define computation constraints.

Data distribution: A mapping from data array dimensions to the target physical topology.

By looking at data alignment first, the data distribution problem has more structure, simplifying the search space of reasonable data distributions.

Figure 3.1(a) shows an example of how arrays `A` and `B` can be aligned with respect to each other. In this case, elements `A(i)` and `B(i+1)` are assigned to the same virtual processor. Figure 3.1(b) shows array `B` distributed over a processor array with P nodes. Each node stores three elements of `B` locally, i.e. elements `B(1:3)` are assigned to the same physical processor.

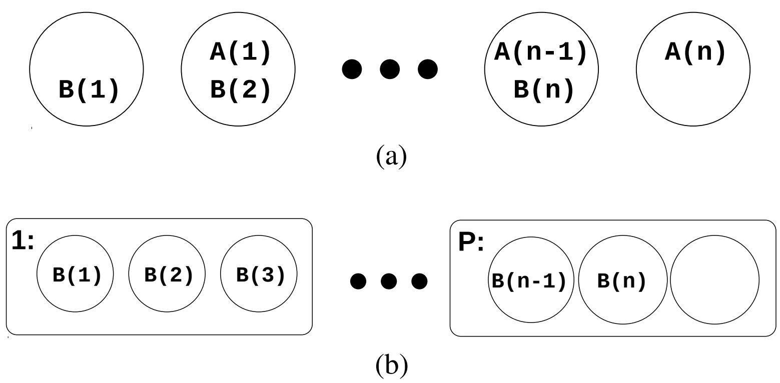


Figure 3.1: Examples of data alignment (a) and data distribution (b).

One approach to discovering the desired data array alignment and distribution is to add user directives to the programming language. This approach is used by Fortran D[FHK+90], Vienna Fortran [CMZ92], and High Performance Fortran (HPF) [For93] among others. This approach relieves the programmer of the task of directly inserting send and receive statements, but the programmer must have a detailed understanding of the memory layout of the program, so the programmer is still working at a low level of abstraction. The best placement decisions will vary between different architectures. With explicit user placement, the programmer must reconsider the data placement for each new architecture.

In [GS91], Gilbert and Schreiber show how to calculate the optimal placement and movement of arrays in an expression given a simple model of communication, but it is not clear that an optimal solution can be found for a more realistic communication model. In [LC90], Li and Chen prove that in the general case it is NP-hard to find an optimal alignment that minimizes communication over the complete program, so a realistic data alignment algorithm must make some heuristic decisions.

Amarasinghe and Lam[AL93] developed an algorithm that finds the tradeoff between parallelism and communication by calculating projections of the iteration space onto the processor space. The programs are Fortran 77 loop nests with loop bounds and references that are affine functions of the enclosing loops indices.

An algorithm has been designed for the Connection Machine Fortran compiler that attempts to minimize and identify alignment communication in the data parallel Fortran program[KLS90]. Similar algorithms have been developed by [LC91b], [GB92b], and [CGST93].

Regardless of the alignment algorithm used, four types of alignment conflicts are revealed by data alignment. For these examples, assume elements `A(i,j)` and `D(i,j)` are aligned.

- **offset conflict:** The lower and upper bounds of the slice indices are off by a constant, but the strides are the same, e.g. `A(1,1:n)` and `D(1,1+c:n+c)`. Data must be shifted along the matching dimension.

- **stride conflict:** The strides of the slices differ, e.g. `A(1,1:n:1)` and `D(1,1:2*n:2)`. Depending on the strides, the data must be redistributed along that dimension. This redistribution often requires a dense communication step.

- **dimension or axis conflict:** The corresponding slices are not distributed along the same dimensions, e.g. `A(1:n,i)` and `D(i,1:n)`. Data must be redistributed between two different dimensions. This redistribution step frequently requires an all-to-all or dense communication step.

- **scalar conflict:** Scalar dimensions of the array occurrences are not aligned, e.g. `A(1,1:n)` and `D(2,1:n)`. This is really another form of offset conflict, but the scalar reference does not have the information that is immediately available with an array slice reference. Resolving such conflicts is more difficult, because it is not immediately clear how to match the scalar array dimensions. For example, in the case below it is not obvious whether to align dimension 1 of `A` with dimension 1 or 2 of `D`.

  ```
  ... = A(1, 1:n) + D(2, 1, 1:n)
  ```

  The CM Fortran algorithm locally searches for the best match while constructing the data alignment graph. It is possible to delay decisions and use the program structure to make a better match of related scalar dimensions. By delaying the assignment of scalar dimensions until the neighboring statements have been analyzed, the compiler can make a more informed choice.

In all cases, these conflicts require regular communication patterns. There may be cases of two array occurrences that have a combination of these alignment conflicts. For instance, `A(1:n)` and `D(21:2*n+21:2)` have an offset and a stride conflict.

The alignment conflicts show where communication may occur in the program. Whether communication is actually required depends on the data distribution. In his thesis[Who91], Wholey addresses the problem of automatically finding a good data distribution. He developed a compiler for a simple data parallel language. His compiler uses a subset of the CM Fortran data alignment algorithm to create a data alignment graph, and the compiler uses a search algorithm to find a low cost distribution for a set of related arrays.

The Fx compiler supports two methods for deriving data placement information: by the `align` and `distribute` user directives and by a data alignment algorithm based on the one used by CM Fortran data alignment algorithm and a distribution algorithm based on Wholey's. Since Fx contains more constructs than Wholey's input language, the Fx compiler cannot directly use alignment and distribution algorithms described by Wholey.

The next sections describe the automatic data placement algorithms. By using either the user directives or the placement algorithms, the compiler can construct a data alignment graph, and it can derive the communication map information described in Section 3.2.

3.1.1 Automatic data alignment

The CM Fortran algorithm first looks at the program text and loop index patterns to create a *data alignment graph*, where the nodes represent array dimensions and the arcs represent alignment constraints. Figure 3.2 shows an example data alignment graph and the corresponding source code. After creating the data alignment graph, the algorithm tries to group array dimensions based on their alignment constraints. Array dimensions that are grouped together are stored on the same processors. Related array dimensions that cannot be grouped together have conflicting alignment requests, so these dimensions require runtime communication to satisfy the alignment requirements. After the data alignment phase, the arcs in the contracted data alignment graph represent all communication that may occur at runtime.

Alignment graph creation

The data alignment algorithm starts by looking at *array occurrences* in the program. An array occurrence is a textual instance of an array in the program. The data alignment algorithm works on the array dimensions separately, so in addition to array occurrences, the compiler works with *bundles*. A bundle is a handle to a set of aligned array occurrence dimensions. Initially, no array dimensions are aligned, so a unique bundle is assigned to each dimension of each array occurrence. Figure 3.2(b) shows the initial data alignment graph for the code segment in Figure 3.2(a).

Alignment constraints between bundles define constraints on how logical arrays should be relatively mapped to avoid communication between the two array occurrences. The initial data alignment graph is constructed with bundles for the nodes and alignment constraints between the bundles shown as arcs.

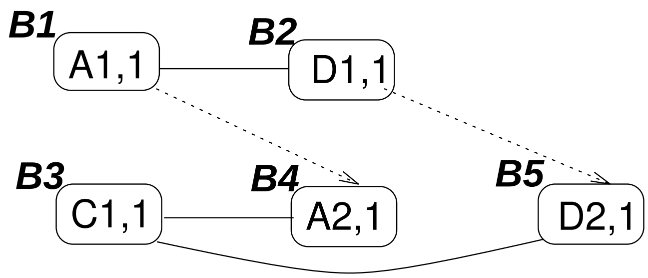
There are two major types of alignment constraints. *Identity* constraints relate different array occurrences of the same logical array in separate statements. Identity arcs are inserted between corresponding bundles from one array use to each next possible array use. The dashed arcs in Figure 3.2(b) show the identity constraints in that example.

Conformance constraints define how array occurrences should be mapped to avoid communication within a single statement. Conformance arcs are inserted from the bundles of left hand side array occurrence to the right hand side array occurrences of the corresponding slice dimension. Adding these arcs results in a graph of bundles and alignment arcs that represents the alignment requirements of the program.

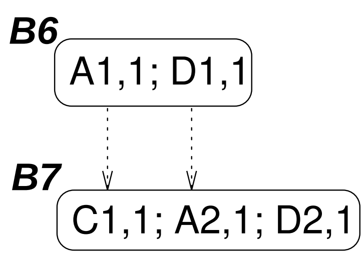
In a section of straight-line code where all array ranges are known, all alignment and conflict

```
A(2:n) = D(2:n) + 1.0
C(1:n) = A(1:n) + D(1:n)
```

(a)



(b)



(c)

Figure 3.2: Example code (a). Initial data alignment graph (b). Partially contracted data alignment graph (c). The solid arcs are conformance constraints. The dotted arcs are identity constraints. The nodes are bundles labeled in the upper left corner. The interior of the bundle lists the array occurrences contained in that bundle. A2,1 *refers to the first dimension of the second occurrence of* `A`.

information can be determined at compile-time. If the array slice ranges are not constants, array indices can be symbolically analyzed to get better information. With code that contains loops and conditionals, locating identity arcs becomes more complicated. Identity arcs may cross basic block boundaries, and a particular array use may have more than one potential last use in different basic blocks. Information from the data dependence phase is used to find all potential identity arcs.

Alignment graph contraction

The initial data alignment graph contains all alignment constraints. Different mappings of logical to physical arrays will satisfy different subsets of these alignment constraints. If all alignment constraints can be satisfied, the resulting parallel program does not require any communication. The data alignment algorithm attempts to find a logical to physical mapping that satisfies the "best" subset of the alignment constraints.

Each bundle of each array occurrence is assigned an initial data-to-node map from data elements to virtual processors. The structure of the data-to-node maps is discussed in greater detail in Section 3.2. For this discussion, it is sufficient to think of the data-to-node map as a linear mapping from each array dimension to a virtual processor dimension.

For array slices, the data-to-node mapping is calculated from the slice parameters. The lower bound is the initial offset, and the stride is the initial multiplier, so the initial mapping for occurrence `A(lb:ub:s)` is $map_A(i) = s \cdot i + lb$. For array dimensions with only scalar references, the initial map is simpler. The scalar is the initial offset and the multiplier is assumed to be one, e.g. `A(x)` is $map_A(i) = i + x$. If the scalar is a loop index, the compiler can derive more information about the initial map from the loop bounds and strides.

By contracting arcs and merging the corresponding bundles, the algorithm finds a subset of the alignment constraints that can be satisfied. Two bundles can be merged if one of the following conditions holds:

1. Bundles do not contain occurrences of the same arrays.

2. For occurrences that appear in both bundles, offsets, strides, and dimensions match.

3. For occurrences that appear in both bundles separated by an identity arc, offsets and strides can be made to match by performing the same adjustments to all entries in one of the bundles.

For the initial graph in Figure 3.2(b) the conformance arcs can be merged since none of the conformance arcs connect bundles that contain occurrences of the same array. After the conformance arcs are merged two identity arcs remain as shown in Figure 3.2(c). From the lower bound of the first occurrence of array `A`, the compiler calculates the initial offset for A1,1 as 2, but the initial offset for A2,1 is 1. By adjusting the offsets in one of the bundles, we can maintain the same mapping for both occurrences of `A`. If we add 1 to all offsets in bundle *B7*, offsets for both `A` and `D` match, so both identity arcs can be contracted. Therefore, this example requires no runtime communication.

The code in Figure 3.3 is a slight variation of the code in Figure 3.2(a). Both examples result in the same initial and partially contracted data alignment graphs (Figures 3.2(b) and (c)), but

```
A(2:n)   = D(2:n) + 1.0
C(1:n-1) = A(1:n-1) + D(2:n)
```

Figure 3.3: Example code that contains an offset conflict.

the identity arcs in the new example cannot be contracted. Since the lower bounds of the slices are different in this example, the initial map offsets are different. By adding 1 to all offsets in bundle *B7*, offsets for `A` match, so the `A` identity arc can be contracted, but the offsets for `D` do not match, so the `D` identity arc cannot be contracted. If the arrays are distributed, this code will require communication between the statements to satisfy the offset conflict.

Figure 3.4(a) shows another example code segment with the initial data graph in Figure 3.4(b). Since no occurrences of the same array are connected by conformance arcs, the conformance arcs are contracted to form Figure 3.4(c). The identity arcs cannot be contracted, because the merged bundle would contain references to multiple dimensions of the same array. For example, merging bundles *B11* and *B13* would result in a bundle that refers to dimensions 1 and 2 of array `A`. Therefore, if any array dimension is distributed in this example, communication will be required to satisfy the identity constraints.

The arrays that correspond to the merged bundles will not need to communicate along that dimension at runtime (e.g. A1,1 and D1,1 in Figure 3.4(c)), because corresponding elements can be stored on the same nodes along that dimension. The arcs that cannot be contracted represent communication that must occur to maintain the desired alignment if the dimensions in the bundles are distributed. For example in Figure 3.4, communication will be required between the first and the second statement if the first dimension of array `A` is distributed, because A1,1 and A2,1 occur in separate bundles separated by an identity arc.

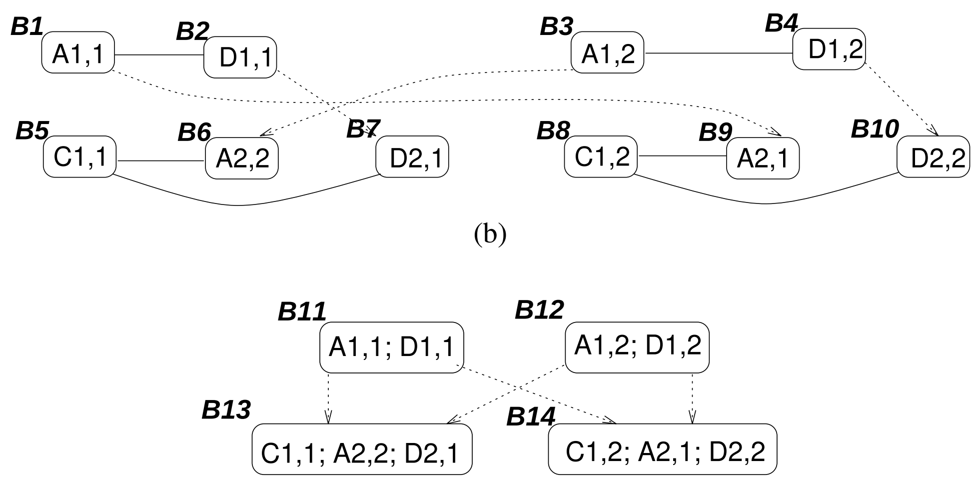
The order in which arcs are contracted in the alignment algorithm affects which communication constraints can be satisfied, so I augmented the CM Fortran algorithm with heuristics directing arc contraction order. The key insight is that the earlier an arc is encountered, the more likely it can be successfully contracted (and the resulting alignment constraint satisfied). Using this insight, it makes sense to contract more "expensive" arcs first. This assumption is used in [Who91] for a heuristic that first contracts arcs in the most deeply nested loops, because more deeply nested code will probably be executed more often. In our implementation, each arc is weighted by the *expected* amount of communication, so the algorithm attempts to contract more heavily weighted arcs first. The expected amount of communication between a pair of bundles is calculated from the compile-time information about the array sizes and control flow arc iteration counts.

Different types of conflicts can result in communication steps of different costs; some conflicts require more expensive communication than others. For example, offset conflicts that are not satisfied may require nearest neighbor communication, but dimension conflicts that are not satisfied may result in more expensive all-to-all communication. We attempt to contract arcs from most expensive type to least expensive type: dimension conflict, stride conflict, offset conflict. This type ordered heuristic is also proposed in [CGST93] and [GB92b].

Data alignment also presents a problem in aligning part of an array. In the case shown in

```
A(1,1:n) = 3.0 * D(1,1:n)
C(1:n,1) = A(1,1:n) - D(1:n, 1)
```

(a)



(c)

Figure 3.4: Example code (a). Initial data alignment graph (b). Partially contracted data alignment graph (c). The graph in (c) cannot be further contracted due to dimension conflicts.

```
do j = 1,n
   f(A(i,1:n), D(j,1:n))
enddo
```

Figure 3.5: Aligning part of `A`

Figure 3.5, the ith row of `A` should be aligned with each row of `D`. This has been called *mobile alignment*, and [CGS93] and [KLD92] have incorporated the ideas of mobile alignment into their alignment algorithms. To maintain the alignment constraints, the ith row of A must be realigned in each iteration. Strictly speaking all rows of A do not need to be moved only the ith row. Changing the owner of the rows or columns of an array separately may be useful, but it adds complexity to tracking array locations at runtime. It is not clear at what granularity it is worthwhile to track array subsection movement. Another alternative is to not change the owner of row i of `A`, but only temporarily shift the data in row i to the appropriate processors.

For simplicity of the implementation, the current prototype shifts the entire array with each subsection movement. For the example of aligning the ith row of A, every row of A will be adjusted to ensure that row i is co-located with row j of D.

Limitations and comparisons

In addition to the CM Fortran data alignment algorithm, there are two other main data alignment algorithms: one developed by Li and Chen for the Crystal compiler[LC91b] and another developed by Chatterjee et al. for HPF[CGST93].

Li and Chen's algorithm was developed for Crystal, a functional language. This algorithm concentrates on avoiding dimension conflicts (maintaining dimension uniform accesses). Since it was developed for a functional language, it assumes that each array is written at most once. Therefore, array references in sequential loops have a time dimension added. These additional requirements are necessary for the analysis but can presumably be removed for execution.

Li and Chen's algorithm does not address multiple separate array instances. All instances of each array dimension are represented as a single node in the component affinity graph, so all occurrences of an array must have the same distribution. The nodes are connected by weighted affinity arcs, depending on how the corresponding array dimensions are related in the program. The arc weights can be 1, ∞, or ϵ. The Crystal algorithm selects nodes that use the same iteration index element and uses an optimal bipartite graph matching algorithm to satisfy the affinity arcs to those nodes.

The Fx data alignment algorithm and the Crystal data alignment algorithms differ primarily on the following points:

- The Fx algorithm uses information about the program's control flow.
- The Fx algorithm separately addresses different occurrences of the same array instance, allowing the same array to be mapped differently at different points in the program.
- The Crystal algorithm only addresses dimension conflicts.
- The Crystal algorithm uses an optimal algorithm to eliminate conflicts between array dimensions; the Fx alignment contraction algorithm is not provably optimal.

The HPF data alignment algorithm developed by Chatterjee et al. is newer and makes more sophisticated decisions that avoid the limits of the owner computes rule. This algorithm also addresses different type of alignment conflicts separately, and the Fx data alignment algorithm augments the CM Fortran data alignment algorithm with this idea of conflict type differentiation.

The HPF algorithm creates an *alignment data graph* for each basic block. Instead of using identity arcs that show control flow, the HPF algorithm uses a more explicit data flow representation over the entire basic block. With this representation, the original statement limits are no longer observed. This algorithm uses information about the amount of data to be moved to determine where data should be calculated. This algorithm is applied to each basic block, and the results from multiple basic blocks are tied together.

The Fx data alignment algorithm and the HPF algorithm differ primarily on the following point:

- The HPF algorithm is not restricted by the owner computes rule. However, the owner computes rule has not been a limitation for our set of sample Fx programs.

3.1.2 Automatic data distribution

By contracting the data alignment graph, the compiler has calculated a mapping from data elements to a virtual template space. The distribution step finds a mapping from the template to the physical processors; in this step, the compiler determines whether data should be distributed over multiple nodes and which distribution pattern to use: cyclic, block, or block-cyclic. For simplicity of implementation, the prototype compiler avoids the data distribution pattern problem and assumes that all distributed dimensions use a block distribution. The communication optimizations in the later phases apply equally well to arrays distributed in cyclic or block-cyclic patterns.

Dimension distribution

The arcs that remain in the data alignment graph after the data alignment algorithm finishes show where communication "may" occur. If the dimensions corresponding to the endpoint nodes are not distributed, the arc does not require any runtime communication. Automatic distribution algorithms attempt to minimize the number of arcs that require communication while maintaining the benefits of parallelism.

In his thesis[Who91], Wholey developed a search-based distribution algorithm for a simple data parallel language. The Fx compiler uses a distribution algorithm based on Wholey's, but Fx contains more constructs than Wholey's input language, so the contracted data alignment graphs tend to be larger. Therefore, the Fx compiler must reduce the search space to make the distribution search practical.

Wholey's algorithm searches the space of potential application distributions to find the "best" distribution. A potential distribution is created by assigning a distribution state to all the array dimensions in each node. The distribution state indicates which physical dimension the array dimension is distributed over. For a machine with a D-dimensional topology, the set of distribution states is $\{i|0 \leq i \leq D\}$, where 0 means the dimension is not distributed and 1 through D indicate that the dimension is distributed along that physical axis. For each potential distribution, the algorithm uses a cost model of the machine to estimate the cost of communication and computation.

Our implementation reduces the search space in three ways. First, not all arrays in a program are related by alignment arcs; there may be several connected components. The compiler divides the array dimensions into sets that are related and distributes each connected component separately.

Second, the cost of the communication that results from different types of alignment arcs differ. Some arcs require relatively cheap shift communication; while others require all-to-all communication. The Fx compiler reduces the number of nodes to search by contracting the arcs that correspond to the "cheaper" potential communication. By contracting these arcs, we make the assumption that these communications have no cost, thus concentrating the search on avoiding the more expensive communication patterns.

Of course, the relative costs of different types of communication depend on the target machine. All-to-all communication compared to shift communication will be relatively cheaper over a cross-bar network than on a mesh network. Still for most realistic machines, exchanging P messages per node in an all-to-all communication will be more expensive than exchanging one or two messages per node in a shift communication.

Finally, larger programs are generally composed of several different subalgorithms. The array distributions for each subalgorithm may not be logically related. However, data structures may be reused, so this division may be difficult for the compiler to discern. Dependence information can be used to find where old values are killed to find subalgorithms. The prototype also includes a directive for the programmer to indicate logical subalgorithms that can be separately distributed.

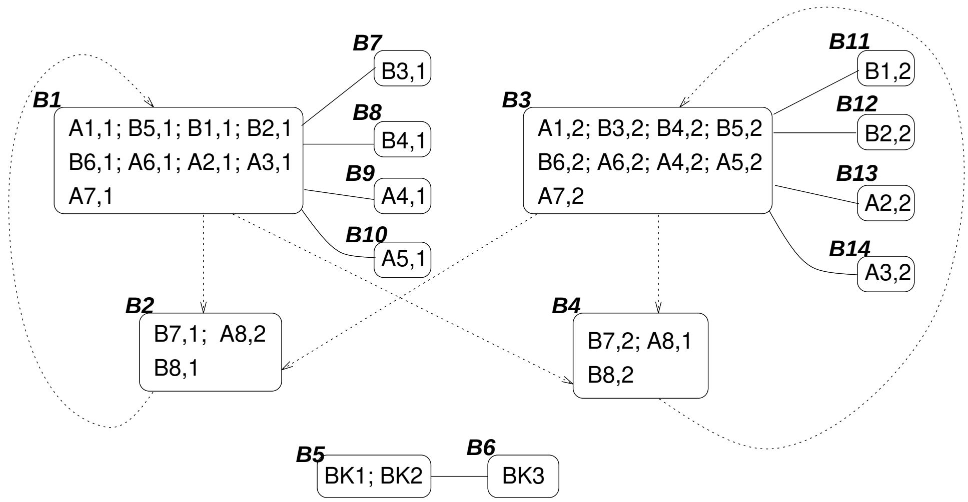
Figure 3.6 shows two versions of a contracted data alignment graph. The original version contains 14 bundles. Bundles *B5* and *B6* are not connected to any other bundles, so the graph contains two connected components that can be distributed separately of sizes 12 and 2. Figure 3.6(b) shows the graph that remains after merging the offset arcs that correspond to less expensive potential communication. This graph has one connected component of size 4 and another of size 2. By reducing the number of bundles from 14 to 4 and 2, the search time is substantially reduced.

3.1.3 Hybrid alignment and distribution analysis

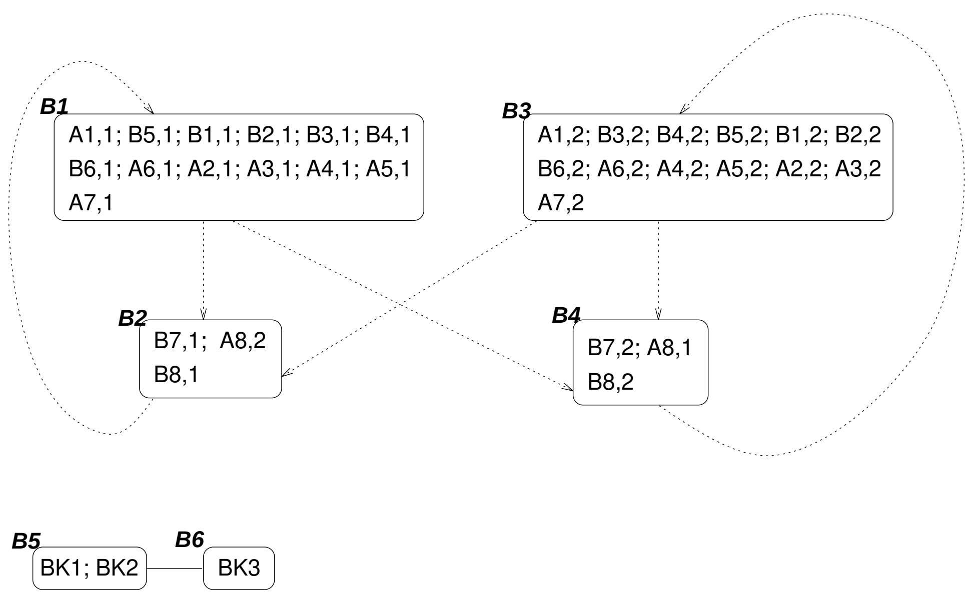
In many parallel programs, it is clear to the programmer how key data structures should be distributed at one or two critical points in the program. If the programmer is allowed to place several array occurrences explicitly, the job of the automatic alignment and distribution algorithm is simplified. The user-specified array occurrences guide the graph contraction in the data alignment algorithm and further reduce the search space of the distribution algorithm.

With automatic data alignment, different occurrences of the same array can be distributed differently. Therefore, the HPF `align` and `distribute` directives are not sufficient for this hybrid data placement, because these directives affect all occurrences of the array in the procedure.

With the HPF `realign` and `redistribute` directives, the programmer can indicate that the placement of a data structure has changed. The current Fx prototype does not support the `realign` and `redistribute` directives. To experiment with hybrid data placement, I implemented the following statement level directive: `localdist(array, array_dim, processor_dim, offset)`. This directive indicates that all occurrences of `array` in the previous statement should have dimension `array_dim` distributed across processor dimension



(a)



(b)

Figure 3.6: Two versions of a data alignment graph constructed from an iterative smoothing and FFT program. Graph (b) is produced by the data alignment algorithm. Graph (c) is further reduced by contracting the offset conflict arcs.

```
do j = 1,n
   if (j .ne.  pcol) then
      a(j,pcol) = 0.0
      a(j,:)  = a(j,:)  - arow1(:)*acol(j)
      localdist(a, 1, 0, 0)
      localdist(a, 0, -1, 0)
   endif
enddo
```

Figure 3.7: Use of localdist directive to force array `A` *to be distributed by rows*

`processor_dim`. If `processor_dim` is a negative number, the corresponding array dimension is not distributed. Figure 3.7 shows the inner loop from a matrix elimination program that uses the `localdist` directive to force the major array `A` to be distributed by rows. The other array occurrences are not specified, and the compiler uses the automatic techniques to derive their alignment and distributions.

