An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines

Chau-Wen Tseng

Rice COMP TR93-199
January 1993

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892

RICE UNIVERSITY

An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines

by

Chau-Wen Tseng

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy,
Noah Harding Professor, Chair
Computer Science

Keith D. Cooper, Associate Professor
Computer Science

Danny C. Sorensen, Professor
Computational and Applied Mathematics

Houston, Texas

January, 1993

(Available as Rice COMP TR93-199 or CRPC-TR93291)

An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines

Chau-Wen Tseng

Abstract

Massively parallel MIMD distributed-memory machines can provide enormous computational power; however, the difficulty of developing parallel programs for these machines has limited their use. Our thesis is that an advanced compiler can generate efficient parallel programs, *if* data decompositions are provided. To validate this thesis, we have implemented a compiler for Fortran D, a version of Fortran that provides data decomposition specifications at two levels: *problem mapping* using sophisticated array alignments, and *machine mapping* through a rich set of data distribution functions.

The Fortran D compiler is organized around three major functions: program analysis, program optimization, and code generation. Its compilation strategy is based on the "owner computes" rule, where each processor only computes values of data it owns. Data decomposition specifications are translated into mathematical *distribution functions* that determine the ownership of local data. By composing these with subscript functions or their inverses, the compiler can efficiently partition computation and determine nonlocal accesses at compile-time.

Fortran D optimizations are guided by the concept of *data dependence*. Program transformations modify the program execution order to enable optimizations. Communication optimizations reduce the number of messages and overlap communication with computation. Parallelism optimizations detect reductions and optimize *pipelined computations* to increase the amount of useful computation that may be performed in parallel. Empirical evaluations show that exploiting parallelism is vital, while message vectorization, coarse-grain pipelining, and collective communication are the key communication optimizations. A simple model is constructed to guide compiler optimizations. Loop indices, bounds, and nonlocal storage are managed by the compiler during code generation.

Interprocedural analysis, optimization, and code generation algorithms limit compilation to only one pass over each procedure by collecting summary information after edits, then compiling procedures in reverse topological order to propagate necessary information. Delaying instantiation of the work partition, communication, and dynamic data decomposition enables interprocedural optimization. Interactions between the compiler and other elements of the programming system are discussed. Empirical measurements show that the output of the prototype Fortran D compiler is comparable to hand-written codes on the Intel iPSC/860 and significantly outperforms the CM Fortran compiler on the Thinking Machines CM-5.

Acknowledgments

I wish to thank my committee, Ken Kennedy, Keith Cooper, and Danny Sorensen, for their guidance on my thesis. My advisor, Ken Kennedy, has provided me with advice, encouragement, and support throughout my graduate career. I am also indebted to him for the opportunity of working in a first-class research environment, whose excellence I am starting to appreciate only as I leave Rice.

Parts of this thesis represent collaborative efforts with rearchers at Rice and Syracuse universities. I particularly wish to acknowledge the significant contributions of Seema Hiranandani, who has been my partner in performing the research, implementation, and experimentation embodied in this thesis. Geoffrey Fox, Mary Hall, Charles Koelbel, Ulrich Kremer, and others all made valuable contributions. I am also grateful to Alan Carle, Mary Hall, Paul Havlak, Kathryn McKinley, John Mellor-Crummey, Mike Paleczny, and other members of the ParaScope research group for providing the underlying software infrastructure for the Fortran D compiler.

Throughout graduate school I have been fortunate to enjoy the company of some truly excellent people. Kathryn McKinley has been my best pal and an endless source of inspiration and support; she is all heart and soul. Kathryn introduced me to the finer points of research and helped acquaint me with my first BibTeX file. Our research discussions are as fun as they are tempestuous. Seema Hiranandani has been a dear friend and an amazingly effective collaborator. Her contributions to the design and implementation of the Fortran D compiler are at least equal to my own. Seema deserves credit for her encouragement and understanding, not to mention her dependability as a lunch companion.