My experience with Fx programs shows that combining user directives and automatic placement is a reasonable trade off between giving programmer control of data placement while not overloading the programmer with data placement details. As the cost models used in the distribution search become more accurate, the need for user hints for distribution should not be as necessary. However, it seems likely that there will always be programs where the compiler does not have as much information about the program performance as the programmer does. Therefore, user distribution directives will continue to be useful.

Limitations and comparisons

Gupta addresses the issues of data placement in his thesis[Gup92]. He evaluated these algorithms in the Paradigm Fortran 77 compiler.

Gupta divides data placement into four parts: array dimension alignment, distribution pattern selection (blocked, cyclic, or blocked/cyclic), block size selection, and dimension distribution. For dimension alignment, he uses as similar approach to Li and Chen[LC91b].

For distribution pattern selection, the Paradigm compiler analyzes the loop structure, to determine if load balancing is required. If load balancing is required, it selects a cyclic distribution. The Paradigm compiler creates a graph of stride relations to find the best block size. It finds a minimum spanning tree of this graph and assigns block sizes along this tree.

Finally, for dimension distribution, the compiler uses a search technique that is somewhat more structured than Wholey's. It first analyzes the program and serializes any dimensions that have no potential for parallelism. If more than two dimensions remain, the compiler searches all possibilities of the distribution assignment for the minimal cost distribution.

The Fx and Paradigm distribution techniques differ on the following points:

- The Paradigm algorithm addresses alternative distribution patterns, and the Fx algorithm only looks at simple block distribution.

- The Paradigm algorithm first attempts to eliminate inherently serial dimensions from the distribution search.

3.2 Communication maps

After the alignment and distribution phases or from user placement directives, the compiler has generated a data alignment graph decorated with distribution information. The data placement phase has calculated a data-to-node map for each array occurrence, but the compiler is interested in node-to-node maps that describe communication patterns. The compiler can mechanically derive the node-to-node maps from the data-to-node maps of different occurrences of the same array.

Node-to-node maps offer a simple, concise structure for describing the communication patterns that result from many dense, regular scientific programs. The later phases of the communication generation and optimization (described in Chapters 4, 5, and 6) leverage off the structure of these communication maps. The node-to-node communication maps operate on the domain and range of k-vector processor IDs. There are three types of communication maps: proper functions, constant functions, and wildcard functions.

The proper functions only have finite integer multipliers and are injective. map_1 is an example of a proper function.

$$map_1(\vec{p}) = \begin{pmatrix} p_1 + 3 \\ p_2 \end{pmatrix}$$

A constant function maps many inputs to one output. By allowing constant functions, the communication maps can describe gathering communication. The second index of map_2 is a constant function, where all data on row p_1 is gathered into the first node of that row.

$$map_2(\vec{p}) = \begin{pmatrix} p_1 \\ 1 \end{pmatrix}$$

A wildcard function maps one input to many outputs. Wildcard functions describe scattering communication. The second index of map_3 is an example of a scattering function. Data from (p_1, p_2) is scattered to some subset of the nodes of row p_1.

$$map_3(\vec{p}) = \begin{pmatrix} p_1 \\ \infty \end{pmatrix}$$

This section describes in more detail the structure of the communication map notation that I developed. It describes how the node-to-node communication maps are generated from data placement information and used in later compiler phases. First, maps to virtual nodes are described. Then I describe the adjustments to these maps that are required to define communication on a finite set of physical nodes.

3.2.1 Data-to-node maps

During the data alignment graph contraction, the compiler calculates the mapping from data indices to node IDs[1]. A data-to-node map is expressed as:

$$map(i_1, i_2, ..., i_k) = map(\vec{\imath}) = S \cdot (m \cdot \vec{\imath} + \vec{o})$$

where m is a diagonal matrix of multipliers and $\vec{o}$ is a vector of offsets. Each diagonal entry m_i of m corresponds to the multiplier for the i^{th} data array index. Similarly o_i is the offset for the i^{th} data array index. If the data array is k dimensional, the offset vector is k long and the multiplier matrix is $k \times k$. S is the $k \times k$ subscript matrix that maps the data index to the appropriate node index. If the i^{th} data array index is mapped to the i^{th} node index, S will be the identity matrix. Otherwise, S is a row permutation of a $k \times k$ identity matrix.

The Fx compiler finds most of its parallelism from array level statements, where subsections of arrays are referenced by *slices*, e.g. `A(1:n)` refers to elements 1 to `n` of `A`. By focusing on array slices, each entry of these data-to-node maps is a simple linear function of a single loop index. For example a slice $(lb : ub : s)$ specifies a data array index from a single iteration variable j by a linear function of the slice parameters, $s \times j + lb$.[2] This mapping notation handles array indices that are simple affine functions, but it does not handle array indices that are used in multiple dimensions, e.g. `A(i,i)`.

Consider the code in Figure 3.8(a). Equation 3.1 uses the data-to-node mapping notation to describe the mapping of the first occurrence of array `A`. In this example, a two dimensional data array `A` with indices $\vec{\imath} = (i_1, i_2)$ is mapped onto a two dimensional node array, and the data alignment graph shows that the mapping is $\vec{p} = (i_1, 0)$. Then m_A, S_A, and o_A are:

$$m_A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} S_A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \vec{o_A} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{3.1}$$

These matrices and vectors can be combined to form the node index vector.

$$map_A(\vec{\imath}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \vec{\imath} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} i_1 \\ 0 \end{pmatrix} \tag{3.2}$$

Given two occurrences of the same array that conflict in the alignment graph, the compiler can calculate the initial and final maps for that array in three steps. The compiler performs these calculations while contracting the data alignment graph and searching for a data distribution.

Consider the two occurrences of `A` in Figure 3.8 with an offset conflict of 1.

1. Calculate A's initial mapping map_A.

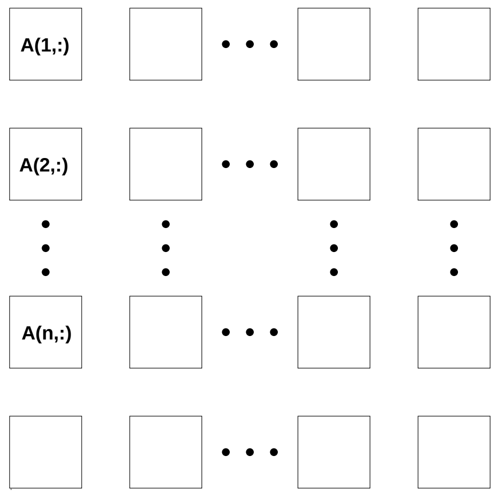
2. Add the conflict information to map_A to get $map_{A'}$

[1] For now consider the nodes to be virtual nodes. Section 3.2.4 addresses the issue of blocking data onto physical nodes

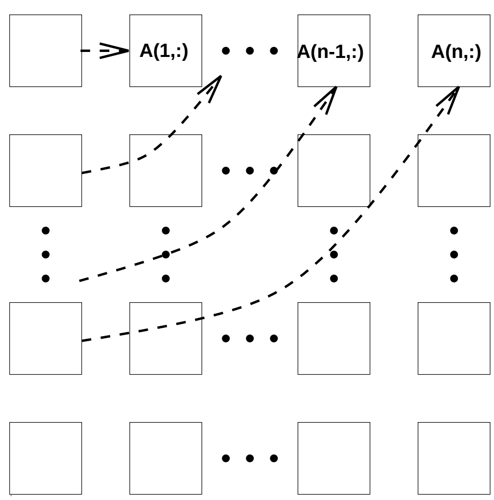
[2] If array indices are functions of multiple loop indices, and these indices are linearly combined and only occur in one dimension of an array occurrence, this limitation may be loosened by keeping a sum of simple maps for each array occurrence, e.g. $map(\vec{\imath}, \vec{\jmath}, ...) = S_1 \cdot (m_1 \cdot \vec{\imath} + \vec{o_1}) + S_2 \cdot (m_2 \cdot \vec{\jmath} + \vec{o_2}) + ...$

```
C(1,1:n) = A(1:n,1)
D(1:n) = C(2:n+1,1) + A(1:n,1)
```

(a)



(b)



(c)

Figure 3.8: Example HPF code in (a). (b) and (c) show the data-to-node mapping for both occurrences of `A`*. The arrows in (c) show the node-to-node communication that is required.*

3. Adjust the subscript matrix S' to S'' to match the distribution of the second occurrence of A.

The arrays are mapped onto a two dimensional node torus or mesh. The first occurrence of `A` is distributed over dimension one. Then the second occurrence of `A` is distributed over dimension two. A's initial mapping is described in Equation 3.2. Figures 3.8(b) and (c) shows the data to node mappings of both instances of `A`. Adding the conflict offset of 1 yields

$$map_{A'}(\vec{\imath}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \vec{\imath} + \begin{pmatrix} 0+1 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} i_1 + 1 \\ 0 \end{pmatrix}$$

To adjust the map to match the new distribution, permute the rows of the S matrix, so array index i_1 is mapped to node index p_2.

$$map_{A''}(\vec{\imath}) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \vec{\imath} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 0 \\ i_1 + 1 \end{pmatrix} \tag{3.3}$$

3.2.2 Node-to-node maps

Given two data-to-node maps, map_1 and map_2, the compiler constructs a node-to-node map that takes a data array index vector and maps the results of map_1 to the results of map_2. This map exposes the node-to-node communication or *communication pattern* needed to change the distribution of a data array from map_1 to map_2. For example, the dashed arrows in Figure 3.8(c) show the node-to-node communication required to move between the two data-to-node mappings of `A`.

With simple algebra, one can calculate α and β, the multiplier matrix and offset vector for the node-to-node map from $\alpha \cdot map_1(\vec{\imath}) + \beta = map_2(\vec{\imath})$.

$$pmap(map_1(\vec{\imath})) = \alpha(S_1 \cdot (m_1 \cdot \vec{\imath} + \vec{o_1})) + \beta = S_2 \cdot (m_2 \cdot \vec{\imath} + \vec{o_2})$$

Solving for the coefficient for $\vec{\imath}$ we get α.

$$\alpha = S_2 \cdot m_2 \cdot m_1^{-1} \cdot S_1^{-1} \tag{3.4}$$

Then we can use α to solve for the offset β.

$$\beta = S_2 \cdot \vec{o_2} - \alpha \cdot S_1 \cdot \vec{o_1}$$

$$\beta = S_2 \cdot (\vec{o_2} - m_2 \cdot m_1^{-1} \cdot \vec{o_1})$$

However, the matrices may be singular, and in fact the multiplier matrix is most likely singular, because the diagonal entries that correspond to non-distributed dimensions are zero. Therefore, we replace each inverted diagonal matrix m^{-1} with an extended inverted diagonal matrix m^+.

Definition 1 *A $k \times k$* **extented inverted diagonal matrix** m^+ *is defined from a $k \times k$ diagonal matrix m. Each diagonal entry of m_i^+ is defined as the inverse of m_i, the i^{th} diagonal element of m. If m_i is non-zero, the inverse of m_i is $1/m_i$. Otherwise, the inverse of m_i is ∞.*

The rational for changing the rule for inverting a zero value comes from the meaning of a 0 entry in the multiplier matrix. If a diagonal entry is 0 in m_1, then the corresponding data dimension was not distributed. If the corresponding entry in m_2 is non-zero, then that data dimension will be distributed. With this interpretation, the $1/0$ is treated as a wildcard ∞, because the data from one node is spread over many nodes in that dimension. Some information is lost by trying to represent this as a simple linear function. This representation assumes the data from one node is spread over all the other nodes in that dimension, but the data may only be spread over a subset of the nodes in that dimension.

For an example of calculating a node-to-node map, consider the data-to-node maps for two instances of A defined in Equations 3.2 and 3.3. The node-to-node map matrices are:

$$\alpha = S_{A''} \cdot m_{A''} \cdot m_A^+ \cdot S_A^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & \infty \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

$$\beta = S_{A''} \cdot o_{A''} - \alpha \cdot S_A \cdot o_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

With these values for α and β the node-to-node function is:

$$pmap(\vec{p}) = \alpha \cdot \vec{p} + \beta = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ p_1 + 1 \end{pmatrix}$$

In this case the data originating at (p_1, p_2) will be moved to node $(0, p_1 + 1)$.

3.2.3 Using communication maps

There are three main types of information to be gleaned from the node communication maps

- Are two communication maps equivalent? Do they represent the same communication pattern?
- Are the ranges of the communication maps disjoint?
- Are the communication patterns “simple”? Can the compiler recognize the pattern?

If communication maps are *equivalent*, the compiler can use the same resource reservations and scheduling when generating code for both patterns. It is trivial to determine whether two maps are equivalent if the node maps are proper or constant functions. The compiler needs only to compare the multipliers and offsets. If the functions involve scattering, the slice or loop bounds are needed to determine whether two node maps are equivalent.

If the ranges of two patterns are disjoint, the compiler can limit which interactions will occur between the two patterns in its scheduling calculations. In general to calculate whether the ranges of the functions are disjoint, the compiler must know the bounds of the input set. However, the input bounds are not always needed in the case of the constant function.

$$pmap_1(\vec{p}) = \begin{pmatrix} p_1 \\ 1 \end{pmatrix} \quad pmap_2(\vec{p}) = \begin{pmatrix} p_1 \\ 3 \end{pmatrix}$$

For example, $pmap_1$ and $pmap_2$ map to disjoint ranges, columns 1 and 3.

The compiler can use the structure of the node-to-node map to recognize many communication patterns. The structure of the multiplier matrix α describes any dimension changes in the communication. If α is a diagonal matrix, no dimension redistributions are required for this communication pattern. Otherwise, α is a row permutation of a diagonal matrix, and the non-zero entries of α define which dimension redistributions must occur. If entry k of row i is non-zero, then data distributed along node dimension k will be redistributed to node dimension i.

Some rows of α may contain only zero values. If row i contains only zeros, no data will be distributed along dimension i. If row i contains the wildcard value ∞, then each data element distributed along dimension i will be scattered to several nodes in the new mapping.[3]

If the entries of α form a diagonal matrix and are all zero or one, the offset vector describes all remaining communication as shifts along each dimension.

3.2.4 Maps on a finite physical array

The previous definitions of communication maps assume that the compiler is mapping to an infinite set of virtual nodes. In reality, the compiler targets a finite set of real nodes.

The finite $pmap$ may return non-integer values for a given node index. This means the input node will be sending to all nodes that match combinations of the ceiling and the floor of the $pmap$ function. This does not occur in the infinite case, because each virtual node is only responsible for a block of one data item for a given array. For example, consider the following map:

$$pmap(\vec{p}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \vec{p} + \begin{pmatrix} 3/4 \\ 7/4 \end{pmatrix}$$

This map indicates that processor $\vec{p} = (p_1, p_2)$ will send data to four different processors $(p_1, p_2 + 1)$, $(p_1 + 1, p_2 + 1)$, $(p_1, p_2 + 2)$, and $(p_1 + 1, p_2 + 2)$.

To take the smaller number of nodes into account, up to two arguments must be added depending on the data distribution. For a block distribution, the map needs a measure of block size. For a cyclic distribution, the map needs a measure of physical node array size. A block-cyclic distribution requires both block size and physical array size.

The cyclic map map^c is defined from the virtual map map^∞ and the vector of physical dimension sizes $\vec{P}$ as follows:

$$map^c(\vec{\imath}) = map^\infty(\vec{\imath}) \bmod \vec{P}$$

where the vector modulo operation is defined as follows

$$\vec{v} \bmod \vec{P} = (..., v_i \bmod p_i, ...)^T$$

The map for a block distribution requires information about the length of each dimension of the data array and the length of each dimension of the processor array. If $\vec{n}$ is a vector of the data array dimension lengths and $\vec{P}$ is the array of physical processor dimension lengths, then

[3] In the current prototype, the compiler simply broadcasts the data to all nodes along the dimension.

the block size matrix $Bsize$ has diagonal entries that correspond to $\lceil (S \cdot m \cdot \vec{n})/\vec{P} \rceil$. With the block size matrix, the block distribution map is

$$map^b(\vec{\imath}) = \lfloor Bsize^{-1} \cdot map^{\infty}(\vec{\imath}) \rfloor$$

For the block cyclic distribution, the block sizes (the diagonal entries of $Bsize$) can be any positive integer value. With a diagonal matrix of block sizes $Bsize$ and the vector of processor dimension lengths $\vec{P}$, the map for the general block cyclic distribution is:

$$map^{b/c}(\vec{\imath}) = (Bsize^{-1} \cdot map^{\infty}(\vec{\imath})) \bmod \vec{P}$$

The node-to-node map can be adjusted for various distribution patterns by constructing it from data-to-node maps that have been adjusted for finite machines. Consider creating a node-to-node map from

$$map_1(\vec{\imath}) = B_1^{-1} \cdot S_1 \cdot (m_1 \cdot \vec{\imath} + \vec{o_1}) \bmod \vec{P}$$

$$map_2(\vec{\imath}) = B_2^{-1} \cdot S_2 \cdot (m_2 \cdot \vec{\imath} + \vec{o_2}) \bmod \vec{P}$$

distributed in a block-cyclic fashion over $\vec{P}$ nodes with block sizes B_1 and B_2.

Using the definition of the matrix α and vector β in Equation 3.4, the new multipliers and offsets for corresponding entries are[4]:

$$pmap_{1\to 2}(\vec{p}) = (\alpha \cdot \vec{p} + \beta) \bmod \vec{P}$$

$$\alpha = B_2^{-1} \cdot S_2 \cdot m_2 \cdot m_1^{+} \cdot S_1^{-1} \cdot B_1$$
$$\beta = B_2^{-1} \cdot S_2 \cdot (\vec{o_2} - m_2 \cdot m_1^{+} \cdot \vec{o_1})$$

After retargeting the communication maps to finite real machines, there may be more communication maps that describe the same real communication pattern. To determine whether two maps describe the same communication pattern, we must also consider the block size matrix and the processor vector. For cyclic and block-cyclic distributions, we test the multipliers and offsets modulo $\vec{P}$. If $m \geq 0$ and $\vec{o} \geq 0$, we can distribute the modulo operation as:

$$\alpha \cdot \vec{p} + \beta \equiv ((\alpha \cdot \vec{p}) \bmod \vec{P}) + (\beta \bmod \vec{P}) \pmod{\vec{P}}$$

Therefore, given two mappings we can simply compare the values of the multiplier matrices and offset vectors modulo $\vec{P}$ to determine whether the maps describe the same communication pattern. Given $pmap_1(\vec{p}) = \alpha_1 \cdot \vec{p} + \beta_1 \bmod \vec{P}$ and $pmap_2(\vec{p}) = \alpha_2 \cdot \vec{p} + \beta_2 \bmod \vec{P}$, $pmap_1$ and $pmap_2$ describe the same communication pattern if $\alpha_1 \cdot \vec{p} \equiv \alpha_2 \cdot \vec{p} \pmod{\vec{P}}$ and $\beta_1 \equiv \beta_2 \pmod{\vec{P}}$.

Once adjusted with block size and processor vectors, the node-to-node map accurately describes which physical processors must exchange data to satisfy the communication pattern. If the target topology is a connected as an n-dimensional mesh or torus, the node-to-node maps also directly map to the target topology, and the communication maps can be used directly

[4]Where m^{+} is the extended invert diagonal matrix defined in Definition 1.

```
do j = 1,n
   A(j,:)=A(j,:)-Arow1(:)*Acol(j)
enddo
```

Figure 3.9: Depending on which dimensions of `A` *are distributed, one or both of* `Arow1` *and* `Acol` *require loop-carried communication.*

to calculate routing and link resource usage. However, for target systems with alternative topologies, an additional mapping from the node-to-node maps to the actual topology is required. For regular topologies such as fat-trees, this mapping is relatively straightforward, but for irregular topologies, additional process mapping algorithms must be used to determine the best way to map the regular problem to an irregular topology[KL70, Sto77, Lo88]. For these irregular topologies, the node-to-node map still provides valuable information about endpoint communication requirements, but it is unlikely that the node-to-node map alone will be useful for directly managing link-level resources.

3.3 Replication and privatization

The compiler can use the program's data dependences to determine which loop's iterations can be executed in parallel. If there are no loop-carried dependences, the loop iterations can be executed independently in any order. However, traditional dependence analysis concentrates on three types of dependences: true dependence (read after write), output dependence (write after write), and anti-dependence (write after read). A compiler for a distributed memory machine must also consider the fourth possible dependence: input dependence (read after read). If the array dimension associated with the input dependence is distributed, the input dependence will require communication.

Consider the example loop in Figure 3.9. If the second dimension of `A` is distributed, the elements of `Arow1` can also be distributed, but the elements of `Acol` are required by each distributed column of `A`. This data requirement can be satisfied by runtime loop-carried communication, but this loop-carried communication prohibits parallelizing the loop. Instead `Acol` can be replicated over all nodes before entering the loop, then all nodes have a local read-only copy of `Acol` while executing the loop.

After the communication analysis phase, the compiler checks all loops in the program that have no loop-carried data dependences. If the loop also has no loop-carried communication, the compiler distributes the loop. If the loop does require loop-carried communication, the compiler determines if it is legal to replace the loop-carried communication with a replication. If all loop-carried communications can be replaced with replications, the compiler then distributes the loop.

Even if the resulting loop cannot be parallelized, replicating arrays may still be beneficial, because the single replication cost may be cheaper than the many loop-carried communication costs. Chatterjee et al. address these trade offs in a more sophisticated network flow algorithm[CGS93]. Their algorithm considers replication due to explicit parallel constructs like `spread` in addition to loop-carried communication.

One problem with this approach is determining the tradeoff between increased memory requirements for replicated variables and the increased execution time due to the lost parallelism. In the prototype compiler, I use the rank of the array as a heuristic for using replication. An array is a candidate for replication if it has a smaller number of dimensions than other arrays in the same bundle.

By using more information about the data dependences in the program, the compiler can possibly eliminate more loop-carried dependences and communication. With array privatization, the compiler attempts to find arrays that are written in the iteration before they are read, so the data in the array is "private" with respect to the iteration[TP92]. The analysis concentrates on array references with indices that are affine function of loop bounds. These arrays tend to be used as local temporaries, so each node can use separate private copies of these arrays permitting the loop to be parallelized. The Fx compiler does not explicitly do the analysis for array privatization, but it does recognize array kills when the entire array is rewritten and avoids inserting indentity conflict arcs in those cases.

3.4 Assigning logical arrays to physical arrays

The data parallel program manipulates logical arrays. Once data is mapped to a distributed system, the logical arrays must be mapped to distributed physical arrays. At different points in the program, the logical array may be mapped to the system differently. These mappings may require physical arrays with different shapes or sizes.

Figure 3.10 shows three redistributions of the logical array `A`. In Figure 3.10(a), `A` is distributed by columns at one point in the program and is distributed by rows at another point in the program. The physical arrays required on the nodes for these two distributions have different shapes. In Figure 3.10(b), `A` is first distributed by columns. Then all of `A` is stored in a larger array on a single node. This case requires two physical arrays of different sizes. Figure 3.10(c) shows the case were `A`'s location is shifted. The physical arrays are the same size and shape in each distribution, but the assignment from data elements to nodes changes.

The prototype compiler handles requirements for different shapes by assigning a different physical array to each instance of the logical array that requires a different shape. Different shapes only apply to array instances that differ by dimension or block size. Array instances that differ only by offset can use physical arrays of the same shape. Therefore, the number of different physical arrays needed for a single parallel array is a function of the number of array dimensions and the number of different block sizes.

3.5 Chapter summary

This chapter describes the phases I implemented to support data placement and communication analysis in the prototype compiler. The prototype compiler can rely on user directives or data placement analysis to determine how data should be mapped to the target machine. This data placement information shows what communication is required in the program. The data

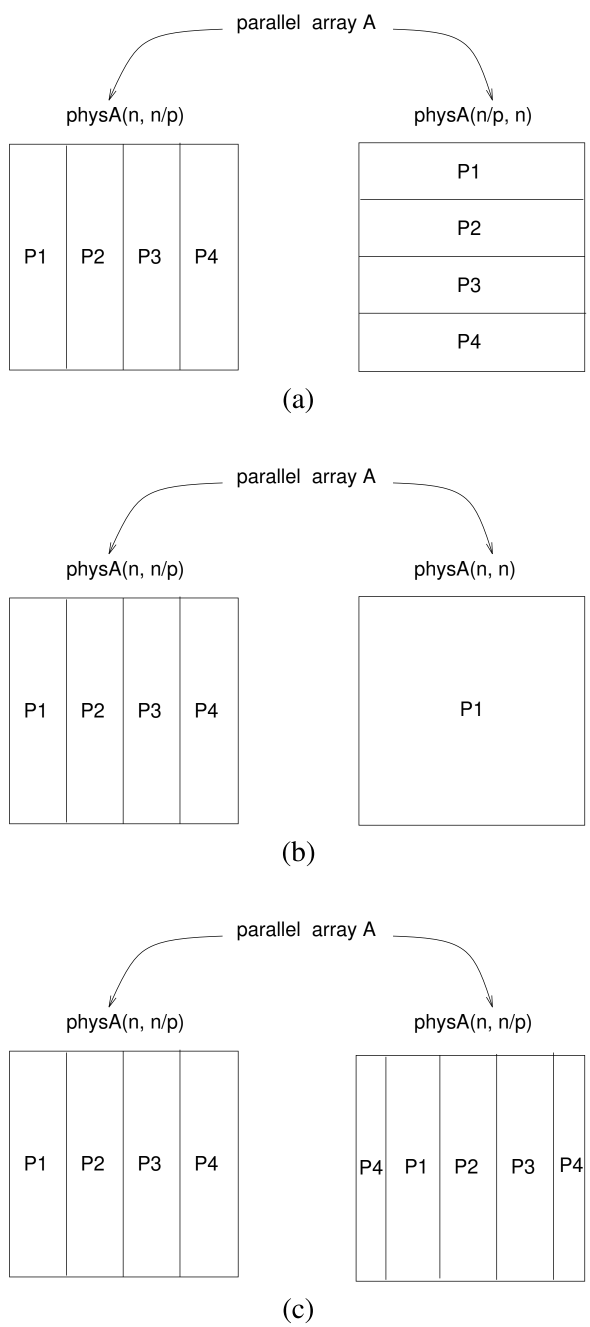


Figure 3.10: Three cases of changing logical array to physical array assignments. Case (a) requires differently shaped physical arrays. Case (b) requires different sized physical arrays. Case (c) reassigns logical array elements within physical arrays of the same size and shape.

placement analysis builds on previous work, but some adjustments were necessary to make the distribution algorithm practical for the Fx language.

I developed a concise communication map representation. Later communication phases use the structure of the communication maps to simplify their operations. The compiler derives the node-to-node communication maps from the data-to-node maps produced by the data placement phase.

The replication phase attempts to expose additional parallelism by converting loop-carried point-to-point communication to a single replication communication outside of the loop. Even in cases where parallelism is not exposed, replacing loop-carried communication with replication can result in more efficient communication. This replication calculation still requires calculations to determine the best trade-off between space for the replication and the reduced execution time.

Once the data mapping has been calculated, the compiler must assign logical arrays to physical arrays for each array occurrence in the program. A single logical array may be mapped to several different physical arrays during the course of the program, because of different array distributions. The number of different physical arrays should be small, because the compiler reuses array space when the size and shape of the array does not change. If the memory requirements are too high, the compiler writer could use the Fortran `equivalence` statement to reuse the same memory locations with different shapes.

Chapter 4

Architecture-directed communication code selection

Chapter 3 described how a parallelizing compiler can detect and manipulate regular communication patterns required in a data parallel program. With information about the communication patterns and the target parallel machine, the compiler can make informed decisions about what type of communication code to generate.

Chapter 2.3 introduced two communication models: the static and dynamic communication resource reservation models. The dynamic model requires little compile-time information and is quite adaptive to runtime changes. The static reservation model can be more efficient, but it requires more compile-time information. With the communication pattern information, the compiler is well suited to taking care of the details that arise from using a static reservation communication model. This chapter describes how a parallelizing compiler can use information about the target machine to select the best communication model for communication code generation. In particular, this chapter describes the compiler phase shown as Simple communication code selection in Figure 1.2.

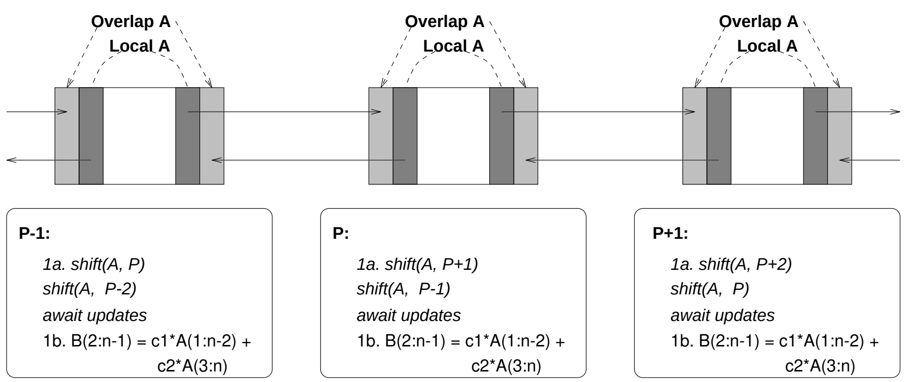
4.1 Communication code selection issues

The communication in Successive Over Relaxation (SOR) is one simple example of a communication pattern that can be recognized at compile time. Figure 4.1(a) shows data parallel pseudo-code for SOR with the required communication steps added (1a and 2a). Consider one step of the SOR computation shown in Figure 4.1(b). In the parallel implementation, both `A` and `B` are distributed over a ring of nodes. Each node owns a subregion of each array, and the compiler detects that each node shares regions of data with its neighbors[Ger90]. The compiler creates overlap areas, where nodes store duplicated data that is owned by the neighboring node but is needed locally for the next computation step. The top of Figure 4.1(b) shows how portions of the `A` array are distributed over three nodes and what communication is needed to keep the overlap regions up to date.

The overlap shift communication can be implemented with static resource reservation over connections or with dynamic resource reservation in the deposit model. The static connection-based implementation has two performance advantages. First, the resources are reserved

```
0.    A = initial values
      do i=1,iter_cnt
1a.      overlap_shift(A)
1b.      B(2:n-1) = c1*A(1:n-2) + c2*A(3:n)
2a.      overlap_shift(B)
2b.      A(2:n-1) = c1*B(1:n-2) + c2*B(3:n)
      enddo
```

(a)



(b)

Figure 4.1: (a) SOR data parallel pseudo-code. (b) One communication step of SOR.

once for the connection, and many messages pass over the connection amortizing the startup overheads. Second, if the connection overhead is low enough, the connection implementation does not first need to pack data into a temporary buffer. Instead it can directly send from and receive into the local array.

Once the compiler has decided on a single communication model, there may be several reasonable implementations in that model. In the dynamic model, different routing methods can be used that trade resources for execution time. For example, on iWarp either torus or mesh routing can be used. The torus routing has a shorter average message distance but requires more resources than the mesh routing.

In the static model, a single communication pattern may be satisfied by several different algorithms. For example, a broadcast can be satisfied by an implementation that communicates over a ring of connections or by an implementation that communicates over a tree of connections. The ring-based implementation will take $O(P)$ steps. The tree-based implementation will take $O(\log P)$ steps, but the tree-based implementation may require more communication resources.

4.2 Communication code selection algorithm

After the communication analysis phase described in Chapter 3, the compiler knows which communication patterns are required in the program. The analysis phase passes on this information by annotating the intermediate representation of the program with communication maps to mark where and what type of communication is required. The annotated intermediate representation and the target machine characterization are input to a *code selection algorithm*. The algorithm uses the communication maps, machine description, array sizes, and iteration counts to chose the best communication model for each communication pattern. The output of this algorithm is an assignment from communication maps to implementations.

The architecture-based communication performance information is collected from an expert on the performance of the target system. The expert knows a number of different implementations that are available for performing a set of common communication patterns. The set of communication patterns recognized by the expert is not complete. Some communication patterns are not considered, and the performance of other communication patterns may not be improved with additional information. The target system expert may be a human (as was the case for the iWarp and Paragon characterizations in this thesis), or potentially, some additional tool could systematically characterize the communication performance on the target parallel system.

This expert information is encoded into a *target-comm-list* that is used by the code selection algorithm. The target-comm-list has entries for each recognized communication pattern. Each entry is a list of information about the set of implementations that can satisfy the indexed communication pattern. Each implementation entry contains information including an execution time model, a set of resource requirements, and a description of how to invoke the implementation.

Figure 4.2 shows an example entry in the target-comm-list. For the prototype compiler, the execution time is estimated by a function of the data size. The resource requirements for the iWarp architecture measure how many logical channels are required on each node.

Template pattern i

Implementation 1
  Time: 80 + 50 x M
  Resource count: 4
  Function name: "connshift"
  ...

.
.
.

Implementation n
  Time: 500 + 75 x M
  Resource count: 0
  Function name: "depshift"
  ...

Figure 4.2: Example entry in the target-comm-list.

for each communication pattern P_i **do**
  entry ← MatchPattern(P_i, *target-comm-list*)
  if *entry* = $\emptyset$ **then**
    Generate default message passing code
  else
    Select fastest implementation that fits in available resources
  endif
endfor

Figure 4.3: Pseudo-code for the communication code selection algorithm.

Communication resources are discussed in greater detail in Chapter 5. In this implementation, the different communication options are implemented by calls to a communication library, so the target-comm-list entry describes the function call and the argument list for each alternative implementation.

Given the communication maps that describe the communication in the program and the architecture-specific performance information in the target-comm-list, the pseudo-code in Figure 4.3 outlines the simple code selection algorithm. A list of the unique communication maps in the annotated intermediate representation are input to the algorithm. For each communication map, the algorithm tries to find a matching template in the target-comm-list. If an entry is found, the algorithm selects the best implementation given the number of available communication resources. If no matching entry is found, the default message passing code is used.

The code selection algorithm relies on a cost model of the possible communication implementations. Similarly, the data distribution algorithm described in Section 3.1.2 relies on a communication cost model to determine which array dimensions to distribute. As with all good compiler problems, communication cost estimates are useful at several points in the compilation, so dividing and ordering the compiler phases is not straightforward. I use a less accurate communication model for the distribution phase. The performance differences between performing an all-to-all communication or no communication are obvious with relatively rough communication cost estimates. However, one could consider iteratively performing the distribution and code selection phases and feeding back information about the cost models between the two phases.

The MatchPattern function uses the structure of the communication map to match the communication map to the appropriate template entry. Section 3.2.3 describes how the structure of the communication map corresponds to different types of data movement. A single communication map may also match several communication templates. For example, a single communication step may require both a shift and a dimension change communication. The communication templates are ordered by the expected cost of the corresponding communication. The cost of the more expensive communication pattern dominates the cost of the communication step.

This simple algorithm works well if there are sufficient communication resources to satisfy the best communication implementation for each communication pattern. This algorithm will also find a valid implementation assignment for programs that require more resources. However, with the communication sequence information, the compiler can make better decisions about which communication patterns would most benefit from faster more resource intensive implementations. Chapter 5 describes an alternative resource management approach to code selection. This approach relies on several simple algorithms that enable the compiler to do a better job of managing the use of limited communication resources.

4.3 Communication code generation

After the implementations have been chosen for each communication map, the code generation step inserts the necessary communication code. The compiler can insert library calls with the appropriate parameters for the communication pattern, or it can directly insert the code needed to satisfy the communication pattern. While directly inserting the communication code may enable better optimizations for the single node code, it can result in unacceptable code growth. A library of communication routines simplifies code generation and reduces code growth perhaps at the cost of limiting single node code optimizations. The prototype compiler inserts function calls to simplify the implementation.

In addition to inserting communication code, the compiler must keep track of what resources are required for all of the communication implementations. After all implementations have been chosen for all communication patterns, the compiler must generate initialization code for the static connections and resource reservations for the pool of resources required for the dynamic communication.

The compiler will not always have complete information at compile-time. For example, some loop bounds, array indices, or array slice bounds may not be known at compile-time.

There are several problems introduced by this lack of compile-time information. The accuracy of the estimates of communication time are affected by unknown loop bounds and array slice bounds. The compiler uses a constant value if the upper and lower loop bounds or array slice bounds are unknown. If the strides or array indices are unknown, compiler will not always be able to fully analyze all communication patterns at compile-time. When the compiler cannot analyze a communication pattern, it must rely on runtime data and dynamic tests as a fall back position. The compiler is capable of generating communication code that uses runtime information to produce and consume the appropriate messages[Sti93].

For a given statement, the compiler may be able to recognize some communication patterns but not others. Rather than mix informed and uninformed implementations in a single statement, the prototype compiler uses the default message passing communication for all communication patterns of the statement if any communication pattern cannot be analyzed.

Additional array assignments may be required to resolve different physical arrays representing the same logical array along different program control paths as described in Section 3.4. The compiler inserts assignments where necessary at the join points of the control flow of the program, similar to the ϕ-functions inserted for static single assignment form (SSA)[CFR$^+$88].

Function and procedure calls can also introduce uncertainty about the mapping of arrays. In the absence of any knowledge of the function, the compiler must assume that all argument arrays and global arrays have moved. In this case, the compiler must insert runtime checks and communication code to return arrays to a known mapping. However, the compiler often has some knowledge about function calls. Efficient data parallel functions can be compiled with the addition of a small set of directives that indicate whether the caller or callee moves procedures on entry and exit[YO94]. The DYNAMIC directive in HPF gives the compiler some of this information by indicating whether an array can be dynamically redistributed. Interprocedural analysis can also be performed to calculate the required communication patterns from the caller code to the callee code.

4.4 Communication code selection evaluation

To illustrate the effects of the communication code selection algorithm, we present measurements from four program kernels: blocked matrix multiplication, successive over relaxation (SOR), two dimensional fast Fourier transform (2D FFT), and Gauss Jordan elimination with pivoting. This is by no means an exhaustive list of dense, scientific programs, but the communication patterns in these programs are representative of the communication patterns that occur in many dense linear algebra and signal processing codes.

SOR Requires nearest neighbor communication into duplicated ghost or overlap regions. This style of communication is required for the class of iterative, stencil computation.

Blocked matrix multiplication Requires nearest-neighbor communication. Many blocked algorithms require a nearest neighbor shifting of the data set[KS91]. During a blocked step, each node calculates on local data, and then shifts the local data to the neighboring processor. Pipelining replication with the next computation step also requires nearest neighbor communication.

Gauss Jordan elimination Requires replication communication. Many elimination and other matrix algorithms require replication of a special row or column (e.g., a pivot row). Both simplex and matrix multiplication can be implemented using replication communication.

2D FFT Requires two all-to-all communication steps. This dense redistribution communication pattern is representative of computations that operate on the data in different phases. For example, some simulation programs (such as airshed modeling) operate over one dimension in the first phase and another dimension in the second phase.

These communication patterns appear in many of the dense linear algebra programs I surveyed. One communication pattern that is missing, is communication based on mismatched strides. For the few programs that I found that required non-unit strides, the compiler could adjust the block size for those arrays to avoid communication. For programs that do have unavoidable stride conflicts, the resulting communication will likely be dense, so the all-to-all communication of the 2D FFT program is a good representative.

For each program, we measured two versions: one that uses the communication implementations suggested by the code selection algorithm and another that uses the best instance of the alternative communication model. All other communication optimizations are performed on both versions, so the effects of differing communication models are isolated. These programs were executed on a 64 node iWarp system and a 60 node Paragon system running SUNMOS[1].

The information in the *target-comm-list* for each architecture can be summarized as follows. For iWarp, the statically reserved connection implementation is always superior to the dynamic resource reservation implementation. The only limit to using the static connection implementation is the limited number of resources to implement connections.

For Paragon, the software connection protocol is superior for sparse communication patterns. For dense communication patterns, the cost of the barrier synchronization required for the dynamic deposit model is amortized, and the barrier synchronization provides another mechanism for network congestion control. For sparse patterns in these programs, NX is competitive with the connection protocol. However, even for these relatively synchronous data parallel programs, the NX implementation requires a substantial amount of system buffer space. For dense patterns, buffering costs dominate, and the NX implementation is not competitive with either the connection or deposit implementations.

4.4.1 iWarp measurements

Figure 4.4 compares the absolute communication performance for all four programs on an 8×8 iWarp. Figure 4.5 shows the normalized communication and computation performance of these programs. Table 4.1(a) shows the percentage change in total execution time from the communication code selected by the algorithm to the communication code in the alternative communication model.

The major arrays in the blocked matrix multiplication are distributed over two dimensions, by rows and by columns. This application requires nearest neighbor shifts. There is no network contention for either the message passing or connection-based implementations, so only the effects of improved resource reservation are apparent.

[1] This system uses A-step network interface chips (NICs).

The arrays of this SOR application are also distributed over two dimensions. SOR also involves nearest neighbor communication, but in this case, the non-local data can be stored in overlap regions as described in Section 4.1. With overlap communication, the connection model can avoid the sender's packing step, so in addition to improved resource reservation this pattern benefits from eliminating a memory copy.

The 2D FFT application iterates over 2D FFTs for many samples. Each 2D FFT requires two all-to-all communication steps. In addition to the improved resource reservation, the all-to-all communication benefits from the improved control of network scheduling in the static connection-based implementation.

The major array of the Gauss Jordan elimination is distributed by columns. It was run on a data set that requires pivoting in each iteration. Each iteration of the elimination requires the replication of the pivot column. Both the dynamic message-based and static connection-based implementations follow a similar pattern for propagating the broadcast data. The connection-based implementation benefits from improved resource reservation.

In all cases, the static connection-based implementation shows substantially improved communication performance. For the communication patterns in the blocked matrix multiplication and Gauss Jordan elimination, the major effect of the static resource reservation is the reduced communication startup overhead. Therefore, as the problem size increases for both cases, the relative impact of the lower startup overhead is reduced. The all-to-all communication in the 2D FFT also benefits from superior control of network routing, and SOR also benefits from eliminating a buffering step. Therefore, the relative impact on communication time for these two programs does not decrease for larger program sizes.

4.4.2 Paragon measurements

The superiority of one communication model over the other is not clear for Paragon, since both models rely on dynamic resource reservation at the hardware level. Figure 4.6 compares the communication performance for all four programs on a 60 node Paragon system. I concentrated on the performance of the deposit model for dynamic resource reservation. Figure 4.7 shows the normalized performance of communication and computation for these programs. Table 4.1(b) shows the effect of the different communication models on total execution time.

The major arrays in the blocked matrix multiplication are distributed over both machine dimensions. The problem requires nearest neighbor communication, so the performance of the software-based connection model is superior.

I measured five iterations of an SOR application. The arrays in the SOR application are distributed by columns over one dimension. The sparse, nearest-neighbor communication pattern encounters little congestion, so the connection-based choice is superior. However, the connection-based overheads are not so low that the connection-based implementation can directly communicate from the processor to avoid a packing step as the iWarp implementation does.

I measured ten iterations of a 2D FFT algorithm. The major array in the 2D FFT is distributed by columns initially, and each 2D FFT requires two all-to-all communication steps. All-to-all communication is a dense communication pattern, so the deposit model is superior. However, for total execution time, the communication model makes little difference, even for

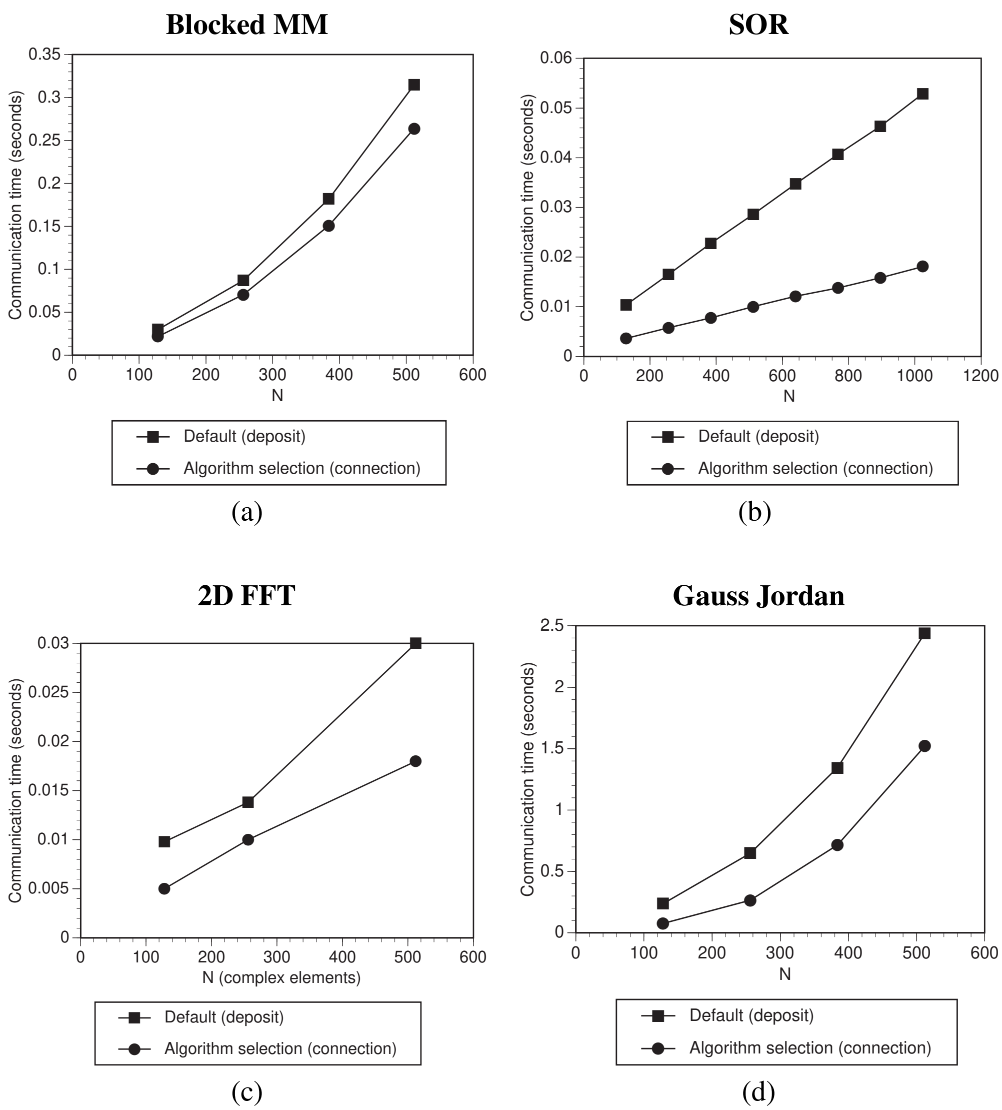


Figure 4.4: Communication performance of different communication models on 8×8 iWarp for (a) blocked matrix multiplication, (b) successive over relaxation, (c) two dimensional FFT, and (d) Gauss Jordan elimination.

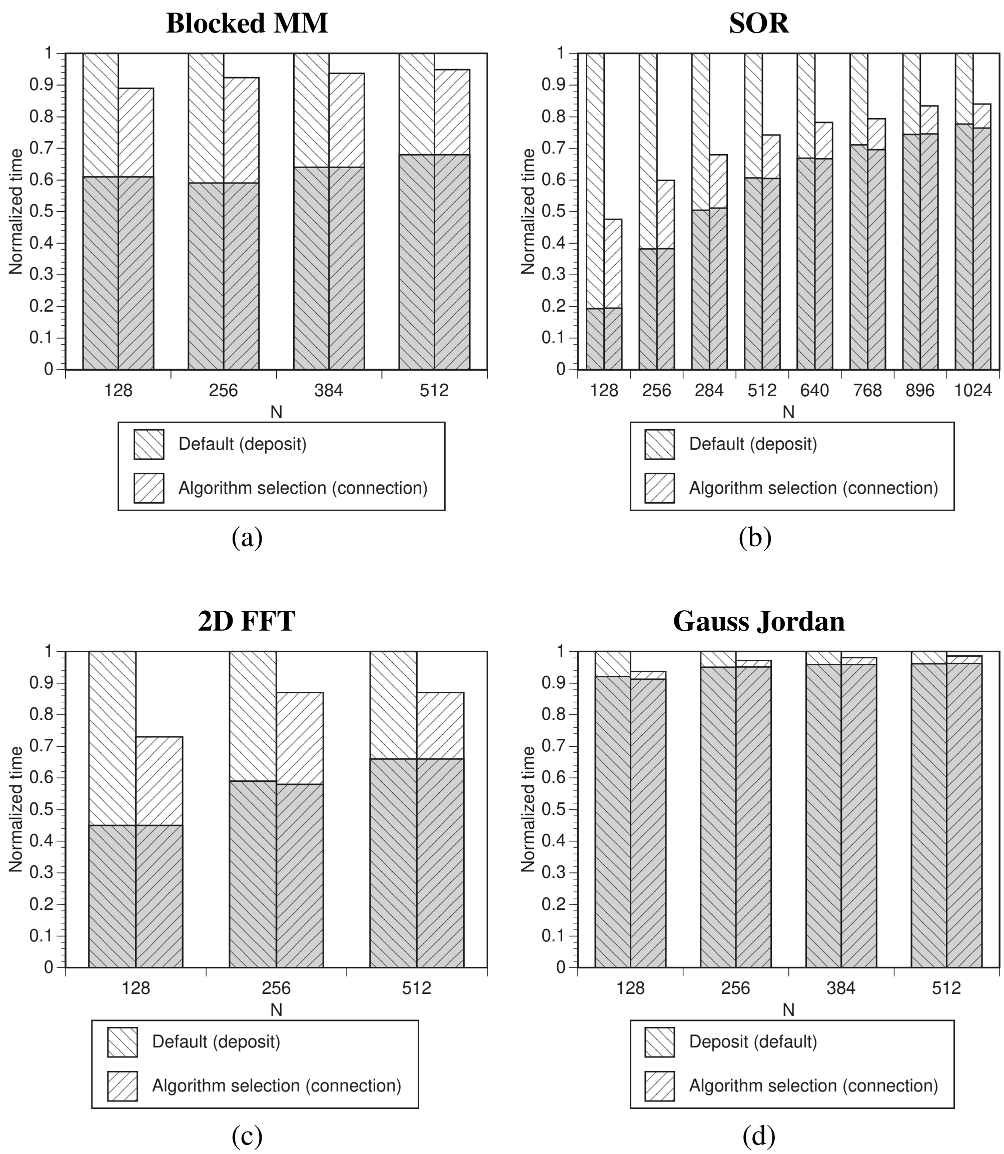


Figure 4.5: Normalized program performance on 8×8 *iWarp for (a) blocked matrix multiplication, (b) successive over relaxation, (c) two dimensional FFT, and (d) Gauss Jordan elimination. The darker areas show the relative amount of time spent in computation, and the lighter areas show the relative amount of time spent in communication.*

iWarp

Problem size	Blocked MM	SOR	2D FFT	Gauss Jordan
128	16%	25%	27%	6%
256	10%	13%	13%	3%
384	7%	9%		2%
512	5%	9%	13%	1%
768		5%		
1024		4%		

(a)

Paragon

Problem size	Blocked MM	SOR	2D FFT	Gauss Jordan
256	36%	20%	-1.1%	7.3%
512	20%	7.2%	-1.8%	4.8%
1024	4.7%	2.5%	-1.6%	3.9%
2048	4.1%	0.6%	-1.7%	1.4%
4096	2.6%	0.5%	-0.6%	0.2%

(b)

Table 4.1: Percentage reduction in total program execution time between the static and dynamic communication models on a 64 node iWarp (a) and a 60 node Paragon (b). Percentages less than zero (as for Paragon 2D FFT) indicate that the dynamic communication model was faster.

	OSF		SUNMOS	
	VCF	NX	VC	NX
Startup overhead (SO)	10 μs	50 μs	50 μs	50 μs
Peak BW	150 MB/s	60 MB/s	150 MB/s	150 MB/s

Table 4.2: Communication characteristics of connections and NX under OSF and SUNMOS operating systems.

small problem sizes.

The major array of the Gauss Jordan elimination is distributed by columns. It was run on a data set that does not require pivoting. The major communication pattern is the replication of the pivot column. Both implementations use information about the physical configuration to replicate data over the physical rows then the physical columns. The real communication performed is between nearest neighbors, so there is little congestion. Therefore, the connection-based implementation shows better performance.

In these measurements, I concentrated on communication models that do not require buffering. By adding buffering, we can compare the performance of the NX library. On the blocked matrix multiplication and the Gauss Jordan elimination programs, the performance of the NX library is almost identical to the performance of the connection-based communication. These two programs are sufficiently well-synchronized that they do not require frequent buffering, although the programs do buffer messages occasionally, so NX still requires substantial system buffering space. The SOR program is less synchronous. The NX implementation must buffer data more frequently, so the connection-based implementation is superior.

4.5 Discussion

The measurements in this chapter show that the choice of the preferable communication model depends on the target machine. The static resource reservation communication is superior over a broader range of communication patterns on iWarp with its support for reserving hardware resources than on Paragon.

Even without direct hardware support, the software-based connection protocol used on the Paragon is beneficial in some cases. A software-based connection protocol avoids system buffering, statically associates information and buffers with the connection, and simplifies the logical communication path, increasing the possibility of lowering communication overheads. With the simpler protocol, optimizations in the data transfer library should be more tractable. For example, the Virtual Channel Facility[MPS93], a connection library developed under OSF, was indeed far more efficient than the regular NX communication library. However, the increased performance exposed more hardware problems making it an unreliable compiler target. Table 4.2 compares the startup overheads and peak bandwidth of NX and the software-based connections under OSF (where VCF was developed) and under SUNMOS (where the connections in this paper were measured).