Pete Keleher was my earliest friend in graduate school, fellow pool shark, and nethack *extraordinere*. He could beat me consistently in tennis even before surgically repairing his knee. Mary Hall has been a close confidant and source of good cheer. Her travails with Chili-dog serve as testimony to why academics should not keep pets. Marina Kalem and Uli Kremer helped found the Fortran D project and establish its early research directions. I appreciated Marina's maturity, wisdom, and enjoyed her humor, especially her oral history of the department. Uli has been a boon companion for working late at night, weekend trips to the bowling alley, and exploring the castles of Europe. Both Marina and Uli have been steadfast in their friendship and support.

Nathaniel McIntosh has been an outstanding friend and colleague since our days developing software for automatic test equipment at Teradyne. I was lucky he decided to pursue a Ph.D. at Rice, as was the department soccer, volleyball, basketball, and softball teams. Preston Briggs has been an excellent friend and fellow jogger, full of entertaining stories about compiler writers and runners. Jerry Roth has been a friend, considerate officemate, and much better basketball player than I. Without the support of these and other friends, I could never have withstood the stresses and uncertainties of graduate school. You have my thanks; I will forever treasure our friendships.

Finally, I wish to acknowledge my parents, Dr. Wen-Shing Tseng and Dr. Jing Hsu. Their love, encouragement, patience, and support have made possible my accomplishments and successes. Thanks for all you have done and sacrificed for my sisters and me, I owe you more than I can say. I only hope someday to be as good to my own children as you have been to me.

Contents

Illustrations

Tables

Chapter 1

Introduction

The power of massively-parallel processing is hindered by the difficulty of parallel programming. Our goal is to develop the compiler technology required to provide a simple yet efficient machine-independent parallel programming model. We show that for dense-matrix numerical computations annotated with data decomposition specifications, an advanced compiler can generate efficient codes for MIMD distributed-memory machines. In this thesis we develop advanced compilation techniques, evaluate them experimentally, and empirically validate their effectiveness in a prototype compiler.

1.1 Background

1.1.1 Machine-Independent Parallel Programming

It is widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to scientists and engineers. Highly parallel supercomputers also provide the best cost/performance ratio of all supercomputers. In particular, distributed-memory machines such as the Intel Paragon or the Thinking Machines CM-5 are among the most powerful, flexible, and scalable parallel computers available. Despite their advantages, parallel machines have only enjoyed limited success because parallel programming is a difficult and time-consuming task. To obtain adequate performance, scientific programmers must write explicitly parallel programs and solve many machine-dependent efficiency issues.

Given these concerns, how can we make parallel supercomputers more accessible and useful for scientific programmers? History shows us that conventional vector supercomputers owe a large part of their success to the introduction of automatic vectorizing compilers. These compilers can take Fortran or C programs written in a *vectorizable style* and automatically convert them to run efficiently on any vector machine [206, 39]. This provides a machine-independent programming model that allows scientific programmers to concentrate on their actual algorithms, introducing high-level parallelism where needed. The compiler handles machine-dependent optimizations for efficient execution. The resulting programs are easily maintained and portable.

Compare this with the task of programming existing parallel machines. Scientists must rewrite their programs in a programming language that explicitly reflects the underlying machine architecture. Options include a message-passing dialect for *multiple-instruction, multiple-data* (MIMD) distributed-memory machines, extended vector & array syntax for *single-instruction, multiple-data* (SIMD) machines, and an explicitly parallel dialect with synchronization for MIMD shared-memory machines. Scientists must also wrestle with machine-specific issues such as improving data locality to take advantage of the memory hierarchy. Even after successfully developing, debugging, and optimizing the resulting parallel program, there is little assurance that it can be easily modified or ported to a different parallel machine. Scientists are thus discouraged from utilizing parallel machines because they risk losing their investment whenever the program changes or a new architecture arrives.