Since software-based connections do not directly reserve hardware network resources, in-

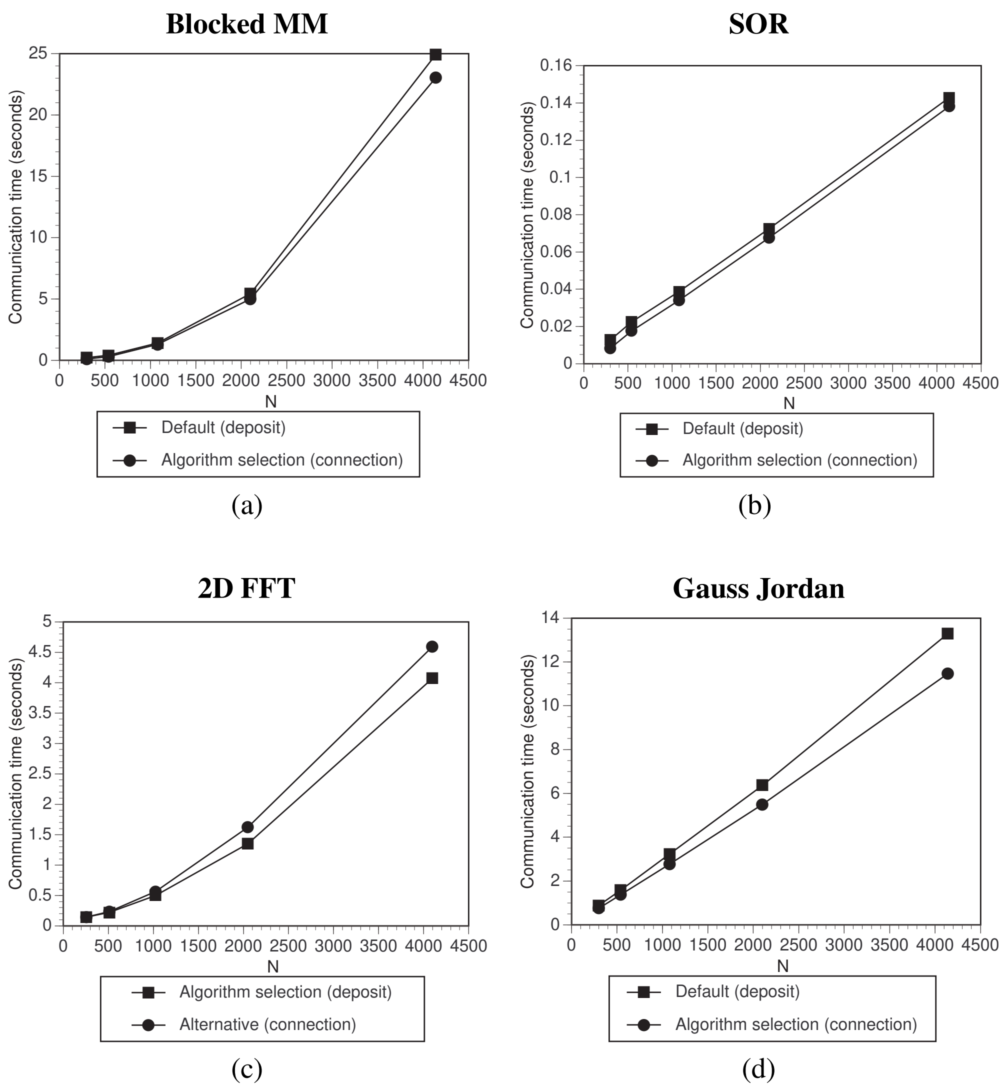


Figure 4.6: Absolute communication time on 60 node Paragon for (a) blocked matrix multiplication, (b) successive over relaxation, (c) two dimensional FFT, and (d) Gauss Jordan elimination.

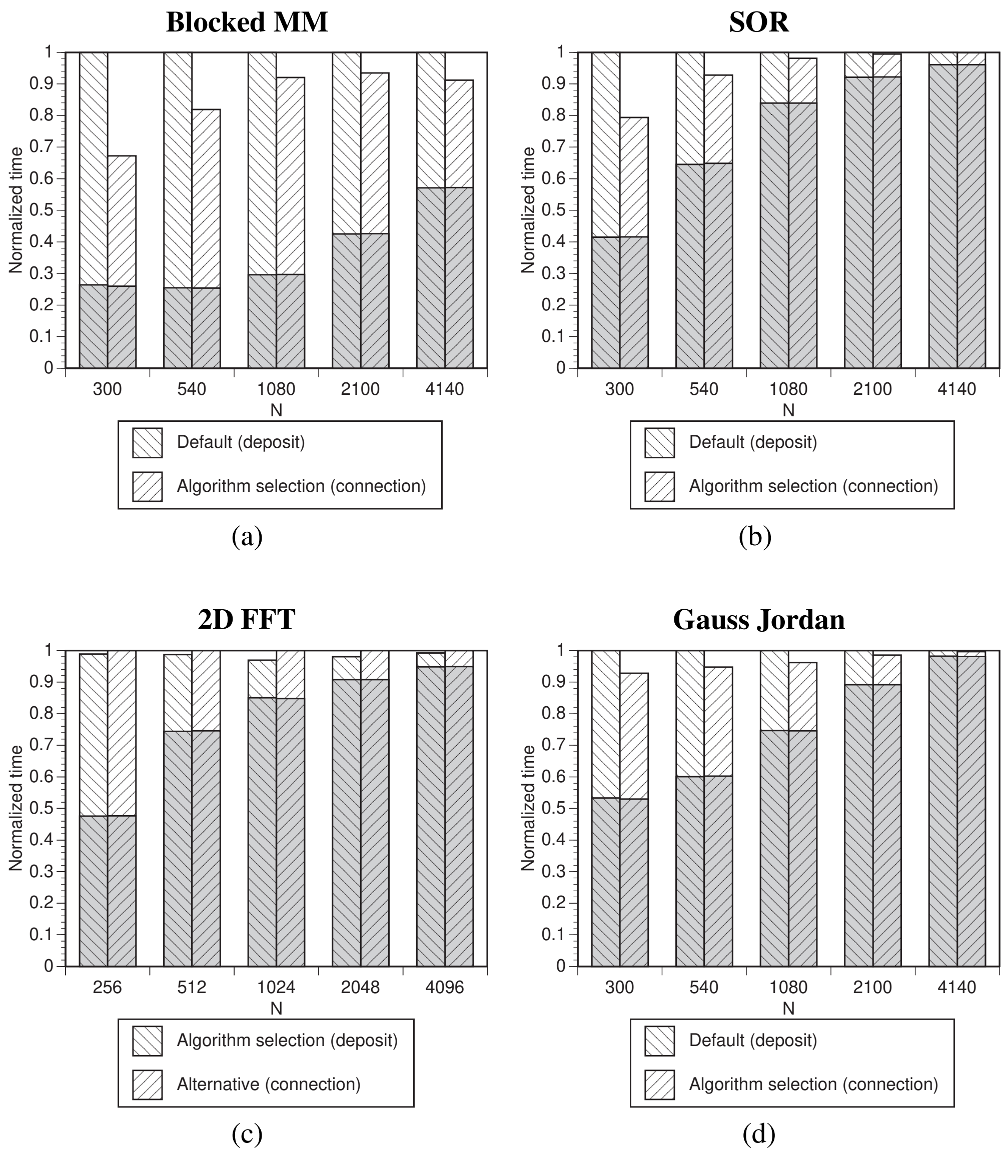


Figure 4.7: Normalize program execution time on 60 node Paragon for (a) blocked matrix multiplication, (b) successive over relaxation, (c) two dimensional FFT, and (d) Gauss Jordan elimination. The darker area shows computation time, and the lighter area shows communication time.

formation about the sequence of communication patterns is not as crucial to target the software-based connections. In this case, the compiler can just generate calls to a collective communication library optimized for the target system. Then the compiler does not need to know as much about the target communication system; the collective communication library can do the analysis to determine for which communication patterns the connection-based protocol is beneficial. For example, the iCC collective communication library[B+94a] developed for the Paragon uses runtime tests of the message size and partition size to select the most appropriate communication algorithm for the collective communication operation. Regardless of whether the compiler or a library is optimizing communication, the software-based connection protocol is another option for optimizing communication performance.

For systems that allow program management of limiting hardware communication resources, compiler information about the sequence of communication patterns is essential to do a good job of resource management. If there are not sufficient resources to open all connections simultaneously, a global view of the program is necessary to determine which communication patterns should use a slower implementation or where static connections should be opened and closed in the program.

For both machines, the choice of the communication model makes a measurable difference in total execution time for most of the application kernels we measured. This change in execution time is larger for smaller problems. As the problem size grows relative to the machine size, the size of the messages exchanged increases, so improvements in communication startup overhead become less important. For most memory-based applications, the cost of computation grows faster than the cost of communication. For example, when multiplying two $N \times N$ matrices, the computation grows as N^3 but only $O(N^2)$ bytes need to be exchanged.

If the input problem size can be scaled large enough relative to the machine size, the communication overhead is negligible. However, many real programs cannot be arbitrarily scaled. For these problems, we would still like to add more computation nodes to speed up the problem. Optimizing communication increases the amount of effective parallelism, making it practical to distribute the arrays into smaller blocks over more processors. Measurements in [SSO+95] show the effect of increased communication efficiency on the scalability of 2D FFT and SOR on the Cray T3D.

4.6 Chapter summary

This chapter describes the simple code selection algorithm that I developed for the Fx compiler. This algorithm uses expert information about communication performance on the target architecture condensed into a simple data structure. With this data structure, the code selection algorithm is relatively straightforward. This chapter presents measurements of several applications, comparing the algorithm's communication code selection with the default dynamic implementation or an alternative implementation. These measurements show that this simple approach does improve the total execution time.

The programs measured in this chapter are relatively small, but they should be seen as building blocks for larger algorithms. If your goal is to solve a single matrix multiplication program, a desktop computer is probably sufficient. Many larger scientific programs and signal processing algorithms use these type of programs as computation steps.

Chapter 5

Communication code selection with limited resources

The previous chapter describes one simple algorithm for selecting from different communication code alternatives. This simple algorithm works well in cases where there is no limiting communication resource or where there are few communication patterns required in the program. However, for larger programs, limited communication resources force the compiler to solve a resource management problem to use the static resource reservation model.

This chapter first describes the limiting communication resources on several parallel systems. Then we abstract away from particular communication resources to define two communication resource management problems and propose several algorithms to solve these problems. These algorithms are used in the Resource management communication code selection phase shown in Figure 1.2. Finally, we present measurements from iWarp to evaluate the effectiveness of these resource management algorithms.

5.1 Examples of communication resources

Regardless of the protocol, communication between nodes in a system requires communication resources. The specific resources depend on the implementation of the communication protocol. For example, statically reserved connections can be implemented with direct hardware support (such as on iWarp) or can be implemented in software on top of some other dynamic hardware protocol (such as done by the VCF library on Paragon).

The availability of communication resources can affect the execution time of transfers in the network and processing at the endpoints. Some network-based communication resources are hardware queues and routing table entries. Some endpoint communication resources are buffering memory and connection tables.

The following sections examine specific communication resources from several parallel systems: iWarp[B+88], the INMOS T9000[MTW93], ATM networks, the Intel Paragon[Div91], and the Stanford Flash[K+94].

5.1.1 iWarp

iWarp supports connections with a limited number of hardware communication queues (logical channels) in the communication agent. Data from different logical channels are multiplexed over the physical bus on a word by word basis. For two nodes to communicate, they build a connection by configuring channels on adjacent nodes to forward data to each other. The connection requires one logical channel on each communicating node and each intermediate node. Figure 5.1 shows several connections in a system with two logical channels per node. When initializing the connection, communicating nodes must know which logical channels are used on their neighboring nodes to correctly initialize the flow-control hardware. For example, in the connection from node (0,0) to node (1,1), node (0,0) must know that logical channel 2 forwards to logical channel 1 on node (1,0), and node (1,0) must know that logical channel 1 forwards to logical channel 1 on node (1,1).

In the example in Figure 5.1, no more connections can involve node (1,1), since all of its logical channels are already in use. If a new long-lived connection is opened at runtime, the source node must insure that there are sufficient logical channels along the projected route. For example, if nodes (0,1) and (1,2) both attempt to open a connection to node (0,2) as shown by the dashed lines in Figure 5.1, only one connection will succeed since only one logical channel remains free on node (0,2).

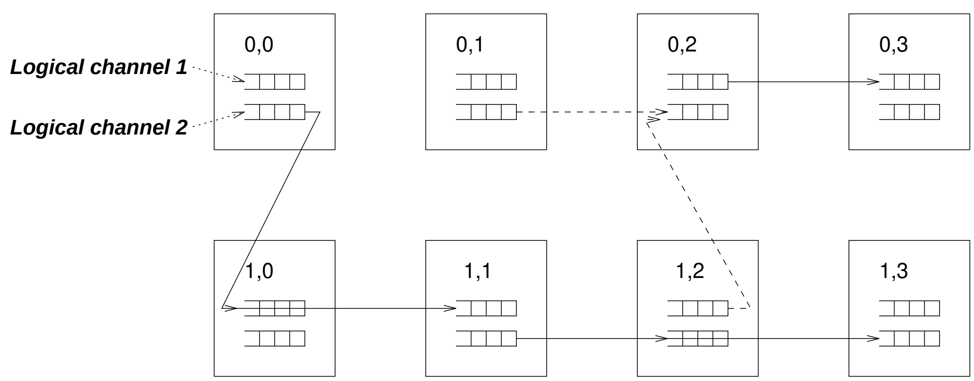


Figure 5.1: Example of connections and logical channel usage. The solid lines show connections. The dashed lines show attempted connections.

On each iWarp communication agent there are 20 logical channels, but some logical channels are reserved by the runtime system and other communication packages, so fewer than 20 logical channels are available for the program's connections. These logical channels are the major limiting communication resource on the iWarp system.

5.1.2 T9000

The INMOS T9000 processors and C104 routing chips are used the create the latest generation of transputer systems. The processors are connected via an interconnection network of routing chips. Each routing chip is a 32×32 crossbar switch. These switches can be used to create most common network topologies: multi-stage network, torus, crossbar, hypercube, etc.

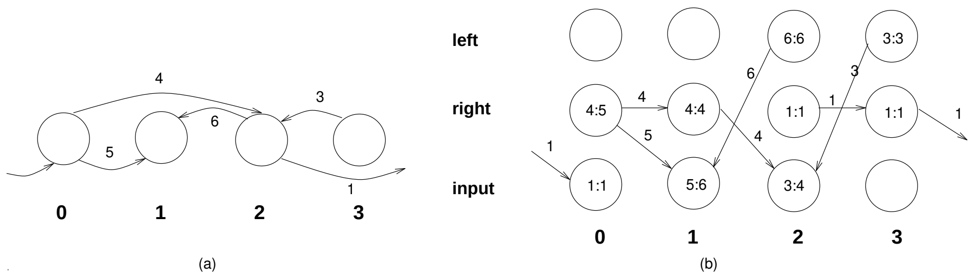


Figure 5.2: The graph (a) shows requested communication pattern. The graph (b) rewrites communication pattern so intervals are made clear.

Connections on this system are implemented by *virtual channels*[Dal92]. Each processor on this system can support an effectively infinite number of virtual channels. The virtual channel state information is stored in a table. When the virtual channel is used, the table is referenced and the data is multiplexed over the corresponding physical link. The size of the endpoint virtual channel table (and number of active virtual channels) must be limited by the size of the endpoint memory, but the routing switches impose a greater restriction on the number of virtual channels, since each routing switch can only support a limited number of different routes.

The switches use interval tables for routing[vLT87]. At the switch, each output channel is represented by a range of numbers, k:k+d. A packet is redirected along a particular channel if the packet ID is in the labeled interval. The same interval can represent several output channels in a switch, and the matching packet is redirected along the first available channel. Intervals that guarantee deadlock-free routing for general communication patterns can be calculated for most common topologies. In addition, given a specific communication pattern, the compiler can in many cases create a set of routing tables that are specialized for that communication pattern.

Figure 5.2 shows how specialized interval tables can be calculated for a simple ring topology. The graph in Figure 5.2(a) is the communication pattern. For a ring, the routing table entries are divided into three sets according to the destination of the arc: left, right, and input. The graph in Figure 5.2(b) shows the communication pattern redrawn with each arc ending in the appropriate set for each node. The route labeled 4 in Figure 5.2(a) starts in the right set of node 0, goes to the right set of node 1, and ends in the input set of node 2. With this new graph, ranges for each set of routing table entries can be calculated based on the routes needed for the communication pattern.

For communication patterns that form a cycle, a single interval range in a given direction may not be sufficient. Consider the communication pattern and interval labeling in Figure 5.3. The interval for the right set on node 0 must be split into two ranges. Node 0 must forward connections labeled 1 and 4, while redirecting the connection labeled 3 to the processor. This splitting must occur on some node because the connections between the right sets form a cycle.

Since space in the interval tables is limited, it may not be possible to have routes set up for all requested implementation functions. In this case, it may be necessary to divide the communication patterns into phases that can be routed together or leave some default (but not necessarily optimal) route table entries for a fall back position. By calculating the routing table information, the compiler can directly manage and specialize the use of the limited routing table resources.

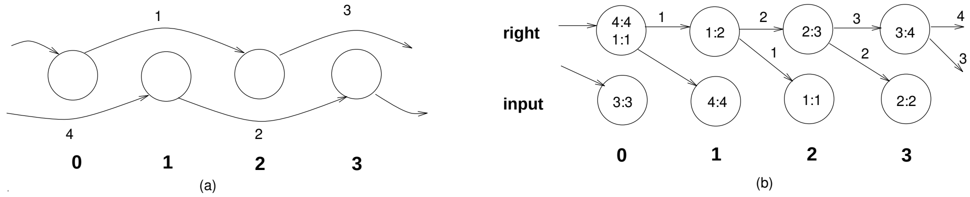


Figure 5.3: Communication pattern that requires more than one interval over one direction.

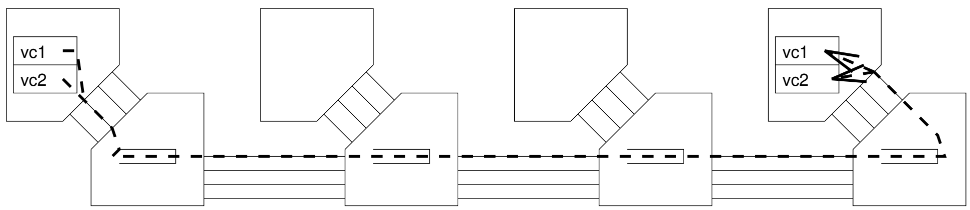


Figure 5.4: Two virtual channels routed over the same physical route utilize all physical bandwidth over that route. A single virtual channel must wait for flow control acknowledgments, so it cannot utilize the complete physical bandwidth.

The communication hardware enforces tight synchronization between the sender and receiver on each virtual channel. Each 32 byte packet must be acknowledged by the receiver before the next packet can be sent. By requiring each packet to be acknowledged before the next packet is sent, the system limits the amount of space needed to buffer messages that arrive too soon. One limitation of this tight channel synchronization is that bandwidth can be wasted while the sender is waiting for acknowledgments. In addition to the message transit time, the equation for sending time must include the acknowledgment time and packet startup time. The equation for the time to send a message of m bytes is $T(m) = 8 \times T_t \times m \times (1 + (2 \times D + 1)/32)$, where T_t is the per bit transit time and D is the number of switches to traverse. The peak physical link bandwidth is 100 Mbit/s. For $D = 8$, the peak bandwidth drops to 78 Mbit/s, and when $D = 16$ the peak channel rate drops to half of the peak physical link rate. For connections over many switches, the compiler can send the message data over several virtual channels routed on the same physical connection. Whenever one virtual channel is waiting for an acknowledgment, the other virtual channel can be sending a packet, so the physical network is used to its full potential. Figure 5.4 shows an example of data sent over two virtual channels vc1 and vc2 that are routed over the same physical connections. If more physical connections are available, the compiler can also split data transfers over multiple physical connections. The benefits of using multiple physical channels depend on how fast the processor can send and receive data. A compiler can certainly calculate this physical channel resource scheduling given communication pattern information and the target system's processor and router parameters.

There are many communication resources in this Transputer system, including endpoint memory for virtual channel tables, routing table entries, and link bandwidth. Depending on the system configuration, any one or more of these resources can be the limiting communication resource.

5.1.3 ATM networks

The asynchronous transfer mode (ATM) network standard is a current attempt to unify voice, video, and data traffic over a common network protocol. In the ATM approach, switches form the interconnection fabric (instead of broadcast media as in Ethernet networks). The ATM standard uses *virtual connections* to separate data streams with different transmission characteristics. Data that travels over virtual connections is divided into small fixed length packets (53 bytes with 48 bytes payload).

With virtual connections, ATM adapters and switches provide hardware support for a connection-based data protocol. The basic elements of the standard are in place and there are several commercial ATM products, but many aspects of the standard are still evolving and the subject of research[BCS93]. These evolving issues include quality of service guarantees[Tur92] and signaling control[CW92] (connection creation and control).

Communication resources are made explicit by quality of service (QOS) guarantees. Each connection can request a QOS guarantee that bounds bandwidth, latency, or jitter (degree of communication variability). For parallel computation, bandwidth and latency are the most interesting variables. While the ATM switch can support a large number of connections, it has a far more limited amount of bandwidth it can provide. There are a number of schemes to efficiently describe bandwidth and latency requests for "bursty"traffic. The compiler can derive additional information to aid these resource management algorithms.

After the communication analysis phase, the compiler has information about communication patterns. The compiler also has information about which communication patterns are active at the same time (i.e. the phases of communication). To reserve communication bandwidth, the connection must give some indication of the amount if bandwidth it requires when it is opened (e.g. maximum or average bandwidth). If the compiler can indicate that groups of communication patterns are disjoint, the system may be able to more intelligently guarantee bandwidth. Alternatively, the compiler can change the QOS bandwidth guarantees between phases. For example, the system in Figure 5.5 will only allow guarantees for 60 MB/s per link. If the connections in Figures 5.5(a) and (b) are opened individually, they cannot all be open at the same time. Instead one set must be used and closed before the other can be opened. However, if the compiler can indicate that the two sets of connections are from disjoint phases, the admission control should allow both sets to be open simultaneously.

If a connection does not receive the amount of bandwidth it requested, the compiler-generated code can use the amount of bandwidth actually reserved to inject data at the appropriate rate. If the system does not have hardware flow control, switches may drop data that arrives too quickly. Sending data at a slower rate is more efficient than re-sending many dropped packets.

However, if the system does have hardware flow control, connection-based communication does not require additional control messages to request data. Instead data that arrives before the node is ready to receive it will stay in intermediate switch buffers and back up through the network. This is similar to connection-based communication on iWarp.

If the system allows nodes to specify routes, the compiler can use the communication pattern information to schedule use of the communication links. However, this implies that the compiler knows the target topology, which is unlikely for a network of workstations.

ATM networked systems also have a number of different communication resources includ-

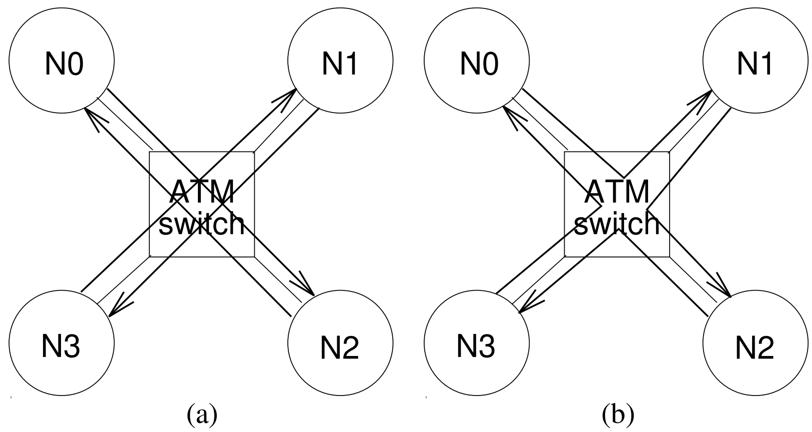


Figure 5.5: Two disjoint sets of connections the parallel application wants to open. Each connection requires 40 MB/s maximum bandwidth. Each link can guarantee up to 60 MB/s maximum bandwidth.

ing physical link bandwidth and switch table space. The limiting communication resource depends are the particular system configuration and how some of the evolving ATM features are eventually implemented.

5.1.4 Paragon

The Intel Paragon is the latest generation of the iPSC series of parallel computers. The nodes on this system are connected in a two dimensional mesh by routing chips. All routing choices on this machine are fixed by the routing chip hardware, so the parallelizing compiler cannot choose the route of a connection. However, routing is deterministic, so the compiler can use information about the router to determine the message schedule of each node. For example, Scott [Sco91] describes an all-to-all communication algorithm that in theory optimally uses the Paragon's network bandwidth. This algorithm knows which routes the Paragon uses and schedules communication in phases, so the routes of messages in each phase use all communication links. By using such techniques, the compiler can manage the network bandwidth resource. This is similar to the physical channel scheduling that the compiler can do for the transputer system.

The Paragon does not have direct hardware support for connections, but connection-based communication can be implemented on top of the packet-based communication hardware as described in Section 2.3.3. Following a connection-based protocol provides structure that is beneficial for software resource reservation and synchronization. Each open connection has some state and buffer space associated with it, and keeping buffers for a large number of connections may not be feasible. A compiler that generates connection-based communication will need to trade off communication speed with the number of active connections. In this case, endpoint memory is a limiting communication resource. For systems with a small number of nodes, this is probably not a factor, since each node has at least 16 MBytes of memory, but as the system size and the number of potential communicating nodes increase, endpoint memory usage may become a limiting factor for static communication resource reservation on Paragon systems.

5.1.5 Flash

Flash is a proposed distributed shared memory machine[K+94]. Each node of the system includes a specialized protocol co-processor called the MAGIC chip that takes care of data movement and cache coherence policies. Since the MAGIC chip is programmable, Flash can flexibly support a number of different cache and data movement policies.

The differences between the bulk transfers and traditional cache coherence protocols are discussed in [HGDG94]. In addition to these protocols, the system programmer can create efficient protocols for other communication patterns. For example, a replication protocol could implement a more efficient algorithm than the naive one-to-many approach, and a simple reduction communication pattern could incorporate some of the computation into the co-processor.

The protocol processing handlers are stored in main memory, but the handlers are accessed from a jump table on the co-processor at runtime. Depending on the size of the jump table, the number of active protocols could be limited.

The protocol processor uses the memory address of a data item to determine which coherence or data transfer protocol to use. If a different protocol is more appropriate for the data array at a different point in the program, the program can re-map or copy the data array or use a single protocol that is not optimal for either communication pattern but reasonable for both.

Based on the Flash design, we can identify a couple potentially limiting communication resources: limited number of active protocols and limited number of protocols per memory location.

5.1.6 General communication resources

This section has described examples of communication resources on several current and upcoming parallel systems. In the static resource reservation model, these resources are exposed to the programmer or compiler for more efficient use and scheduling.

For regular communication patterns on symmetric systems such as torus and hypercubes, the resource requirements are quite uniform over the system. About the same number of resources are required on each node or switch. For other systems (such as meshes), there may be hot spots in the network that are used more frequently. In this case, the resource requirements on the "hot" nodes or switches are representative of the resource requirements for the entire system.

For both types of systems, it is reasonable to use a single number to approximate the requirements for each limiting resource of each communication implementation on the system. If the node (or router) with the maximum requirements changes between different communication implementations, it is not reasonable to simplify the resource requirements to a single number. Instead, the compiler must track resource requirements on all nodes to ensure peak resource usage over several communication patterns. Section 5.4 addresses extensions to the resource management problem that address multiple limiting resources and multiple representative nodes.

For iWarp and the other systems discussed in the chapter, it is reasonable to assume that resource requirements on a single node or router represents the resource requirements of system-

P_i	ith communication pattern
C_i	Number of times P_i is executed
F_{ij}	Function that implements P_i
R_{ij}	Number of resources required by F_{ij}
V_{ij}	Value of F_{ij} (a function of execution time)

Table 5.1: Summary of the parameters used to define the communication resource problem.

wide, regular communication patterns. On the torus-based iWarp, resource use is relatively even on the nodes, so the resource use on any node is representative of the system as a whole. On an ATM network with a less regular topology, the bandwidth of a single "hot" link represents the limiting case of system-wide communication steps.

5.2 Resource problems

Communication on parallel systems consumes resources from a limited pool. The resources are released after each message in the dynamic resource reservation model, but the resources are reserved between messages in the static resource reservation mode. The specific resources vary between different systems. By abstracting away from the specific communication resource, one can define general resource management problems for a compiler to solve when targeting communication for the connection-based model.

In this section, I make two simplifying assumptions about resource requirements. First, the algorithms in this section approximate the resource requirements of entire communication pattern by the resource requirements of a single node or link. The previous section describes the rationale for this approximation. Second, the algorithms in this section assume that there is a single limiting communication resource. Section 5.4 discusses how the resource management algorithms can be adjusted when these resource model restrictions are loosened.

I begin by defining the compiler's communication model introduced in Chapter 2 more precisely. After communication analysis, the compiler has a list of N communication patterns, $P_1..P_N$ that occur in the program. Each communication pattern P_i is associated with a location in the original program and is executed C_i times. Each communication pattern P_i can be satisfied by one of h alternative implementing functions $F_{i1}..F_{ih}$.

Each function F_{ij} has a cost and a benefit associated with it. The cost is R_{ij}, the number of limiting communication resources needed to implement function F_{ij}. The benefit is V_{ij}, an estimate of the execution time of F_{ij} based on the message size and machine size. Each node on the system has R communication resources available, and all communication resources can be reclaimed by a phase switch that executes in time *Sw*. Table 5.1 summarizes these resource problem parameters.

Given this model, the compiler has two resource problems to solve. The *function packing* problem determines which functions should be used to implement the required communication patterns. This problem attempts to minimize the expected communication time given a limited set of communication resources. The *phase division* problem uses program control flow infor-

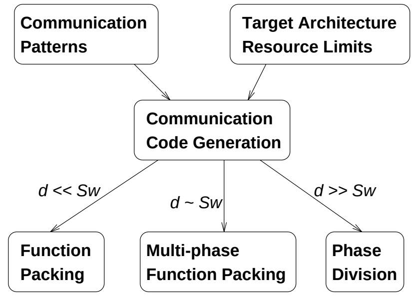


Figure 5.6: Relation between communication resource problems and the cost of phase switch Sw *and average performance difference of different communication implementations* d.

mation to determine how the required communication patterns should be divided into phases to minimize the cost of reclaiming resources assuming an implementation is already selected for each pattern. The compiler can solve both problems simultaneously by selecting the appropriate implementations and phase divisions in the *multi-phase function packing* problem.

The relation between the phase switch time and implementation execution time differences on a particular architecture directs which problems must be solved. Assume the compiler considers two functions that perform in time V and $V + d$ on a machine with a phase switch time of *Sw*. Then the possible execution times, depending on resource availability, are V, $V + d$, and $V + Sw$ [1]. If $d \geq Sw$ there is no reason to ever use the slower function, because the execution time of doing the phase switch still leaves room for improvement using the faster though more expensive function. Therefore, the compiler should only consider functions where $d < Sw$.[2]

The cost of the phase switch on the target system directs the compiler towards the resource allocation problems that must be solved. Figure 5.6 shows how the phase switch time *Sw* and the average performance difference d dictate which problems must be solved. If phase switches are fast, it makes sense to always use the fastest communication function, so only the phase division problem needs to be solved. If phase switches are slow, it makes sense to optimize for the cheapest function in resource usage so only the single phase function packing problem needs to be solved. Considering both the function packing and phase division problems is only an issue if the execution time of a phase switch is about the same as the communication performance differences.

The number of nodes in the target system and the size of the data set also affect which problem should be addressed. The phase switch on most systems will increase as the number of nodes increase, and the difference in implementation times will change as the data set size changes.

[1] Depending on the program structure, two phase switches may be required. In this case, the option $V + Sw$ is replaced by $V + 2 \times Sw$.

[2] Not strictly true, because the phase switch synchronizes the machine before changing the communication state. Avoiding synchronization may be a factor for programs that are inherently asynchronous, but this is beyond the scope of this thesis.

5.2.1 Function packing

The problem of choosing implementations for communication patterns given R resources can be optimally solved by a dynamic programming algorithm[CLR90]. Dynamic programming is a variation of the divide and conquer approach that uses tables to save intermediate results. Each entry is calculated using input information and previously calculated entries. As the table entries are calculated, the algorithm stores which inputs were used for each entry. After the table has been filled, the algorithm traces back from the last entry to find which inputs contributed to the minimal result. Since table entries are calculated exactly once, the computation time is proportional to the size of the table.

The dynamic programming algorithm for the function packing problem minimizes the total communication time for the available communication resources. The algorithm calculates entries in an $N \times R$ table T. Each row is associated with one communication pattern. Entry $T[i,j]$ contains the expected time to execute patterns $P_1..P_i$ given j communication resources, i.e. $\sum_{k=1}^{i} R_{kx} \leq j$ where F_{kx} is the selected function for pattern P_k. If it is not possible to satisfy all patterns in j resources the entry is set to ∞. Entry $T[i,j]$ is calculated from the cost information for various implementations of P_i and values in the previous row.

The first row of the table is initialized as follows:

$$\begin{array}{l|l} T[1,j] = C_1 \times V & \text{V} = Min_{k=1}^{h}(V_{1k}) \text{ such that } R_{1k} \leq j \\ T[1,j] = \infty & R_{ik} > j \text{ for all } 1 \leq k \leq h \end{array}$$

To complete row i, consider P_i and its satisfying functions $F_{i1}..F_{ih}$. Entry $T[i,j]$ is calculated to determine the best function to use with j resources available. If no function fits, the entry is ∞.

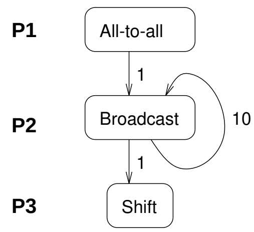
$$T[i,j] = Min_{k=1}^{h}(T[i-1, j-R_{ik}] + C_i \times V_{ik})$$

The algorithm also notes which implementation $F_{i,k}$ is used to calculate $T[i,j]$.

Completing the table takes $O(hNR)$ steps. After completing the table, $T[N,R]$ contains the minimal sum of expected communication times for N patterns in R resources. By tracing back the choices that lead to $T[N,R]$, the algorithm can determine which set of implementing functions to use.

Figure 5.7(a) shows the flow graph of the communication patterns for a simple program. Figure 5.7(b) shows a table of the alternative communication functions with their costs and benefits for each pattern. Each communication pattern can be implemented using the basic message passing system with no more communication resources. This alternative is $F_{i,1}$ for each pattern P_i. The other alternatives use additional communication resources to improve performance.

Figure 5.7(c) shows the table that the dynamic programming algorithm calculates for this example. Each row represents a pattern and each column represents a number of resources. For this example we considered a maximum of R=10. Filling in the row for P_1 is simple. The only function that fits in columns 0 to 7 is $F_{1,1}$. For the remaining columns, $F_{1,2}$ is faster and fits in at least 8 resources. Similarly, filling in columns 0 and 1 for P_2 requires no choices, only $F_{2,1}$ fits. For entry $T[2,2]$ there are two alternatives, $T[1,2] + C_2 \times V_{2,1} = 100 + 10 \times 25 = 350$ and $T[1, 2-2] + C_2 \times V_{2,2} = 100 + 10 \times 15 = 250$. $F_{2,2}$ provides the minimum, so the second option is used. Similarly, calculating the value of $T[2,8]$ requires finding the minimum of three options:



(a)

$P_1, C = 1$

	R	V
$F_{1,1}$	0	100
$F_{1,2}$	8	10

$P_2, C = 10$

	R	V
$F_{2,1}$	0	25
$F_{2,2}$	2	15
$F_{2,3}$	6	10

$P_2, C = 1$

	R	V
$F_{3,1}$	0	30
$F_{3,2}$	3	20

(b)

	0	1	2	3	4	5	6	7	8	9	10
P_1	100	100	100	100	100	100	100	100	10	10	10
P_2	350	350	250	250	250	250	200	200	200	200	160
P_3	380	380	280	280	280	280	230	230	230	230	190

(c)

Figure 5.7: Simple function packing example using three communication patterns. Part (a) shows communication flow from the original program. Table (b) shows the costs and benefits of the alternative functions. Table (c) shows the pattern versus resource availability table that results from the function packing algorithm.

$T[1,8] + C_2 \times V_{2,1} = 260$, $T[1, 8-2] + C_2 \times V_{2,2} = 250$, and $T[1, 8-6] + C_2 \times V_{2,3} = 200$. Entry $T[3,10]$ shows the minimum sum of expected execution times for these three patterns given 10 resources per node. After tracing back, we find that the functions used for this case are $F_{1,2}$, $F_{2,2}$, and $F_{3,1}$.

If P_i and P_j are instances of the same communication pattern, they can use the same communication resources. To be optimal, the execution counts of all the same patterns must be added together, so the resource requirements are properly weighted. Consider a program that performs a row shift communication pattern in two separate statements. Both instances of the row shift communication can use the same communication resources.

If the patterns $P_1..P_N$ are distinct, the patterns can be entered into the table in any order. The solution for each pattern P_i builds on the solutions calculated for the previous patterns $P_1..P_{i-1}$. This problem exhibits an optimal substructure, and indeed the dynamic programming algorithm finds an optimal solution if patterns $P_1..P_N$ are distinct. Therefore, if the original N required patterns can be condensed into a list of M distinct patterns $Q_1..Q_M$, the $Q_1..Q_M$ patterns can be used to find the optimal function packing.

The example in Figure 5.8 shows how optimality is destroyed when instances of the same pattern are not combined. Figure 5.8(a) and (b) shows three patterns where P_1 and P_3 are really instances of the same pattern. The function packing algorithm calculates a minimum expected time of 11 using $F_{1,1}$, $F_{2,2}$, and $F_{3,1}$. Figures 5.8(c) and (d) show the case where the two instances of the same communication pattern P_1 and P_3 are considered together. This instance of the function packing algorithm only considers two patterns $P_1' = P_1 \cup P_3$ and P_2 with execution counts $C_1 = 4$ and $C_2 = 3$. This results in a lower expected time of 10 using

$P_1, C = 2$

	R	V
$F_{1,1}$	1	2
$F_{1,2}$	3	1

$P_2, C = 3$

	R	V
$F_{2,1}$	1	2
$F_{2,2}$	3	1

$P_3, C = 2$

	R	V
$F_{3,1}$	1	2
$F_{3,2}$	3	1

(a)

	1	2	3	4	5
P_1	4	4	2	2	2
P_2	∞	10	10	7	7
P_3	∞	∞	14	14	11

(b)

$P'_1, C = 4$

	R	V
$F_{1,1}$	1	2
$F_{1,2}$	3	1

$P'_2, C = 3$

	R	V
$F_{2,1}$	1	2
$F_{2,2}$	3	1

(c)

	1	2	3	4	5
P'_1	8	8	4	4	4
P'_2	∞	14	14	10	10

(d)

Figure 5.8: Function packing algorithm with a duplicated communication pattern ($P_1 = P_3$). Table (a) shows the costs and benefits for alternative functions for the three patterns. Table (b) shows the values calculated by the function packing algorithm considering P_1 and P_3 separately. Table (c) the costs and benefits with P_1 and P_3 considered together as P'_1. Part (d) shows the results of the function packing algorithm with the combined patterns.

functions $F_{1,2}$ and $F_{2,1}$.

5.2.2 Phase division

The compiler can use a greedy algorithm to divide the communication requirements of a program into phases if an implementing function has already been chosen for each communication pattern. The greedy algorithm works by building a graph out of the pattern information. In the initial graph, each node represents a pattern instance. Two nodes are connected by an edge if the program flow control indicates that patterns can execute one after the other (i.e. they can be temporally adjacent). Each edge is weighted by the number of times the program moves from one pattern to the other.

For example, consider the pseudo-code in Figure 5.9(a) that contains four communication patterns. These four communication patterns can be represented by the graph in Figure 5.9(b) where the resource costs of each pattern are listed to the left of each node.

By contracting edges, the greedy algorithm can transform the initial graph into a new graph where each node represents the set of communication patterns in a communication phase, and the edges represent necessary phase switches. Figure 5.10 shows the outline of this algorithm. The algorithm merges nodes if the communication patterns contained in both nodes can be implemented in the set of available resources R. The algorithm attempts to merge nodes connected by the edges with the largest weights first. This edge ordering biases phase divisions out of loop nests.

Assume the initial graph of communication flow has N nodes and E edges, and assume there are M unique communication patterns ($M \leq N$). The cost of sorting the edges is

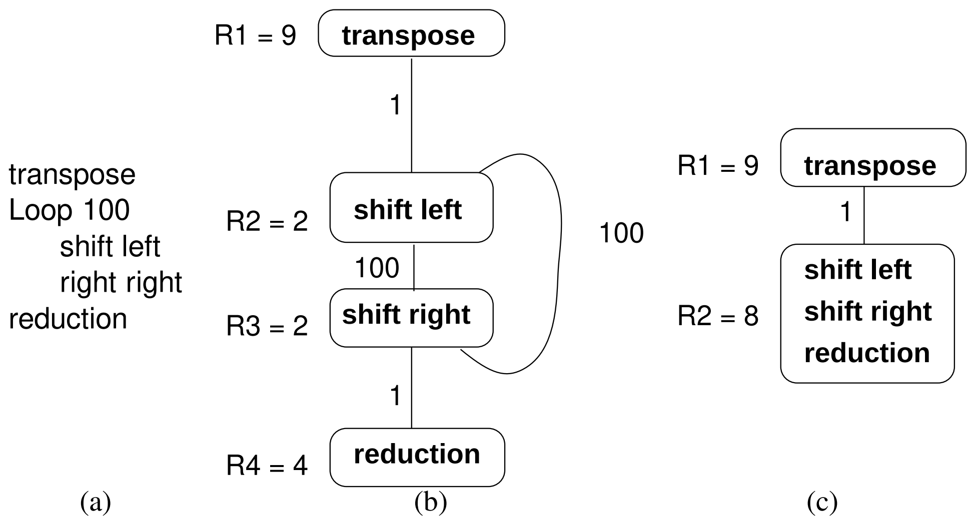


Figure 5.9: (a) Pseudo communication code. Initial (b) and final (c) graphs for the greedy algorithm example.

GreedyDivision(*edgelist*) $\rightarrow$

  Sort *edgelist* from high to low value

  for each *e* **in** *edgelist* **do**

    EdgeContract(e, R)

  endfor

EdgeContract(e,R) $\rightarrow$

  if ResourceUsage(head(e) $\cup$ tail(e)) $\leq R$ **then**

    tail(e) $\leftarrow$ head(e) $\cup$ tail(e)

    head(e) $\leftarrow$ tail(e)

  endif

Figure 5.10: Pseudo-code for the greedy phase division algorithm.

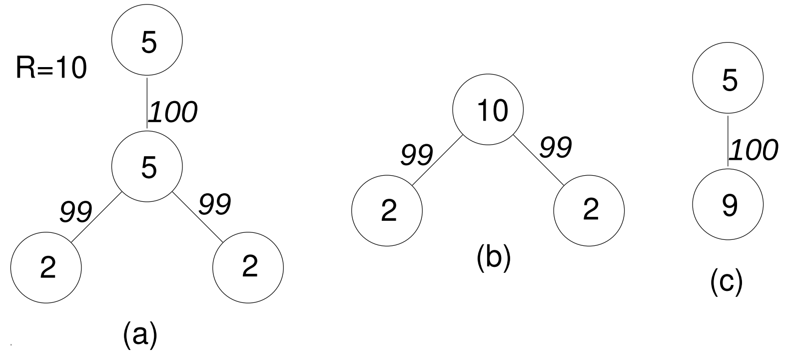


Figure 5.11: Example of non-optimal greedy edge contraction. (a) shows the initial graph with edges labeled by weight and nodes labeled by resource requirements. Given 10 resources (b) shows the phases that result from contracting the heaviest node first. (c) shows the phases that result from contracting the lighter nodes first.

$O(E \log E)$. If each node contains a length M bit-vector that indicates whether a particular communication pattern is contained in the node, then calculating the resource usage of a union of two nodes is an $O(M)$ operation. Therefore, the cost of the entire phase division algorithm is $O(E \log E) + O(EM)$. If the class of input graphs is not restricted, E is $O(N^2)$, but graphs derived from well-structured programs tend to have a relatively low degree, so E is better approximated by $O(N)$. Scientific programs are generally well-structured, so this algorithm operates in $O(MN)$ time for scientific programs.

By applying this greedy algorithm given $R = 10$ total resources to the example in Figure 5.9, the graph in Figure 5.9(c) remains. In the first step, the heaviest edges between the shifting patterns are contracted. Since the edges are undirected, both edges between the shifting nodes are contracted in the same step. Then the only other edge that can be successfully contracted is the one to the reduction pattern. Therefore, two phases remain, one for the transpose and another for all the other communication patterns, so only one phase switch is needed.

Contracting the most expensive arc first is a reasonable heuristic to keep phase switches out of deeply nested loops, but since both the edges and the nodes have values, this greedy algorithm does not always produce an optimal solution. Figure 5.11 shows a case where contracting two edges with smaller weights yields a better division than contracting the heavier arc first.

While the algorithm is not optimal for general graphs, we can make stronger statements about the graphs that result from the scientific programs considered in this thesis. The control flow in these data parallel programs primarily results from loops. In the arguments that follow, we consider communication pattern graphs that result from programs with loops and **where** statements for control flow. For this style of data parallel program, **if** statements tend to be used for error checking (where communication along the alternative branch does not matter) or for higher level control flow (e.g. checking whether the iterative solution is within the error bounds).

By looking at the most deeply nested loops first (i.e. contracting the most heavily weighted edges), the greedy algorithm can be divided into a series of steps of assigning phases to the

LoopDivision(*looplist*) $\rightarrow$
Sort loop nests in *looplist* from high to low iteration counts
for each *loop* **in** *looplist* **do**
phaseset $\leftarrow$ BodyDivision(*loop*)
Replace *loop* in enclosing loops with FirstPhase(*phaseset*)
and LastPhase(*phaseset*)
endfor

Figure 5.12: Pseudo-code for the per loop phase division algorithm.

patterns in the most deeply nested loop not assigned so far. The patterns in this loop can be viewed as weighted nodes connected by a ring of uniformally weighted edges. The previously contracted loops nested in this loop can be modeled as one or two nodes. Given an algorithm for assigning phase divisions within one loop body, the pseudo-code for this algorithm is shown in Figure 5.12.

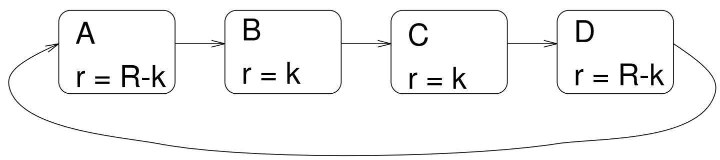
The prototype compiler also uses the naive greedy approach over the loops. The simple greedy method is not optimal over evenly weighted arcs that result from loops. For loops of less than four nodes, the greedy approach will find the optimal phase division. Since all nodes are neighbors, the order of edge contraction is unimportant.

For these larger loops, the order of edge contraction does matter. In this case, For the greedy approach can introduce at least $\lfloor n/4 \rfloor$ extra phases over a loop of n nodes. By contracting one edge incident on a node, we may prevent the other edge incident on that node from begin contracted. Consider a loop of four nodes: A, B, C, D, where each edge can be contracted, except the arc from D to A (see example in Figure 5.13). Depending on which edge incident to C is contracted, either two phases or three phases are required. By repeating this pattern, we can construct cases, where the order of arc contraction can vary the number of phases by $\lfloor n/4 \rfloor$. While there are cases where naive arc contraction can be costly, the number of communication patterns in each loop of the programs we encountered were relatively small. Section 5.3 discusses the performance of the greedy phase division algorithm.

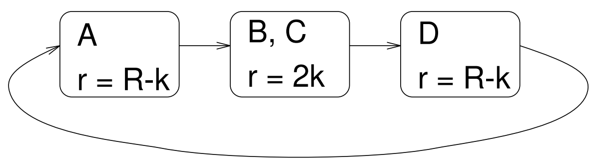
Lemma 1 *Given a graph $G(N, E)$ of n weighted nodes connected by a ring of uniformally weighted edges, a new graph $G'(N', E')$ that minimizes $|N'|$ while bounding the weights of nodes in N' to R can be formed by contracting edges of G in $O(n^4)$ steps.*

Dynamic programming can optimally solve this flat phase division problem in $O(n^4)$ steps. Given a ring of n nodes, we can fill in $n(2n - 1)$ entries such that entry $T[i, j]$ contains the set of optimal phase divisions for patterns P_j to P_{j+i}. The last row of the table contains phase divisions for all possible sequences of the ring.

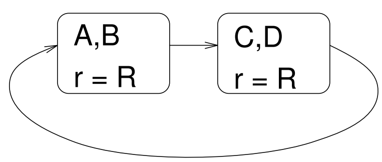
There may be several valid ways to divide a sequence into an equivalent number of phases. For example, consider a sequence of four nodes with the following weights: $R/2$, $R/3$, $R/3$, $R/3$. Given a bound of R, there are two divisions into two phases: ($R/2$, $R/3$)-($R/3$, $R/3$) and ($R/2$)-($R/3$, $R/3$, $R/3$). For the purposes of filling in the dynamic programming table, only



(a)



(b)



(c)

Figure 5.13: Example four node loop (a) where order of edge contraction affects final number of phases. When the edge from C to B is contracted first (b), three phases are required. When the edge from C to D is contracted first (c), two phases are required. R is the maximum number of resource allowed per phase, and $k < R/2$.

```
mindiv ← i+1
For each g in T[i-1,j+1] do
    g ← Contract(Edge(P_j, FirstPhase(g)), R)
    if DivisionCount(g) < mindiv then
        savediv ← { g }
        mindiv ← DivisionCount(g)
    else if DivisionCount(g) = mindiv then
        savediv ← savediv ∪ g
    endif
endfor
For each g in T[i-1,j] do
    g ← Contract(Edge(LastPhase(g), P_{j+i}), R)
    if DivisionCount(g) < mindiv then
        savediv ← { g }
        mindiv ← DivisionCount(g)
    else if DivisionCount(g) = mindiv then
        savediv← savediv ∪ g
    endif
endfor
T[i,j] ← FirstLastUnique(savediv)
```

Figure 5.14: Pseudo-code to calculate the set of minimal phase divisions for patterns P_j to P_{j+i}.

differences in the first and last phases of the sequence are important. For example, given eight nodes A to H, both of the following divisions are equivalent for passing onto later rows: (A,B)-(C,D,E)-(F,G)-(H) and (A,B)-(C,D,E,F)-(G)-(H). To check all phase division possibilities, the dynamic programming algorithm must pass on all minimal phase divisions that differ in the first and last phases, which is bounded by $O(d^2) < O(n^2)$ given d phases.[3]

The first row of the table contains the resource requirements of each communication pattern. Each subsequent row relies on two entries from the previous row. To calculate the phase divisions for patterns P_j to P_{j+i} (T[i,j]), we examine the phase divisions for patterns P_j to P_{j+i-1} (T[i-1,j]) and P_{j+1} to P_{j+i} (T[i-1,j+1]). To fill in entry T[i,j], the algorithm examines all i pattern sequences that can be constructed from contracting edges from P_j to each possible division in T[i-1,j+1] and from contracting edges from P_{j+i} to each possible division in T[i-1,j]. The algorithm determines the minimum number of phases required and saves all minimum count phase divisions that differ in the nodes required in the first and last phase divisions. Figure 5.14 shows this calculation in more detail. Since there are $O(n^2)$ table entries and $O(n^2)$ potential phase divisions to check for each entry, the time to fill out the table is bounded by $O(n^4)$ steps.

The order of contracting edges in a n node sequence affects the number of phases in the final

[3]More precisely at most $(\sum_{i=1}^{d} \sum_{j=1}^{min(d,n-d+2-i)} 1)$ phase divisions of d phases are possible that differ only in the first and last phases over n nodes.

sequence. This algorithm finds the optimal phase divisions for a sequence of nodes, because it examines all phases that result from contracting edges in all feasible combinations.

In step one, filling entries T[1,j] considers the cases where each edge is contracted first. The algorithm tries to merge all adjacent pairs of nodes.

Inductive claim: Each subsequent row T[i,j] attempts to contract all feasible edges in step i.

Assume the claim holds for rows $i < k$. Consider row k. For each entry T[k,j], the algorithm uses entries T[k-1,j] and T[k-1,j+1]. By the inductive claim, these entries were calculated by trying all combinations of contracting the edges in steps 1 through $k-1$. By using the algorithm in Figure 5.14, the algorithm attempts to contract all edges in step k.

For each row k, $n - k$ entries are calculated. To calculate each entry, the edges at the beginning of the sequence and at the end of the sequence are contracted, i.e. edges (1,2) through (n-k,n-k+1) and edges (n-1,n) through (k-1,k) for the entire row. For $k \leq n/2$, every edge is examined. For $k > n/2$, some edges are not examined because it is not feasible to contract these nodes after step k. For example, consider edge $n/2-1$. It does not make sense to contract this node after step $n/2$. The other edges in the sequence are divided by edge $n/2$, and these divided edges can be contracted in parallel without affecting the other side. If edge $n/2 - 1$ is to be contracted last, it does not need to wait until after step $n/2$. In general, it does not make sense to delay contracting an edge j after step $Max(j, n-j)$, by a similar argument that edges in the sequences divided by j can be contracted in parallel.

By filling in the table, the algorithm considers contracting the edges in all feasible orders. Therefore, the algorithm examines all potentially minimal phase divisions, and T[n,1] contains minimal phase divisions for the original sequence. To handle rings, we repeat the original sequence, so all combinations of edge contractions over the ring are considered.

Phase division decisions made in the inner loops may affect the number of phase divisions that are required in the outer loops. However, assuming the number of iteration counts required in the inner loops is substantially more than the number of iteration counts required in the outer loops, concentrating on the inner loops first results in the best global phase division.

Lemma 2 *Given two nested loops, where the inner loop is executed $x \cdot y$ times and the outer loops is executed y times, the outer loop must eliminate x phases for every one phase added to the the inner loop.*

Suppose the optimal phase division of the inner loop requires d phases. By adjusting the inner loop division to $d + 1$ phases, we may be able to better divide the phases of the outer loop. The extra phase division in the inner loop results in $x \cdot y$ extra phase switches. We must eliminate x phases in the outer loop to match the cost of the extra $x \cdot y$ phase switches. Therefore, unless the inner loop count x is very small, biasing phase division to inner loops first is the more practical approach.

5.2.3 Function packing in multiple phases

The algorithm described in Section 5.2.1 picks implementations for communication patterns that minimize total communication time given a limited number of communication resources in a single phase. The algorithm in Section 5.2.2 tries to minimize communication time by dividing the program into phases assuming each communication pattern already has a fixed implementation.

The dynamic programming algorithm in Section 5.2.1 can be augmented to both pick implementations and divide the program into phases, but the results are no longer provably optimal because instances of the same communication patterns must be considered separately. Phases only make sense for communication patterns that occur in the same portion of the program. Therefore, it is not reasonable to replace the list of program patterns P with the unified list of program patterns Q, because the unified patterns do not preserve this temporal locality.

By looking at the original pattern list and changing the table updating rules, the function packing dynamic programming algorithm can be adjusted to insert phase switches where locally beneficial. Since the communication patterns are not distinct, the order in which the patterns are added to the table will affect the final result. We use the naive ordering of patterns based on the statement numbering.

Since communication resources can be reused for two instances of the same function in the same phase, each table entry must keep track of the set of functions used to calculate that value, so the resource cost of a function is only charged once. These functions are stored in the set $T[i,j].L$. Then $T[i,j]$ is the minimum over all function alternatives of:

$$\begin{array}{l|l} \text{T[i-1,j]} + C_i \times V_{ik} & \text{if } F_{ik} \in T[i-1,j].L \\ \text{T[i-1,j-}R_{ik}\text{]} + C_i \times V_{ik} & \text{if } R_{ik} \leq j \\ \text{T[i-1,R]} + C_i \times V_{ik} + G(P_{i-1}, P_i) \times Sw & \end{array}$$

The first rule is used if a function to implement P_i has been previously selected (and stored in the set T[i-1,j].L). No additional resources are consumed, and T[i,j].L is set to T[i-1,j].L. The second rule is used if the implementing function has not been previously selected. In this case the function set T[i,j].L is assigned T[i-1,j-R_{ik}].L $\cup\{F_{ik}\}$. The last rule is used if a new phase is started for the implementing function. Since this rule reclaims all resources, the algorithm uses the most optimistic value for implementing the previous $i-1$ patterns, T[$i-1,R$]. In this case, the function set is reset to $\{F_{ik}\}$. Function G calculates how many times the program execution moves from P_{i-1} to P_i. This number is calculated by looking at the loop structure of the program and finding the least common loop that includes both P_{i-1} and P_i.

5.3 Evaluation of resource management

In this section, we evaluate the effects of two of the communication resource algorithms defined in Section 5.2 on two example programs: a least squares program and an image analysis program. Both programs are written in Fx and are executed on an 8×8 iWarp system.

The limiting communication resource on iWarp is the logical channel. There are 20 logical channels per node, but 5 are used by the runtime system. For the two example programs, all statically reserved implementations cannot simultaneously fit in 15 logical channels, so another 5 channels must be used for the dynamic message passing system leaving 10 logical channels for static connection-based implementations.

To develop a good cost model of communication on iWarp, I measured a variety of implementations of a set a frequently occurring communication patterns over a range of message sizes. From these measurements, I derived a quadratic execution time model for the different implementations of these communication patterns based on the message size. The resource management algorithms use these simple cost estimates to calculate the benefits of each implementation (V_{ij}).

Figures 5.15 and 5.16 show the main communication steps of both programs and the iWarp communication resource requirements for each connection-based implementation. Each pattern can also be implemented by a dynamic implementation with no additional communication resources. iWarp uses the deposit message passing implementation described in Section 2.3.3. This implementation is optimized for compiler-generated communication. Only the resource reservation and routing control differ between the static and dynamic implementations on iWarp.

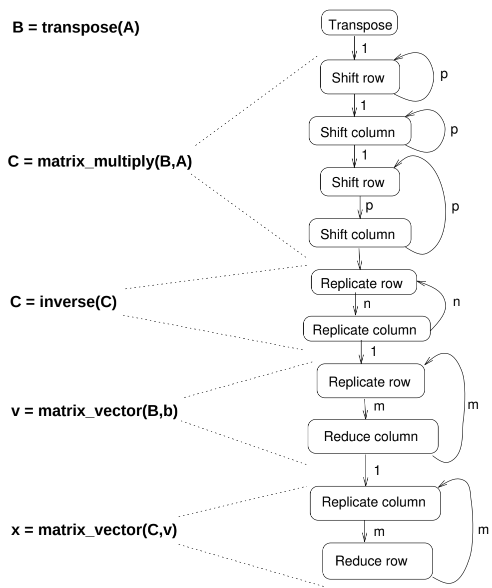
The least squares problem uses a series of simple matrix operations to solve an over-determined system of equations. Given a $n \times m$ linear system of $A \cdot x = b$, the least squares solution computes $x = (A^T \cdot A)^{-1} \cdot (A^T \cdot b)$. Figure 5.15 shows the basic operations of a least squares program and the corresponding communication patterns. In this example, the initial matrix A is distributed over two dimensions, and the vector b is distributed over one row. A transpose over two processor dimensions is needed to find A^T. The program uses a blocked matrix multiplication algorithm which requires nearest neighbor shifts. Gauss-Jordan elimination is used to calculate the matrix inverse. The elimination requires replicating the pivot row and column. Finally two matrix vector multiplications are performed to calculate x. Each matrix vector multiplication requires a replication over one dimension and a reduction over the other dimension. I evaluated this program executing on $n \times m$ input arrays where $m = n/2$.

The image analysis program operates over a $n \times n$ array of complex numbers. The main data array is distributed by rows treating the node array as a ring. The image analysis program is a three step program. The first step calculates a smoothing convolution. The second step performs a two dimensional fast Fourier transform (FFT) by calculating one dimensional FFTs over the rows, redistributing the array, and then calculating FFTs over the columns. Redistributing from the row distribution to the column distribution requires an all-to-all communication pattern. Finally, a sum is calculated over the entire array to determine the upper and lower bounds for valid data. These steps are iterated k times over k different input images.

5.3.1 Function packing evaluation

Figure 5.17 shows the absolute communication times that result from varying the communication pattern implementations for the least squares and the image analysis programs. Figure 5.18 shows the normalized communication and computation times for both programs. In all graphs, the line labeled All connections shows the theoretical communication performance assuming the fastest implementation can always be used. At the other extreme, the line labeled No connections shows the performance using only the default dynamic implementations (using no additional communication resources). The line labeled Algorithm's assignment shows the case where the function packing algorithm described in Section 5.2.1 picks the communication pattern implementations given R=10 resources[4].

[4]An iWarp node has 20 logical channels. Five are used by the runtime system, and five are used by the message passing system, so 10 remain for connection-based scheduling.

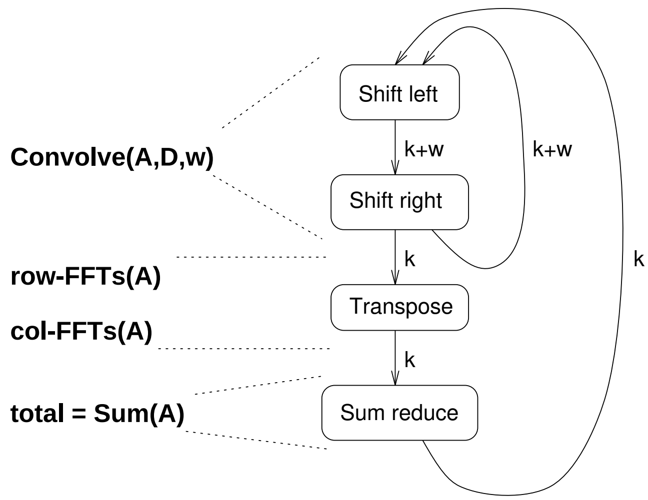


(a)

Pattern	Resources/ node
Transpose	9
Shift row	2
Shift col	2
Replicate row	4
Replicate col	4
Reduce row	4
Reduce col	4

(b)

Figure 5.15: Part (a) outlines the steps and communication patterns used in evaluated the least squares program. The arrays are distributed over two dimensions. Part (b) shows the number of communication resources required per node for the best connection-based implementation for each pattern on an 8×8 *iWarp.*



(a)

Pattern	Resources/node
Shift left	3
Shift right	3
Transpose	8
Sum reduce	8

(b)

Figure 5.16: Part (a) outlines the steps and communication patterns used in the image analysis algorithm. The arrays are distributed over one dimension. Part (b) shows the number of communication resources required per node for the best connection-based implementation for each pattern on an 8×8 *iWarp.*

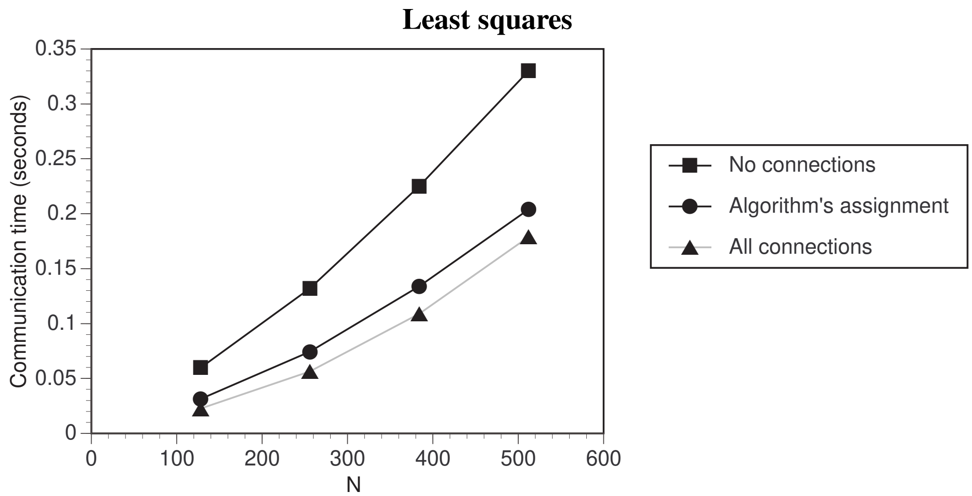
For the least squares program, the function packing algorithm picks the connection-based implementations of the row and column replications and the row shift. The remaining patterns use the default best dynamic implementation.

For the image analysis program, the function packing algorithm picks the connection-based implementation of the transpose, and the shift and reduction patterns use the default dynamic implementations.

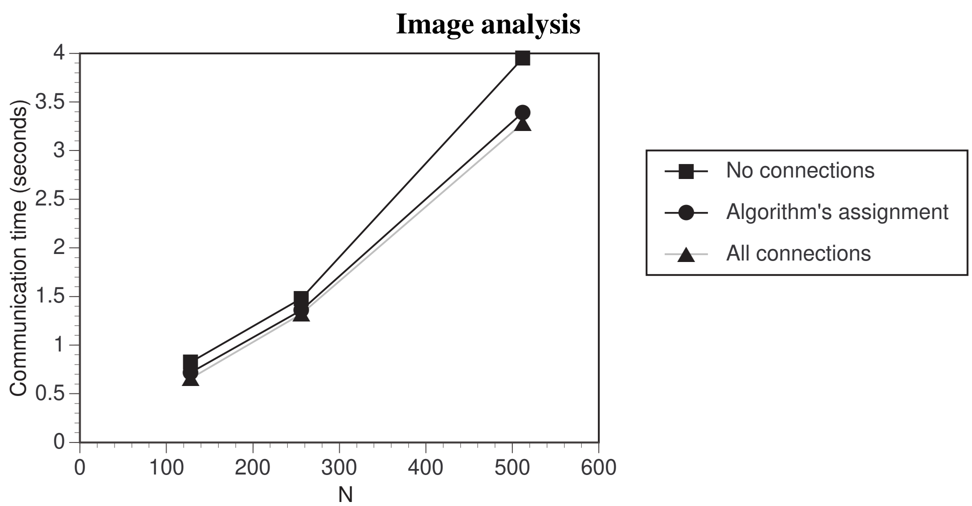
Table 5.2 compares the total execution time of the function packing choices with the all connection-based and all dynamic extremes. These numbers show that the function packing algorithm choices are reasonably close to the all connection-based ideal and measurably better than the dynamic default.

5.3.2 Phase division evaluation

The best way to perform a phase switch on iWarp is to synchronize the system and locally copy in the new communication state. With this approach, the main cost of the phase switch is the cost of synchronization. The best way to perform a barrier synchronization on iWarp involves dedicating logical channels to the synchronization routine. Using this hardware-based barrier synchronization, the phase switch time (*Sw*) on an 8×8 system is no more than 2000 cycles or 100 microseconds. This is substantially less than the generally observed performance differences of different pattern implementations. Therefore, by the argument in Section 5.2, phase division is the problem that needs to be solved for an 8×8 iWarp system that uses a hardware-based barrier.

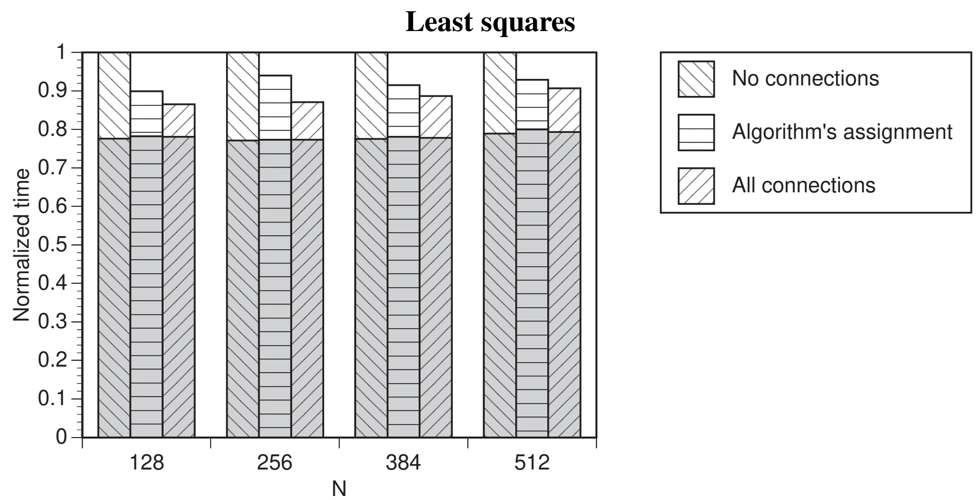


(a)

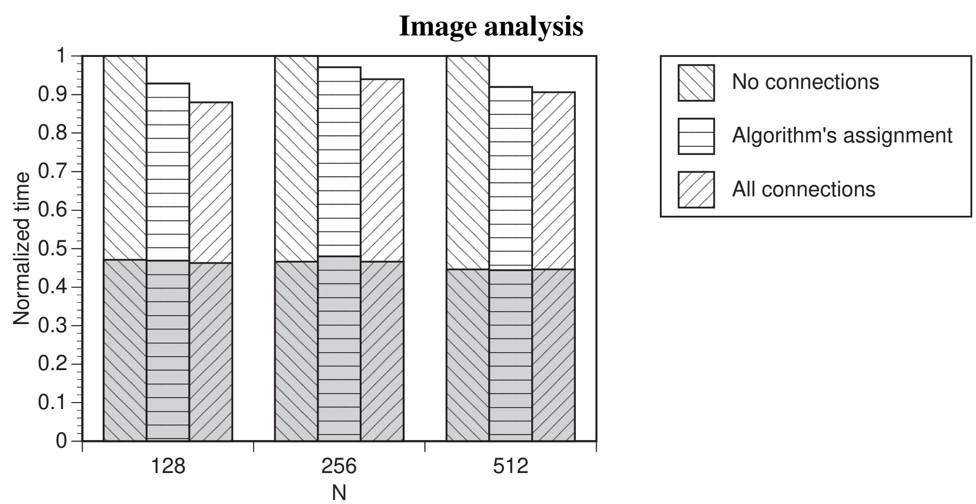


(b)

Figure 5.17: Graph of communication time for different implementation choices of the least squares program (a) and the image analysis program (b). The line label Algorithm's assignment *shows the performance of the programs that use the function selected by the function packing algorithm assuming $R = 10$ resources are available.*



(a)



(b)

Figure 5.18: Graph of normalized execution time for different implementation choices of the least squares program (a) and the image analysis program (b). The darker regions show computation time and the lighter regions show communication time. The bar labeled Algorithm’s assignment *shows the performance of the programs that use the function selected by the function packing algorithm assuming $R = 10$ resources are available.*

Least squares

Problem size	All connections	No connections
128	-3.7 %	10.0 %
256	-3.3 %	9.9 %
384	-3.0 %	9.0 %
512	-2.7 %	7.0 %

(a)

Image analysis

Problem size	All connections	No connections
128	-4.8 %	7.1 %
256	-3.0 %	3.0 %
512	-1.5 %	8.0 %

(b)

Table 5.2: Relative changes in total execution time of the least squares program (a) and the image analysis program (b) using the communication choices of the function packing algorithm. Entries are the percentage changes in total execution time compared to using all connection-based implementations (assuming we had enough resources) and all dynamic implementations (using no statically reserved communication resources).

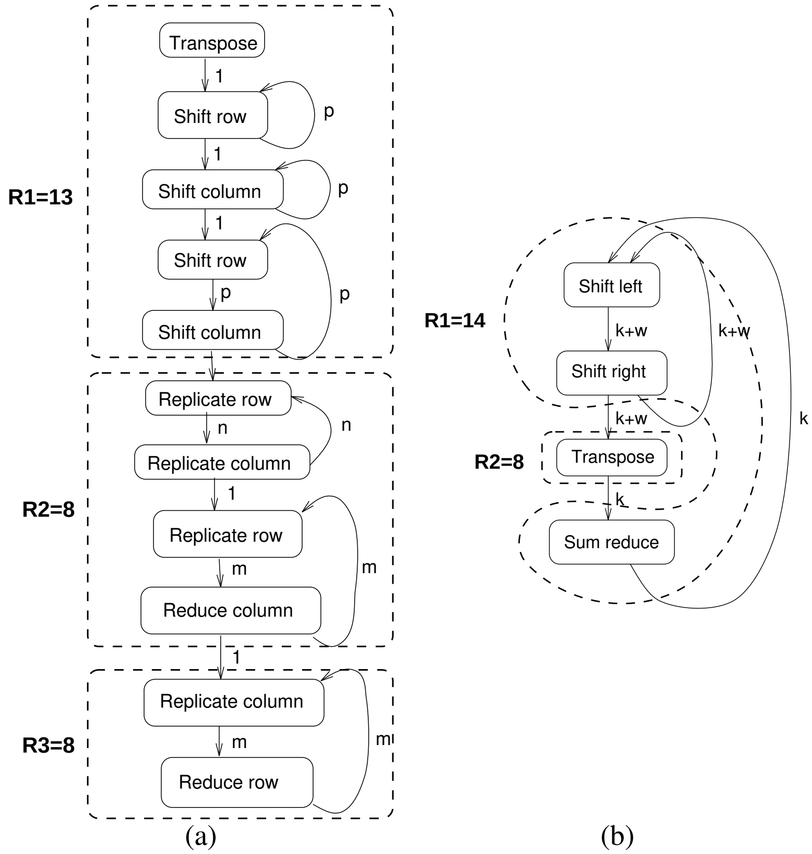


Figure 5.19: The dashed boxes enclose phases chosen by the greedy phase division algorithm assuming $R = 15$ resources are available for the least squares program (a) and the image analysis program (b). Each phase is labeled with the number of resources it requires.

Figure 5.19 shows the phase divisions of the example programs that result from the greedy algorithm described in Section 5.2.2. Since only connection-based implementations are used, no resources need to be set aside for the message passing system, so $R = 15$ resources are available in each phase.

The phase divisions for the least squares program result in two phase switches taking only 200 microseconds. The image analysis program requires two phase switches per iteration adding 3% to the communication time for $n = 128$ and less than 1% when $n = 512$. In both cases, the communication performance is very close to the All connections option in Figure 5.17.

While phase switches are relatively fast on iWarp, there are measurable performance penalties to the simple-minded approach of making each pattern its own phase and performing a phase switch between each pair of patterns. In the least squares program, this simple-minded phase division adds 70% to the communication time for $n = 128$ and 53% for $n = 512$. In the image analysis program, two phase switches are already performed in each iteration, so the

Least squares

Problem size	Greedy algorithm	Naive
128	-0%	-18.7%
256	-0%	-17.2%
384	-0%	-14.9%
512	-0%	-12.7%

(a)

Image analysis

Problem size	Greedy algorithm	Naive
128	-0.7%	-1.8%
256	-0.4%	-1.0%
512	-0.2%	-0.4%

(b)

Table 5.3: Effect of phase division selection on total execution time for least squares (a) and image analysis (b). Entries are percentage change in total execution time from the theoretical optimum of no phase divisions compared to the greedy algorithm phase division (Greedy algorithm) and the naive approach of adding phase switches between all pairs of patterns (Naive).

additional cost of adding a phase switch after every pattern is not so severe. Table 5.3 compares the effects of the greedy and naive phase divisions on the total execution time for both programs against the ideal case of no phase switches.

These measurements show that the simple phase division algorithm works reasonably well. The cost of switching given the phases selected by the phase division algorithm is much lower than the naive approach. A good phase division is even more important for machines without hardware support for barrier synchronization.

5.4 Extending the resource model

The resource management algorithms described in Section 5.2 make two simplifying assumptions about the communication resource model: there is a single limiting resource and the resource requirements of a single node or link characterize the resource requirements of the entire system. Both of these restrictions can be loosened by extending the resource management algorithms in similar ways.

While iWarp has only one real limiting communication resource, other systems may have a number of potentially limiting resources. For example, communication on ATM networks may be limited by both physical bandwidth and hardware connection table entries.

Assume the target system has d limiting communication resources bound by $R_1, ...R_d$. The function packing algorithm can be extended by filling in a $d + 1$ dimensional table instead of a two dimensional table, where each resource is assigned a separate axis. This extended function

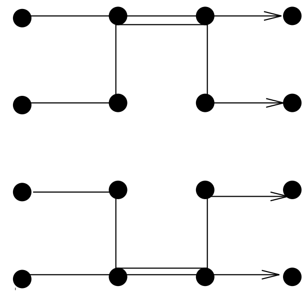


Figure 5.20: An alternative routing for a shift from the first column to the last column that avoids hot spots in the center of the mesh.

packing algorithm takes $O(hN \prod_{i=1}^{d} R_i)$ steps, where h is the number of alternative functions and N is the number of unique communication patterns.

The greedy phase division algorithm can also be extended to consider multiple limiting resources. Each alternative implementation has d numbers that describe the resource requirements for that implementation. When merging a node, d numbers must be checked per pattern instead of one. Therefore, the extended phase division algorithm requires $O(dMN)$ steps, where N is the number of communication patterns and M is the number of unique communication patterns.

Assuming there is a single hot node or link that approximates the resource requirements of the entire system may also be a false assumption on some systems. The hot node may vary between different patterns, or some communication implementations may be clever and route around the hot spots in the system. For example, consider the alternative routes in Figure 5.20 that route around hot nodes in the center of the mesh to send data from the first column to the last column.

We can augment the function packing and phase division algorithms to understand multiple hot links the same way we augmented them to consider multiple limiting resources. If there are d nodes that can have the maximum resource load for all common patterns, then each communication implementation must have the resource requirements on all d nodes defined. The function packing algorithm can still be augmented with extra dimensions in the table, and the phase division algorithm can use more information when merging the nodes. At the extreme, the system may need information about resource requirements for every node (or link) in the system. In this case, function packing takes $O(hNR^P)$ steps and phase division takes $O(PMN)$ steps where P is the number of nodes (or links).

5.5 Chapter summary

This chapter described examples of limiting communication resources that exist on several systems. By abstracting away the details of the specific resources, I revealed several simple problems that the compiler can solve to trade between communication resource use and communication performance, and I described three simple algorithms that solve these resource management problems.

My measurements on iWarp show that these simple algorithms yield reasonable results for real programs. The benefits of intelligent phase division should be more apparent on machines like Paragon without hardware support for barrier synchronization. Barrier synchronization on a 64 node Paragon system takes an order of magnitude longer than synchronization on the same number of iWarp nodes.

However, the resource pressure is not nearly as severe for software resources in the connection-based model on Paragon, so phase division is not an important problem on Paragon. Instead, eliminating barrier synchronization is important for the dynamic, deposit message passing model. The next chapter introduces an algorithm that determines when explicit synchronization can be eliminated from the dynamic deposit model.

Chapter 6

Synchronization elimination in the deposit message passing model

The previous two chapters concentrated on algorithms and data structures required to generate communication code following the static resource reservation model. Communication generated according to the dynamic resource reservation model can also benefit from information about the communication patterns.

Our experience with the Fx compiler has shown that the deposit model is an efficient example of the dynamic resource reservation model for compiler-directed communication generation[SSO+95]. Chapter 2.3.1 describes how the deposit model requires synchronization between the sending and receiving nodes to ensure safe communication. In this chapter, we propose a method using only communication pattern and program knowledge to reduce the amount of additional synchronization required. This method has been implemented in the prototype compiler (shown as phase Dynamic control elimination in Figure 1.2), and we evaluate the benefits of eliminating the redundant synchronization.

6.1 Synchronization requirements in parallel systems

Synchronization is needed for programs on parallel systems when two processes (or threads) are operating on the same data location to ensure that the reader is using the proper value. Figure 6.1(a) shows a writer and reader relationship under the shared memory model. Thread one writes a location, and thread two reads that location twice. On the first read, it wants the value before the write, and on the second read, it wants the value after the write. To ensure that the reading constraints are maintained, a synchronization point is required after the first read and before the write.

The remaining figures show the same situation under the distributed memory model using different communication models. Figure 6.1(b) shows two threads in the traditional message passing model. By buffering messages that arrive too early, this model ties thread synchronization with data transfer. By using hardware or software control messages, the connection-based model also ties data transfer with synchronization. Since the sending thread does not directly modify a memory location on the receiving thread, no additional synchronization is required.

Figure 6.1(c) shows two threads using the deposit message passing model described in

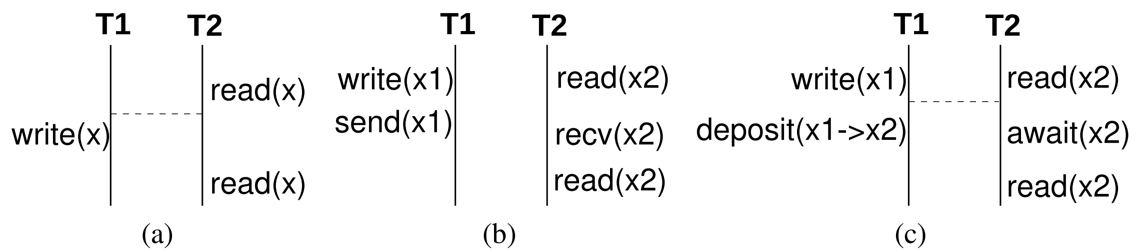


Figure 6.1: Example of synchronization requirements in three memory/communication models (a) shared memory, (b) explicit message passing, and (c) deposit message passing. The extra synchronization points are shown by the dashed line.

Section 2.3.1. Each processing element has a private address space, but the processing element must directly write (i.e. deposit) into a remote address space. After the message is deposited, a semaphore is updated on the remote node to notify the receiver that the new value has appeared. Since the source thread directly changes values on the destination thread without direct action by the receiving thread, a synchronization point is required after the first read and before the deposit.

This chapter addresses synchronization in compiled data parallel programs that use the deposit model for data transfer. The issues of synchronization analysis have also been addressed for other memory models and program models. Working in the shared memory model, Subhlok addresses the problem of synchronization in Fortran 77 with parallel do and case statements[Sub90]. He developed a data flow algorithm that determines whether the current synchronization points in the program are sufficient to ensure that there are no data race conditions. This algorithm can be used to optimize the number of required synchronization points by iteratively removing synchronization points and re-running the data flow algorithm. By concentrating on the data parallel computation model, I address a simpler problem domain, so I am able to develop a more powerful synchronization elimination algorithm.

Tseng addresses barrier elimination for parallel programs compiled for a shared-memory system using the SUIF compiler[Tse95]. Unlike traditional shared memory compilers, this compiler performs analysis to determine how data and computation should be distributed over the machine. Tseng developed a phase that analyzes the pattern of local and non-local memory accesses to determine when barriers are unnecessary or when barriers can be replaced by cheaper synchronization methods. In the deposit message passing model (unlike the shared memory model), the receiving node knows when data has been updated, so my algorithm can be more aggressive about using communication analysis to eliminate barriers.

In [DZO92], Dietz, Zaafrani, and O'Keefe propose a static scheduling approach to eliminate the need for synchronization in some cases. However, their approach relies on a careful timing analysis of the system and the program, and such timing accuracy is not reasonable on most of today's parallel systems because of variations at runtime due to cache misses, interrupt handlers, and network congestion.

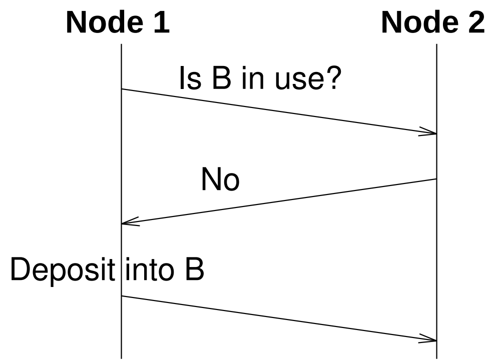


Figure 6.2: In general, nodes 1 and 2 must negotiate whether the target buffer is available before node 1 can safely deposit onto node 2.

6.2 Synchronization requirements of the deposit model

The data parallel compiler generates computation organized into global "steps". Within each step, the compiler generates code that sends from and receives into disjoint buffer segments, so the code within a communication step is guaranteed to access the communication buffers correctly. Between steps additional communication may be necessary to ensure that target buffers are ready to be overwritten. This synchronizing communication can be in the form of explicit control messages or combined control messages such as barrier synchronization or combining trees. Figure 6.2 shows the logical exchange required between nodes 1 and 2 to guarantee that it is safe for node 1 to deposit data into buffer B on node 2. The Fx compiler inserts barriers between communication steps to insure that this synchronization is maintained.

Depending on the sequence of communication patterns, this additional control information can be implicitly piggy-backed on communication exchanges of previous steps. For example, consider the successive over-relaxation (SOR) problem introduced in Chapter 4. In the parallel implementation shown in Figure 4.1, both `A` and `B` are distributed over a ring of nodes. Each node owns a subregion of each array, and each node shares *overlap* regions with its neighbors. In the overlap region, the nodes store duplicated data that is owned by the neighboring node but is needed locally for the next computation step.

In the overlap shift, each node communicates with its two neighboring nodes, e.g. node P exchanges data with nodes $P+1$ and $P-1$. Before statement 2a, node P has communicated with its neighbors in statement 1a, so it knows that nodes $P-1$ and $P+1$ have at least started executing statement 1. Node P knows that its local portion of `B` is up to date and safe to send. Since statement 1b does not access the overlap region of `B` and $P-1$ and $P+1$ have at least reached statement 1a, it is safe for processor P to deposit its new values for `B`. Therefore, the second overlap shift can be executed without any additional control messages or synchronization. By a similar argument, the first overlap shift does not require additional synchronization either. In both cases, each node can use messages from previous communication steps to know how far the neighboring nodes have progressed in the program.

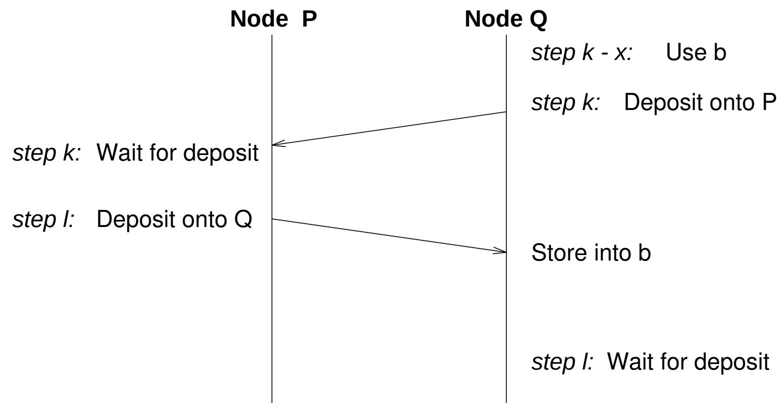


Figure 6.3: Timeline of an example data exchange between nodes P *and* Q. Q *does not use* b *between steps* P *can deposit into* b *with no additional control messages.*

6.3 Synchronization elimination algorithm

The ad hoc argument for the SOR example can be generalized to find unnecessary control messages in any data parallel program. This section formalizes the conditions when previous communication steps can implicitly carry control information and describes a data flow algorithm that finds where these conditions are met in the program

Figure 6.3 shows an example data exchange. At step l of the program, assume node P must send data to buffer b on node Q. To ensure that Q is no longer using the current data in b, node P must know that Q has also reached step l or some previous step k and no longer requires the old data. If Q has sent a message to P in step k and does not use buffer b until step l, then node P knows that buffer b is no longer being used and can send the new data without any additional negotiation.

In the SOR example, node $P+1$ sends data to node P at step $k = 1$, and node $P+1$ does not use the overlap region of buffer $b = \mathsf{B}$ again until after step $l = 2$. Therefore, node P knows that it can safely deposit data into buffer $b = \mathsf{B}$ at step $l = 2$ without additional synchronization.

These conditions can be defined by the *BufferReady* predicate. The set $BufferReady_l$ encodes when it is safe to piggy-back the sequence information for statement l. For each entry $< P, Q, b >$, node P can deposit data into buffer b on node Q with no additional control messages. This set is defined in terms of the following predicates.

Pred$_l$ The set of statements that are immediate predecessors of statement l.

Used$_l(P, b)$ True if node P uses buffer b during statement l.

Sent$_l(P, Q)$ True if node P deposits a message onto node Q during statement l.

With these predicates, the *BufferReady* set can be recursively defined as follows:

$$\begin{aligned} BufferReady_l = \{< P, Q, b > \quad | \quad & \forall k \in Pred_l : \\ & \neg Used_k(Q, b) \wedge (Sent_k(Q, P) \vee < P, Q, b > \in BufferReady_k)\} \end{aligned} \tag{6.1}$$

This set can be calculated by a forward data flow algorithm[1]. The *gen* set contains all pairs of nodes that communicate in step l combined with all arrays used in the computation. Each communication in step l can potentially carry synchronization information to subsequent steps.

$$gen(l) = \{< P, Q, b > |Sent_l(Q, P)\} = \{< P, Q, * > |Sent_l(Q, P)\} \tag{6.2}$$

The *kill* set contains all pairs of nodes and buffers in $Used_l$ combined with all sending nodes. Any array that is used on the receiving node kills any possibility that previous messages can be used to piggy-back state information about the array.

$$kill(l) = \{< P, Q, b > | \, Used_l(Q, b)\} = \{< *, Q, b > | \, Used_l(Q, b)\} \tag{6.3}$$

Set intersection is the confluence operator for this problem, so the *in* and *out* sets are calculated from the *gen* and *kill* sets as follows:

$$in(l) = \bigcap_{k \in Pred_l} out(k)$$

$$out(l) = (in(l) \cup gen(l)) - kill(l)$$

Initially, all *in* and *out* sets are the universal set, the set containing all triples. After reaching the fixed point of the data flow computation, *in(l)* contains a conservative approximation of $BufferReady_l$.

6.3.1 Working with communication maps

Instead of working with each pair of nodes separately, the data parallel model lets the compiler work more succinctly with communication maps. The triples of the *BufferReady* set ($< P, Q, B >$) are replaced with pairs $< m, b >$, where m is a communication map that describes the mapping from all processors P to all processors Q, and b is the target buffer. The data flow equations can be changed to operate over communication maps instead of node pairs. The *gen* set includes all maps m that describe communication patterns required in statement l.

$$gen(l) = \{< m, b > |\forall P : (m(Q) = P) \rightarrow Sent_l(Q, P)\}$$

For a $< Q, b >$ in $Used_l$, the *kill* set includes all maps that include Q in their range.

$$kill(l) = \{< m, b > |\exists P : (m(P) = Q) \rightarrow Used_l(Q, b)\}$$

The same equations can be used to calculate the *in* and *out* sets using the pairs of communication maps and buffers. However, set intersection and union becomes slightly more complicated.

Taking the intersection and union of communication maps does not always result in a simple, single communication map. All other communication maps are subsets of the all-to-all communication pattern, $*(p) = \{q | 0 \leq q < P\}$., so the union of any map m with $*$ is $*$, and the intersection of m with $*$ is m. For other combinations, our algorithm is conservative. For union and intersection, the algorithm creates a compound communication map, which contains a list of simple communication maps to intersect or union.

[1]This algorithm is presented using the standard data flow format described in [ASU87]

Stmt	Gen set	Kill set
0	$\{\}$	$\{\}$
1	$\{<+\mathit{shift}, A><-\mathit{shift}, A><+\mathit{shift}, B><-\mathit{shift}, B>\}$	$\{<*, A>\}$
2	$\{<+\mathit{shift}, A><-\mathit{shift}, A><+\mathit{shift}, B><-\mathit{shift}, B>\}$	$\{<*, B>\}$

Table 6.1: gen *and* kill *sets for the SOR example.*

6.4 Examples

This section describes how the synchronization elimination algorithm can be applied to two applications to show the benefits and the drawbacks of this approach.

6.4.1 SOR

First we describe the computation of the *in* and *out* sets for the SOR example of Figure 4.1 The communication maps in this example are: $+\mathit{shift}(p) = p + 1$, $-\mathit{shift}(p) = p - 1$, and the all-to-all map $*(p) = \{q|0 \leq q < P\}$. Table 6.1 shows the *gen* and *kill* sets computed for each statement.

The universal set for this example is $\{<*, A><*, B>\}$. For a processor Q that uses a buffer b in a statement, the kill set includes all maps with Q in the map's domain. For simplicity, we make the conservative assumption that Q is in the range of all maps, so all kill sets used by the compiler are of the form $<*, b>$.

Initially, all *in* and *out* sets are the universal set. The fixed point in this example is reached after two iterations. Table 6.2 shows the *in* and *out* sets for the first two iterations. Once the fixed point has been reached, the *in* sets of each statement contain information that approximates the *BufferReady* predicate. For example, the *in* set of statement 1 shows that it is safe to deposit data into array *A* on the right and left neighboring nodes without additional synchronization. Similarly, the *in* set of statement 2 shows that it is safe to deposit data into array *B* on the right and left neighboring nodes. Therefore, the data flow algorithm shows that SOR can piggy-back all sequencing information on previous communication exchanges.

6.4.2 Two dimensional FFT

Programs that have repeating, regular communication patterns are obvious candidates for explicit synchronization elimination. The two dimensional fast Fourier transform (2D FFT) code shown in Figure 6.4 is one such program. The array `A` is distributed by columns. The nodes compute FFTs over the columns, transpose `A` into `B`, compute FFTs over the columns of `B`, and transpose `B` back to `A`. The transpose operations require all-to-all communication steps.

One might assume that the all-to-all communication guarantees that all nodes have passed the previous communication step by the time any node reaches the next communication step, but a node may be delayed at the previous communication step after it has exchanged data with some of the other nodes. Figure 6.5 shows an example situation of a delayed node. In this example, node 3 is delayed after sending data to node 1. Node 1 receives both messages,

After iteration 1

Stmt	in	out
0	$\{<*, A><*, B>\}$	$\{<*, A><*, B>\}$
1	$\{<*, A><*, B>\}$	$\{<*, B>\}$
2	$\{<*, B>\}$	$\{<+\mathit{shift}, A><-\mathit{shift}, A>\}$

After iteration 2

Stmt	in	out
0	$\{<*, A><*, B>\}$	$\{<*, A><*, B>\}$
1	$\boxed{\{<+\mathit{shift}, A><-\mathit{shift}, A>\}}$	$\{<+\mathit{shift}, B><-\mathit{shift}, B>\}$
2	$\boxed{\{<+\mathit{shift}, B><-\mathit{shift}, B>\}}$	$\{<+\mathit{shift}, A><-\mathit{shift}, A>\}$

Table 6.2: in *and* out *sets for the first two iterations of the data flow algorithm. Boxed entries show the approximation of the* BufferReady *set for the communicating statements.*

```
do i=1,iter_cnt
0. input(A)
1. fft(A)
2. transpose(A,B)
3. fft(B)
4. transpose(B,A)
5. output(A)
enddo
```

Figure 6.4: Code for two dimensional fast Fourier transform (2D FFT).

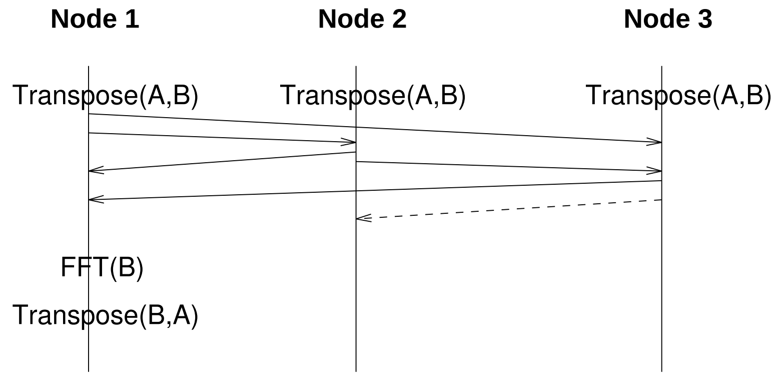


Figure 6.5: Three nodes computing a 2D FFT. Node 3 is delayed before it finished sending to node 2 as shown by the dashed arrow.

Stmt	Gen set	Kill set
0	$\{\}$	$\{<*, A>\}$
1	$\{\}$	$\{<*, A>\}$
2	$\{<*, A><*, B>\}$	$\{<*, A>\}$
3	$\{\}$	$\{<*, B>\}$
4	$\{<*, A><*, B>\}$	$\{<*, B>\}$
5	$\{\}$	$\{<*, A>\}$

Table 6.3: gen *and* kill *sets for the 2D FFT example.*

computes the FFTs and continues onto the next communication step. Node 2 deposits both of its messages but is stalled waiting for the message from node 3. It is not safe for node 1 to deposit data into `A` until node 3 finishes sending data from `A`. Since both arrays are used in every communication step and a node may either be in communication step l or $l+1$, additional synchronization is necessary to ensure that all nodes are in the same communication step.

Table 6.3 shows the *gen* and *kill* sets for the 2D FFT program. The data flow algorithm takes two iterations to reach a fixed point, and Table 6.4 shows the *in* and *out* sets for the first two iterations. The *in* sets for the transpose statements are empty, so the all-to-all communication for the transpose statements requires an additional synchronization step.

However, when calculating two 2D FFTs in each iteration, no additional synchronization is needed. Figure 6.6 shows such an example. Alternating transpose operations work on disjoint sets of buffers, so the target buffer of the current all-to-all communication was not used in the previous all-to-all communication.

6.5 Improving the approximation

The data flow algorithm described in Section 6.3 does not take into account which subsections of the arrays are read or written during a communication step. By increasing the accuracy of the

After iteration 1

Stmt	in	out
0	$\{<*,A><*,B>\}$	$\{<*,B>\}$
1	$\{<*,B>\}$	$\{<*,B>\}$
2	$\{<*,B>\}$	$\{<*,B>\}$
3	$\{<*,B>\}$	$\{\}$
4	$\{\}$	$\{<*,A>\}$
5	$\{<*,A>\}$	$\{\}$

After iteration 2

Stmt	in	out
0	$\{\}$	$\{\}$
1	$\{\}$	$\{\}$
2	$\boxed{\{\}}$	$\{<*,B>\}$
3	$\{<*,B>\}$	$\{\}$
4	$\boxed{\{\}}$	$\{<*,A>\}$
5	$\{<*,A>\}$	$\{\}$

Table 6.4: in *and* out *sets for the first two iterations of the data flow algorithm over the 2D FFT.*

```
do i=1,iter_cnt
   input(A)
   fft(A)
   transpose(A,B)
      input(C)
      fft(C)
      transpose(C,D)
   fft(B)
   transpose(B,A)
   output(A)
      fft(D)
      transpose(D,C)
      output(C)
enddo
```

Figure 6.6: Code that computes two 2D FFTs in each iteration. This program does not require additional synchronization.

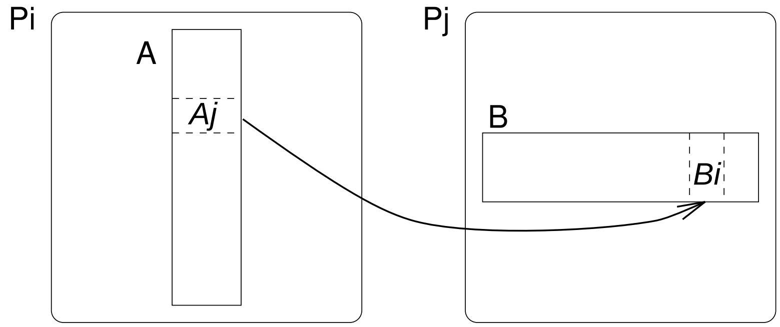


Figure 6.7: The array subsection division during a transpose from `A` *to* `B` *in 2D FFT.*

analysis to keep track of array subregions, the need for synchronization can be reduced further.

By analyzing array subsections, the compiler can determine that additional synchronizations are not really needed in the original 2D FFT code shown in Figure 6.4. During the transpose in statement 2, node Q sends data from a subsection of `A` (A_x) to node P only, and A_x is not used by Q in statement 3. In statement 4, node P needs to deposit data into A_x on node Q. Since node P received a message from Q's last use of A_x, P knows that the old values of A_x are no longer in use even if node Q has not finished statement 2.

During a transpose operation from `A` to `B` over N processors, the compiler can logically divide the arrays into N disjoint subsections such that node P_i deposits its local section A_j into section B_i on node P_j as shown in Figure 6.7. Figure 6.8 shows another timeline of the delayed node in the all-to-all communication of Figure 6.5, but in this timeline, each arrow is labeled with the local array subsection that is sent or overwritten. With this labeling, it is clear that node 1 can continue and send its messages since the data in array subsections A1 on nodes 2 and 3 has already been sent.

Using this array subsection division, the *kill* set defined in Equation 6.3 can be augmented as follow.

$$kill(l) = \{< P, Q, b > \mid Used_l(Q, b) - SoleReceiver_l(P, Q, b)\}$$

The triple $< P, Q, b >$ is in $SoleReceiver_l$ if node P is the only recipient of data from buffer b on node Q. If b is used by Q for communication, and node P is the only recipient of that data, then even if node Q has not completed statement l before node P starts the next communication statement, node P knows that Q is finished with the data in b.

Table 6.5 shows the augmented *gen* and *kill* sets for each statement of the original 2D FFT program. The data flow algorithm reaches a fixed point in two iterations. The *in* and *out* sets for these iterations are shown in Table 6.6. The *in* sets that approximate the *BufferReady* set for the communicating statements are boxed. For statement 2, the *in* set shows that P_i can deposit data into array subsection A_j on P_j without additional synchronization. This is the communication required by the transpose operation, so no additional synchronization is required.

The 2D FFT example shows that the compiler can improve its data flow approximation by analyzing how array subsections are read and written. In some cases the compiler does not have the information to perform this analysis, and in other cases the benefits of the improved approximation may not be worth the cost of the additional analysis.

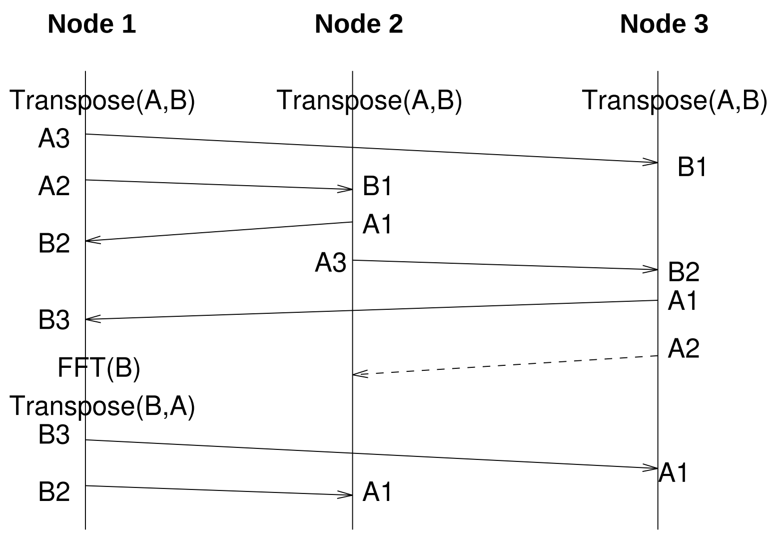


Figure 6.8: Three nodes computing a 2D FFT. Node 3 is delayed before it finished sending to node2 as shown by the dashed arrow. Arrows are labeled with local array subsection that is sent or received.

Stmt	Gen set	Kill set
0	$\{\}$	$\{<*,*,A>\}$
1	$\{\}$	$\{<*,*,A>\}$
2	$\{ <*,*,A><*,*,B> \}$	$\{ <P_i, P_j, A_k> \mid j \neq k \}$[a]
3	$\{\}$	$\{ <*,*,B> \}$
4	$\{ <*,*,A><*,*,B> \}$	$\{ <P_i, P_j, B_k> \mid j \neq k \}$
5	$\{\}$	$\{ <*,*,A> \}$

[a] $\{ <P_i, P_j, A_k> \}$ - $\{ <P_i, P_j, A_j> \}$

Table 6.5: Augmented gen *and* kill *sets for the 2D FFT example.*

After iteration 1

Stmt	in	out
0	$\{<*,*,A><*,*,B>\}$	$\{<*,*,B>\}$
1	$\{<*,*,B>\}$	$\{<*,*,B>\}$
2	$\{<*,*,B>\}$	$\{<*,*,B>\}\cup\{<P_i,P_j,A_k>\|j=k\}\}$[a]
3	$\{<*,*,B>\}\cup\{<P_i,P_j,A_k>\|j=k\}\}$	$\{<P_i,P_j,A_k>\|j=k\}$
4	$\{<P_i,P_j,A_k>\|j=k\}$	$\{<*,*,A>\}\cup\{<P_i,P_j,B_k>\|j=k\}$
5	$\{<*,*,A>\}\cup\{<P_i,P_j,B_k>\|j=k\}$	$\{<P_i,P_j,B_k>\|j=k\}$

[a] $\{<*,*,B>\}\cup(\{<*,*,A>\}$ - $\{<P_i,P_j,A_k\|j\neq k\}\})$

After iteration 2

Stmt	in	out
0	$\{<P_i,P_j,B_k>\|j=k\}$	$\{<P_i,P_j,B_k>\|j=k\}$
1	$\{<P_i,P_j,B_k>\|j=k\}$	$\{<P_i,P_j,B_k>\|j=k\}$
2	$\boxed{\{<P_i,P_j,B_k>\|j=k\}}$	$\{<P_i,P_j,A_k>\|j=k\}$
3	$\{<P_i,P_j,A_k>\|j=k\}$	$\{<P_i,P_j,A_k>\|j=k\}$
4	$\boxed{\{<P_i,P_j,A_k>\|j=k\}}$	$\{<P_i,P_j,B_k>\|j=k\}$
5	$\{<P_i,P_j,B_k>\|j=k\}$	$\{<P_i,P_j,B_k>\|j=k\}$

Table 6.6: in *and* out *sets for the first two iterations of the data flow algorithm over the 2D FFT program using array subsections. Boxed entries approximate the* BufferReady *set for the communicating statements.*

6.6 Effects of synchronization elimination

Data flow synchronization elimination has been implemented in the communication optimization phase of the Fx compiler. This implementation recognizes redundant synchronization in many programs that include repeating communication patterns such as the SOR and the pipelined 2D FFT programs described in Section 6.3. For simplicity of implementation, the prototype compiler does not analyze array subsections as described in Section 6.5.

While the algorithm can recognize redundant synchronization, it may not always make sense to eliminate the synchronization step. For machines like the Cray T3D with dedicated synchronization hardware, a subset synchronization takes a matter of 5 to 10 cycles, far less than a complete communication step. Even the synchronization implementation on iWarp is very fast compared to a complete communication step. Removing synchronization on these machines will not result in very significant changes in execution time unless the program is inherently asynchronous.

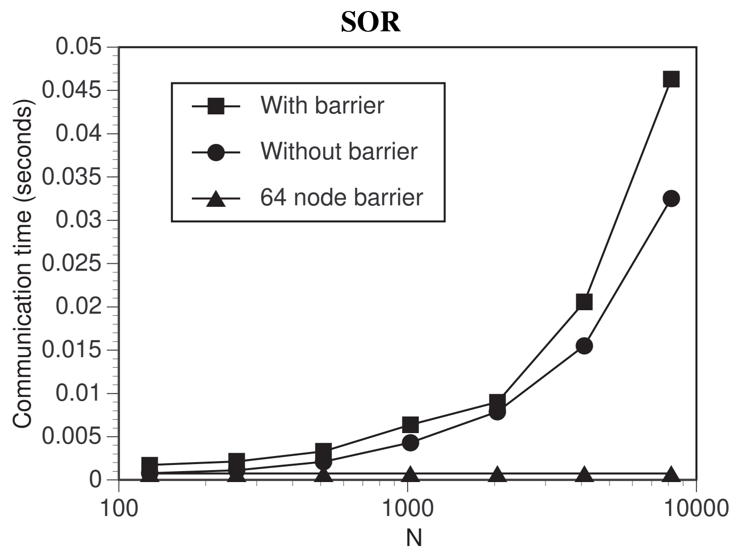
However, for machines like Paragon, synchronization elimination shows more potential for performance improvement. Paragon has no dedicated hardware for synchronization. Instead, it sends control messages over the data network to implement barrier synchronization, so the synchronization overhead can be relatively expensive compared to the communication step time. We measured the effects of barrier synchronization for two dimensional SOR and 2D FFT on 64 nodes of a Paragon system running the OSF operating system.

Figure 6.9(a) shows the average communication time for one iteration of the SOR program. The line labeled With barrier shows the communication time including the barrier synchronization. The line labeled Without barrier shows the communication time without the barrier synchronization. The line labeled 64 node barrier shows the time to execute a barrier synchronization over 64 nodes.

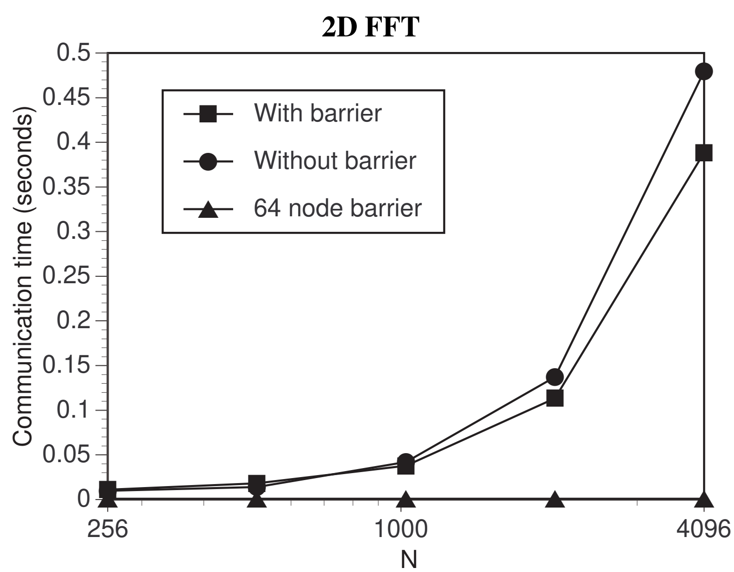
The difference in communication times remains relatively constant for small input sizes, and this difference corresponds to the barrier synchronization time. The different grows as the problem size increases, perhaps due to asynchronous behavior between the SOR steps. Table 6.7(a) shows the difference in total execution time due to the barrier synchronization elimination. The performance difference for small problems sizes is quite noticeable. As the problem size increases, the relative amount of communication decreases, so the relative benefit in total execution time decreases as the problem size grows.

2D FFT requires an all-to-all communication pattern, where every node sends a unique message to every other node. This is a dense communication pattern, prone to network congestion. By contrast, SOR requires only nearest neighbor communication, so it is not likely to encounter congestion delays. Unlike the SOR example, the performance effects of eliminating synchronization for the 2D FFT are not as clear.

Figure 6.9(b) shows the average communication time for one iteration of the pipelined 2D FFT program. Again, the line labeled With barrer shows the time for the implementation that does not eliminate the synchronization, and the line labeled Without barrer shows the time for the implementation that eliminates the synchronization. The total amount of communication time is quite large compared to the synchronization time (shown by the line labeled 64 node barrier barely visible above the x axis). Merely eliminating the overhead of performing a synchronization should not greatly affect the communication time.



(a)



(b)

Figure 6.9: Average communication time for one iteration on a 64 node Paragon system for SOR (a) and 2D FFT (b).

Problem size	Speed up
128	50 %
256	36 %
512	18 %
1024	10 %
2048	1.3 %
4096	1.7 %
8192	1.4 %

(a)

Problem size	Speed up
256	6.5 %
512	7.5 %
1024	-2.1 %
2048	-2.6 %
4096	-1.9 %

(b)

Table 6.7: Tables of the percentage of total program execution speed up due to synchronization elimination for two applications: SOR (a) and 2D FFT (b).

For small problem sizes, eliminating the synchronization slightly improves communication performance. However, for problem sizes greater than 512×512, removing synchronization actually makes the communication performance worse. Table 6.7(b) shows the effect of barrier elimination on total execution time.

For all-to-all communication, synchronization actually performs two functions. In addition to ensuring that no data is deposited before its time, the synchronization acts as a congestion control mechanism, limiting the set of messages that can be in the network at one time. Without synchronization, one node can be lucky and insert many messages for step i while delaying nodes trying to insert messages for step $i - 1$. Eventually, all nodes must wait for the delayed nodes. By allowing some nodes to go to the next step total execution time can increase.

Similar performance improvements due to using synchronization for congestion control have been shown in [HKO$^+$94] for all-to-all communication on iWarp and T3D and for communication on the CM-5[BK94]. In [SS94], Stamatopolus and Solworth propose a network architecture that incorporates barrier synchronization for congestion control into regular message passing communication. Their simulations show how varying the frequency of synchronization affects the bandwidth available for the application.

6.7 Discussion

The previous section evaluates the effect of barrier elimination on two simple programs to determine the effect of barrier elimination on extremes in communication patterns. These measurements indicate that on the Paragon barrier elimination is beneficial for "sparse" communication patterns, but for "dense" communication patterns the benefits of eliminating synchronization are over shadowed by the costs of losing the synchronization for network congestion.

For other systems without hardware support for barrier synchronization, the tradeoffs may be different. For example, workstation clusters connected by high-speed networks (e.g. ATM switches or FDDI rings) are not likely to have hardware barrier synchronization support, but such systems may still benefit from using the deposit model to avoid buffering. It is not clear how synchronization affects network congestion in systems arranged in non-mesh or irregular topologies. Current measurements that show the impact of synchronization on network congestion are concentrated on mesh-connected systems (iWarp, Paragon, T3D) or at least systems with regular topologies (CM-5).

Some of the benefits of the deposit model can be achieved by using the standard message passing interface. By posting receives "far enough" in advance, the programmer can guarantee that the receiver is always ready to consume the incoming message directly. In fact, the Paragon User's Guide suggests posting a receive and then exchanging control messages to ensure the data will not be buffered[Int94]. By using the the standard message passing interface in this manner, the synchronization elimination analysis can also be used to show where receives can be posted to guarantee that buffering will be unnecessary.

The code segments in Figure 6.10 shows variations in communication code for the SOR data parallel example in Chapter 4. Figure 6.10(a) and (b) show communication codes using the deposit and standard message passing interfaces that rely on barrier synchronization to ensure that messages will never be buffered. After running the barrier elimination algorithm described in Section 6.3, the code that uses the deposit model interface can merely eliminate the barrier synchronization (as shown in Figure 6.10(c)). To eliminate synchronization from the code that uses the standard interface requires a bit more work. The programmer must move the receive call before the send that implicitly carries the control information. In essence, the receive requests must be *pipe-lined* (as shown in Figure 6.10(d)).

6.8 Chapter summary

This chapter describes how the compiler can recognize when additional synchronization is unnecessary in the deposit message passing model. For many programs with repeating communication patterns, the communication is self-synchronizing and additional synchronization is not needed for correctness.

However, our observations of barrier elimination on Paragon show that the performance effects of barrier synchronization are not straightforward. Eliminating barriers will not be beneficial unless at least one of the following conditions hold.

- The communication pattern is not prone to congestion on the target network.
- The barrier synchronization overhead is relatively large compared to the total communication time.
- The application code can benefit from asynchronous execution.

Therefore, synchronization elimination should not be applied blindly. To avoid detrimental side effects, the compiler writer must understand the secondary effects of synchronization on network congestion in the target system.

```
do i=1,iter_cnt
   barrier_sync()
   deposit(A,p-1); deposit(A,p+1)
   wait(num_msgs)
   B(lb:ub) = c1*A(lb-1:ub-1) +
      c2*A(lb+1:ub+1)
   barrier_sync()
   deposit(B,p-1); deposit(B,p+1)
   wait(num_msgs)
   A(lb:ub) = c1*B(lb-1:ub-1) +
      c2*B(lb+1:ub+1)
enddo
```

(a)

```
do i=1,iter_cnt
   receive(A,&a1status); receive(A,&a2status)
   barrier_sync()
   send(A,p-1); send(A,p+1)
   wait(a1status); wait(a2status)
   B(lb:ub) = c1*A(lb-1:ub-1) +
      c2*A(lb+1:ub+1)
   receive(B,&b1status); receive(B,&b2status)
   barrier_sync()
   send(B,p-1); send(B,p+1)
   wait(b1status); wait(b2status)
   A(lb:ub) = c1*B(lb-1:ub-1) +
      c2*B(lb+1:ub+1)
enddo
```

(b)

```
do i=1,iter_cnt
   deposit(A,p-1); deposit(A,p+1)
   wait(num_msgs)
   B(lb:ub) = c1*A(lb-1:ub-1) +
      c2*A(lb+1:ub+1)
   deposit(B,p-1); deposit(B,p+1)
   wait(num_msgs)
   A(lb:ub) = c1*B(lb-1:ub-1) +
      c2*B(lb+1:ub+1)
enddo
```

(c)

```
receive(A,&a1status); receive(A,&a2status)
do i=1,iter_cnt
   receive(B,&b1status); receive(B,&b2status)
   send(A,p-1); send(A,p+1)
   wait(a1status); wait(a2status)
   B(lb:ub) = c1*A(lb-1:ub-1) +
      c2*A(lb+1:ub+1)
   receive(A,&a1status); receive(A,&a2status)
   send(B,p-1); send(B,p+1)
   wait(b1status); wait(b2status)
   A(lb:ub) = c1*B(lb-1:ub-1) +
      c2*B(lb+1:ub+1)
enddo
```

(d)

Figure 6.10: SOR pseudo-code for node `p`*. The communication code shows differences between using the deposit model(a) and the standard message passing interface(b) to avoid buffering. (c) and (d) show how the two interfaces can use information from the synchronization elimination algorithm to avoid using additional control messages.*

Chapter 7

Conclusions

For most distributed applications, the relative cost of communication diminishes as the size of the problem increases (relative to the machine's size). When the amount of work per node is large enough, the communication overhead is unimportant, and communication optimizations are not vital. However, optimizing communication performance increases the amount of effective parallelism, making it practical to distribute the arrays into smaller blocks over more processors. Therefore, reducing communication overhead is essential for massively parallel systems.

To reduce the overheads of communication, it is necessary to use a communication model that is appropriate for the target machine. Our measurements on iWarp and Paragon show that the best communication model depends on the communication pattern and the target architecture. The static connection-based model is superior on iWarp, because software can directly manage the hardware communication resources, but the static reservation of software communication resources is also beneficial on Paragon for sparse communication patterns.

Therefore, the best implementation of a parallel application depends on the communication patterns required and the target architecture. With communication pattern analysis and information about the target architecture, the data parallel compiler is better suited to performing architecture-dependent optimizations than the human programmer.

I verified that a compiler can use information about architecture to improve communication performance by building a prototype communication generation and optimization phase in the Fx compiler. My application measurements on iWarp and Paragon show that using knowledge about the target architecture makes a significant performance difference; these differences can affect total execution time by up to 30%.

The techniques described in this thesis are not tightly bound to HPF. Some of the techniques are useful outside the domain of parallel compilers; they can be useful for runtime libraries or even dedicated human programmers. For example, a runtime library can perform some of the communication model tradeoffs described in Chapter 4 particularly for systems like the Paragon that do not have strongly limited communication resources. A tool might also be able to perform the synchronization elimination analysis by examining message passing code with calls to a collective communication library.

However, all of these communication optimization techniques rely on knowing the global communication patterns. The data parallel programming model provides a global view of the program that greatly simplifies the identification of the communication patterns. In general, looking a parallel program at a higher level of abstraction simplifies many of these global communication optimizations and assignments.

7.1 Future work

The thesis makes several limiting assumptions in Chapter 2.

- Programs use the data parallel compiler model.
- All communication patterns are regular.
- The target system is dedicated to a single application.

In addition, my measurements have concentrated on programs in the scientific domain that run on explicit message passing machines. Loosening any one of these restrictions reveals a large area of additional work.

Alternative compilation models Combining task and data parallelism has generated a lot of interest recently[SSOG93, Fos94]. Depending on the compilation model, it may also be practical to derive the inter-task communication patterns and optimize these patterns for the target architecture. Some preliminary measurements with Fx on iWarp show that varying the inter-task communication strategy for different problem sizes can be beneficial[GHS94].

However, scheduling resource usage for communication patterns in both the task and the data parallel models can be difficult, since communication traffic in the data parallel model may interfere with traffic from the task parallelism model, and visa versa. The performance estimates used by the communication code selection algorithm may depend on what task parallel communication is also occurring. Similarly, the resource management algorithms must be aware of both the data parallel communication and the task parallel communication.

Irregular communication patterns Many important problems require communication patterns that cannot be analyzed at compile time. However, some knowledge about the communication pattern is frequently known at compile-time, such as the maximum number of nodes each node communicates with (i.e. the density of communication). By using compile-time knowledge of the target system and partial knowledge about the communication pattern, the compiler can make some decisions about the appropriate communication model. For example on the Paragon system, knowing whether a communication pattern is sparse or dense is enough information to select between the static and dynamic resource reservation models.

In many programs, the runtime communication pattern changes infrequently. A architecture-specific communication selection algorithm could be executed at runtime by using a strategy similar to the Chaos runtime library[SAS92]. This library performs an *inspector* phase that calculates the runtime schedule once for many executions of the communication pattern. Similarly, a communication selection algorithm could gather global information about the communication pattern and make a code selection once for many iterations.

Operating system interactions If the parallel system is shared between multiple applications either by space-sharing or time-sharing nodes, a dedicated runtime system is not sufficient and a true operating system is required. The operating system must manage shared system resources to ensure the applications execute safely and fairly. In a multi-user system, the network is a shared resource. Therefore, the system cannot directly allow the user program to schedule the shared communication resources.

Much of these operating system services are in place to prevent errant or malicious programs from hurting the rest of the system. Once the compiler is debugged, compiler-generated parallel programs should not need all of these safeguards. The compiler should be able to guarantee that the code it generates follows the sharing policies required by the operating system.

Future distributed-memory machines My research has concentrated on distributed-memory machines. A number of distributed shared memory machines have recently been proposed (e.g. Flash[K$^+$94], Shrimp[B$^+$94b]). These architectures follow from several architectural decisions that are different from distributed-memory machines. In particular, these machines move data at a finer granularity.

Current distributed shared-memory machines such as Dash and KSR[Ken92] have fixed caching and data movement strategies. These fixed strategies are analogous to Paragon's fixed routing strategies. The compiler can use similar software-based protocols on these machines to work around the particular hard-wired data movement strategies.

Newer distributed shared-memory designs such as Flash enable the software to change the caching protocol on a page by page basis. This flexibility is analogous to the flexibility of machines like iWarp and ATM networks that reveal their resource managements strategies to software control. With this programmability, the compiler can suggest more suitable communication protocols for different patterns.

Section 5.1.5 describes how aspects of the programmable protocols can be viewed as communication resources. Depending of the number of protocols that can be simultaneously in use and the cost of changing protocols, the resource managements algorithms described in Chapter 5 may also be useful for managing these flexible distributed shared memory machines.

The developing electro-optical networks also show some performance characteristics that can benefit from static resource reservation. In these networks, each node has a number of mirrors that can be used to send data optically to a number of different destinations[Exm95]. Each mirror is directed by electronics, and the cost of redirecting a mirror is relatively high. In this case the mirror is the communication resource, and the directed optical beam corresponds to a connection.

Other program domains Most research in parallelizing compilers and communication optimizations has concentrated on scientific programs. While this is an important problem domain, there are many other problem domains that could benefit from parallel computation such as database, multi-media, and signal processing domains.

If problems in these domains have repeating communication patterns than can be analyzed or traced, similar architecture-dependent optimizations can also be beneficial for them. For example, many database applications have repeating data access patterns. On distributed databases, these data access patterns translate to communication patterns.

Consider company X that performs the same set of database queries every hour on a distributed database of potentially changing data. By tracing previous accesses, the system knows that processor A will broadcast a series of messages during the query, and the remaining processors will redistribute data from several tables to form the join of several tables. With this communication pattern information and knowledge about the communication architecture, the system can select the most appropriate models to perform this communication. If the interconnection network exposes limited communication resources (e.g. ATM switches), the system can also use the communication pattern information to intelligently schedule the communication resources.

7.2 Closing statement

This thesis argues that it is possible and beneficial for a parallelizing compiler to use architecture-specific information when generating communication code. In this dissertation, I have supported this argument by the following:

- Describing the impact of static and dynamic communication resource reservation on communication performance and communication models.
- Describing a compiler framework that can use architecture-specific communication information for communication code selection.
- Presenting algorithms that enable the compiler to target and optimize communication for static and dynamic communication resource reservation: resource management code selection for the static resource reservation model and synchronization elimination for the dynamic resource reservation model.
- Presenting program measurements to show the benefits of these optimizations on iWarp and Paragon parallel systems.

If code generated by parallel compilers is to reach the efficiency of "hand-parallelized" code, compilers must generate communication code beyond the standard message passing interface. Just as P-code is not sufficient for uniprocessor optimizations, the compiler must understand the target communication architecture to generate efficient communication code. Either the compiler must export application communication pattern knowledge to lower communication levels, or the compiler must import specific information about the target architecture and environment.

With the current development of radically different distributed-memory machines from ATM workstation clusters to distributed shared-memory systems, it does not appear that a canonical communication architecture will appear in the near future. Therefore, it will continue to be important for parallelizing compilers to generate communication code tailored for the target machine.

Bibliography

[A^+88] F. Allen et al. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[AKLS88] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of ACM/SIGPLAN PPEALS 1988*, pages 42–56, New Haven, CT, September 1988. ACM.

[AL93] S. Amarasinghe and M. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Programming Language Design and Implementation*, pages 126–138, Albuquerque, NM, June 1993. ACM.

[ASU87] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 10.5 Introduction to Global Data-Flow Analysis, pages 608–623. Addison-Wesley, 1987.

[B^+88] S. Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[B^+90] S. Borkar et al. Supporting Systolic and Memory Communication in iWarp. In *Proc. 17th Intl. Symposium on Computer Architecture*, pages 70–81. ACM, May 1990. A revised version has appeared as technical report CMU-CS-90-197.

[B^+94a] M. Barnett et al. Building a High-Performance Collective Communication Library. In *Proceedings of Supercomputing '94*, pages 107–116, Washington, D.C., November 1994.

[B^+94b] M. Blumrich et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.

[BCS93] E. Biagioni, E. Cooper, and R. Sansom. Designing a Practical ATM LAN. *IEEE Network*, 7(2):32–9, March 1993.

[BHMS91] M. Bromley, S. Heller, T. McNerney, and G. L. Steele, Jr. Fortran at ten gigaflops: The connection machine convolution compiler. In *Proceedings of SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ont., June 1991.

[BHS+94] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[BK94] E. Brewer and B. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *International Parallel Processing Symposium*, pages 858–867, Cancun, Mexico, April 1994.

[BW93] A. J. C. Bik and A. G. Wijshoff. Compilation techniques for sparse matrix computation. In *Proceedings of ICS '93*, pages 416–424, Tokyo, Japan, July 1993.

[C+87] A. Carle et al. A Practical Environment for Scientific Programming. *Computer*, 20(11):75–89, November 1987.

[C+92] A. Choudhary et al. Compiling Fortran 77D and 90D for MIMD Distributed-Memory Machines. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[CFR+88] R. Cytron, R. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of Principles of Programming Languages*, 1988.

[CGS93] S. Chatterjee, J. Gilbert, and R. Schreiber. Mobile and Replicated Alignment of Arrays in Data-Parallel Programs. In *Supercomputing '93*, pages 420–429, 1993.

[CGST93] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[CKP+92] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. *LogP: towards a realistic model of parallel computation*. Technical Report UCBC 92-713, Univ. of California, Berkeley, 1992. expanded version of paper in 4th Symp. on PPoPP.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 16. Dynamic Programming, pages 301–328. The MIT Electrical and Computer Science Series. The MIT Press, Cambridge, MA, 1990.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Viena Fortran. *Scientific Programming*, 1:31–50, fall 1992.

[CW92] L. A. Crutcher and A. G. Waters. Connection Management for an ATM Network. *IEEE Network*, 6(6):42–55, November 1992.

[Dal92] W. J. Dally. Virtual-Channel Flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[Div91] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[DP93] P. Druschel and L. L. Peterson. FBufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, Asheville, NC, December 1993.

[DZO92] H. G. Dietz, A. Zaafrani, and M. T. O'Keefe. Static Scheduling for Barrier MIMD Architectures. *The Journal of Supercomputing*, 5:263–289, 1992.

[Exm95] I. Exman. Optical Implementation of Collective Communications. Talk at Carnegie Mellon University., March, 10 1995.

[Fel93] E. W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. Ph.D. thesis, University of Washington, 1993.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. *Fortran D Language Specification*. Technical Report TR90-141, Rice University, December 1990.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 1.0.*, May 1993.

[Fos94] I. Foster. Task parallelism and high performance languages. *IEEE Parallel and Distributed Technology*, 2(3):27–36, Fall 1994.

[FSW93] A. Feldmann, T. M. Stricker, and T. E. Warfel. Supporting sets of arbitrary connections on iWarp through communication context switches. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 203–212, Schloss Velen, Westfalia, Germany, July 1993.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[GB92a] M. Gupta and P. Banerjee. A Methodology for High-Level Synthesis of Communication on Multicomputers. In *International Conference on Supercomputing*, pages 357–367, 1992.

[GB92b] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–93, September 1990.

[GHS94] T. Gross, S. Hinrichs, and J. Subhlok. Construction and Delivery of Messages for Modular Parallel Programs. In *Transputer Research and Applications 7*, H. Arabina, editor, Transputer and Occam* Engineering Series, pages 176–185, Amsterdam, October 1994. IOS Press.

[GLS93] S. L. Graham, S. Lucco, and O. Sharp. Orchestrating Interactions Among Parallel Computations. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 100–111, Albuquerque, NM, June 1993.

[GPBS94] L. Gravano, G. D. Pifarre, P. E. Berman, and J. L. C. Sanz. Adaptive Deadlock- and Livelock-Free Routing with All Minimal Paths in Torus Network. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1233–1251, December 1994.

[Gre93] D. Greenburg. Efficient Wiring of Reconfigurable Parallel Processors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 318–324, Schloss Velen, Westfalia, Germany, July 1993.

[GS91] J. Gilbert and R. Schreiber. Optimal Expression Evaluation for Data Parallel Architectures. *Journal of Parallel and Distributed Computing*, 13:58–64, 1991.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[HGDG94] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, San Jose, October 1994.

[HHKT92] M. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural Compilation of Fortran D for MIMD Distributed Memory Machines. In *Supercomputing '92*, Minneapolis, MN, November 1992.

[Hin95] S. Hinrichs. Simplifying Connection-Based Communication. *IEEE Parallel and Distributed Technology*, 3(1):25–36, Spring 1995.

[HJ92] D. Henry and C. Jeorg. A Tightly-Coupled Processor-Network Interface. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, Boston, MA, October 1992.

[HKO$^+$94] S. Hinrichs, C. Kosak, D. O'Hallaron, T. Stricker, and R. Take. *An Architecture for Optimal All-to-All Personalized Communication*. Technical Report CMU-CS-94-140, Carnegie Mellon, 1994. Extended version of paper presented at SPAA '94.

[HQL$^+$91] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[Int94] Intel Corporation. *Paragon User's Guide*, 1994.

[Isl94] N. Islam. *Customized Message Passing and Scheduling for Parallel and Distributed Applications*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1994.

[K+93] D. Kuck et al. The Cedar System and an initial performance study. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[K+94] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[Ken92] Kendall Square Research, Waltham, MA. *Kendall Square Research Technical Summary*, 1992.

[KK79] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, September 1979.

[KL70] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[KLD92] K. Knobe, J. D. Lukas, and W. J. Dally. Dynamic Alignment on Distributed Memory Systems. In *Third Workshop on Compilers for Parallel Computers*, pages 394–404, Viena, Austria, July 1992.

[KLS90] K. Knobe, J. D. Lukas, and G. L. Steele, Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[Kon94] S. Konstantinidou. Segment router: a novel router design for parallel computers. In *ACM Symposium on Parallel Algorithms and Architecture*, Cape May, NJ, June 1994. ACM.

[KS91] H. T. Kung and J. Subhlok. A new approach for automatic parallelization of blocked linear algebra computations. In *Proceedings of Supercomputing '91*, pages 122–129, Albuquerque, NM, November 1991.

[L+92] D. Lenoski et al. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

[L+93] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 272–85, 1993.

[LC90] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. In *Frontiers90: 3rd Symposium Frontiers Massively Parallel Computation*, pages 424–433, 1990.

[LC91a] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[LC91b] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[LG94] D. Lahaut and C. Germain. Static Communication in Parallel Scientific Programs. In *PARLE '94: Parallel Architectures and Languages Europe*, C. Halatis et al., editors, Lecture Notes in Computer Science, pages 262–276, Athens, Greece, July 1994. Springer-Verlag.

[Lo88] V. M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[MB93] C. Maeda and B. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 244–255, Asheville, NC, December 1993.

[MMRW94] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.

[MPI93] The Message Passing Interface Forum. *Draft Document for a Standard Message Passing Interface*, November 1993.

[MPS93] W. Moore, M. Pandit, and W. Shubert. *Tau Software Virtual Crossbar Facility*. Intel Supercomputer Systems Division, January 1993. Research report.

[MTW93] M.D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers, and Transputers: Function, Performance, and Application*. IOS Press, Inc., Amsterdam, Netherlands, 1993.

[PR94] P. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Scalable High-Performance Computing Conference*, pages 184–190, Knoxville, TN, May 1994.

[PS92] L. L. Pollock and M. L. Soffa. Incremental Global Reoptimization of Programs. *ACM Transactions on Programming Languages and Systems*, 14(2):173–200, April 1992.

[Pug91] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, 1991.

[RG89] S. Richardson and M. Ganapathi. Interprocedural Optimization: Experimental Results. *Software - Practice and Experience*, 19(2):149–169, February 1989.

[RLW94] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[RS87] J. Rose and G. Steele, Jr. C*: an extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, volume 2, St. Petersburg, FL, May 1987.

[RSW91] M. Rosing, R. B. Schnabel, and R. P. Weaver. The DINO Parallel Programming Language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[S+92] E. J. Schwabe et al. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical solution of pdes. In *Proceedings of the First Annual Summer Institute on Issues and Obstacles in the Practical Implementation of Parallel Algorithms and the Use of Parallel Machines in Parallel Computation (DAGS/PC '92)*, pages 48–62. Dartmouth Institute for Advanced Graduate Studies, June 1992.

[SAS92] A. Sussman, G. Agrawal, and J. Saltz. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(4):73–86, 1992.

[Sco91] D. S. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *The Sixth Distributed Memory Computing Conference Proceedings*, pages 398–403, 1991.

[SG95] T. Stricker and T. Gross. Optimizing Memory System Performance for Communication in Parallel Computers. In *Proc. 22nd Intl. Symposium on Computer Architecture*, pages 308–319, Santa Margherita Ligure, Italy, June 1995.

[SOG94] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.

[SS94] J. Stamatopoulos and J. A. Solworth. Increasing network bandwidth on meshes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 336–345, Cape May, NJ, June 1994. ACM.

[SSO+95] T. M. Stricker, J. Stichnoth, D. R. O'Hallaron, S. Hinrichs, and T. Gross. The Performance Impact of Fast Synchronization in Parallel Computers. In *International Conference on Supercomputing*, pages 1–10, Barcelona, Spain, July 1995.

[SSOG93] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, CA, May 1993.

[Sti93] J. Stichnoth. *Efficient compilation of array statements for private memory systems*. Technical Report CMU-CS-93-109, Carnegie Mellon University, February 1993.

[Sti94] J. Stichnoth. Optimizing Data Movement for Regular and Irregular Parallel Scientific Applications. Thesis proposal, August 1994.

[Sto77] H. S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[Str91] T. Stricker. Message routing on irregular 2d-meshes and tori. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 170–177, Portland, OR, April 1991. Also appeared as Technical Report CMU-CS-91-109, Carnegie Mellon School of Computer Science.

[Sub90] J. Subhlok. *Analysis of Synchronization in a Parallel Programming Environment*. Ph.D. thesis, Rice University, September 1990.

[Sun90] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–39, December 1990.

[TP92] P. Tu and D. Padua. Array Privatization for Shared and Distributed Memory Machines. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, Boulder, CO, October 1992.

[Tse89] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Computers*. Ph.D. thesis, Carnegie Mellon University, 1989.

[Tse95] C.-W. Tseng. Compiler Optimizations for Eliminating Barrier Synchronization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[Tur92] J. S. Turner. Managing Bandwidth in ATM Networks with Bursty Traffic. *IEEE Network*, 6(5):50–58, September 1992.

[Val90] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[vLT87] J. van Leeuwan and R. B. Tan. Interval Routing. *The Computer Journal*, 30(4):298–307, 1987.

[War95] T. E. Warfel. *Tasks and Connection Sets: Choreographed Communication on a Reconfigurable Connection-based Parallel Computer*. Ph.D. thesis, Carnegie Mellon University, August 1995.

[Who91] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. Ph.D. thesis, Carnegie Mellon University, 1991.

[YO94] B. Yang and D. R. O'Hallaron. Procedure Call Models for Distributed Parameters in Data Parallel Programs. In *Scalable Parallel Libraries Conference II*, October 1994.