Programming MIMD distributed-memory machines is a particularly difficult process. Processors have separate address spaces, and inter-processor data movement takes place through calls to machine-specific communication libraries. Users must thus write message-passing Fortran 77 programs that deal with address translation, synchronizing processors, and communicating data using messages. The process is time-consuming, tedious, and error-prone. Significant blowups in source code size are not only common but expected. Parallel computers are thus not likely to be widely used until they are easier to program in a machine-independent manner without sacrificing efficiency.

1.1.2 Compiler Assistance

The goal of this thesis is to solve the parallel programming problem by developing the compiler technology needed to establish a machine-independent programming model. It must be easy to use yet perform with acceptable efficiency on different parallel architectures. In particular, we focus on *data-parallel* scientific codes that apply identical operations across large data sets, since these applications easily scale up to take advantage of massive parallelism [101].

One approach would be to identify a *data-parallel* programming style for Fortran that may be compiled to execute efficiently on a variety of parallel architectures. However, researchers working in the area, including ourselves, have concluded that such a programming style is needed but not sufficient in general. The reason is parallel programming is a difficult task in which many tradeoffs must be weighed. In converting from a Fortran program, the compiler simply is not able to always do a good job of picking the best alternative in every tradeoff, particularly since it must work solely with the text of the program. As a result, the programmer may need to add additional information to the program for it to be correctly and efficiently parallelized.

But in accepting this conclusion, we must be careful not to give up prematurely on the goal of supporting machine-independent parallel programming. In other words, if we extend Fortran to include information about the parallelism available in a program, we should not make those extensions dependent on any particular parallel machine architecture. From the compiler technologist's perspective, we need to find a suitable language for expressing parallelism and compiler technology that will translate this language to efficient programs on different parallel machine architectures.

1.1.3 Parallel Architectures and Programming Models

We begin our search for a suitable programming model by examining existing paradigms for parallel programming. Figure 1.1 depicts four different machine types and the dialect of Fortran commonly used for programming on each of them: Fortran 77 for the sequential machine, Fortran 90 for the SIMD parallel machine (*e.g.,* the TMC CM-2, MasPar MP-1), message-passing Fortran for the MIMD distributed-memory machine (*e.g.,* the Intel iPSC/860, Intel Paragon, Thinking Machines CM-5) and Parallel Computing Forum (PCF) Fortran [136, 166] for the MIMD shared-memory machine (*e.g.,* the Cray Research C90 Y-MP, BBN TC2000 Butterfly). Each of these languages seems to be a plausible candidate for use as a machine-independent parallel programming model.

Research on automatic parallelization has already shown that Fortran 77 is unsuitable for general parallel programming. However, message-passing Fortran looks like a promising candidate—it should be easy to implement a run-time system that simulates distributed memory on a shared-memory machine by passing messages through shared memory. Unfortunately, most scientific programmers reject this alternative because programming in message-passing Fortran is difficult and tedious. In essence, this would be reduction to the lowest common denominator—programming every machine would be equally hard.

Starting with PCF Fortran is more promising. It seems plausible that we might be able to use the parallel loops in PCF Fortran to indicate which data structures should be partitioned across the processors—data arrays accessed on different iterations of a parallel loop should probably be distributed. So what is wrong with starting from PCF Fortran? The problem is that the language is nondeterministic. If the programmer inadvertently accesses the same location on different loop iterations, the result can vary for different execution schedules. Hence PCF Fortran programs will be difficult to develop and require complex debugging systems.

Fortran 90 is more promising, because it is a deterministic language. The basic strategy for compiling it to different machines is to block the multidimensional vector operations into submatrix operations, with different submatrices assigned to different processors. We believe that this approach has a good chance of success. However, there are questions about the generality of this strategy. SIMD machines are not yet viewed as general-purpose parallel computers. Hence, the programs that can be effectively represented in Fortran 90 may be only a strict subset of all interesting parallel programs. We would still need some way to express those programs that are not well-suited to Fortran 90.

More importantly, we find that selecting a data decomposition is one of the most important intellectual steps in developing data-parallel scientific codes. Though many techniques have been developed for automatic data decomposition, we feel that the compiler will not be able to choose the most efficient data decomposition

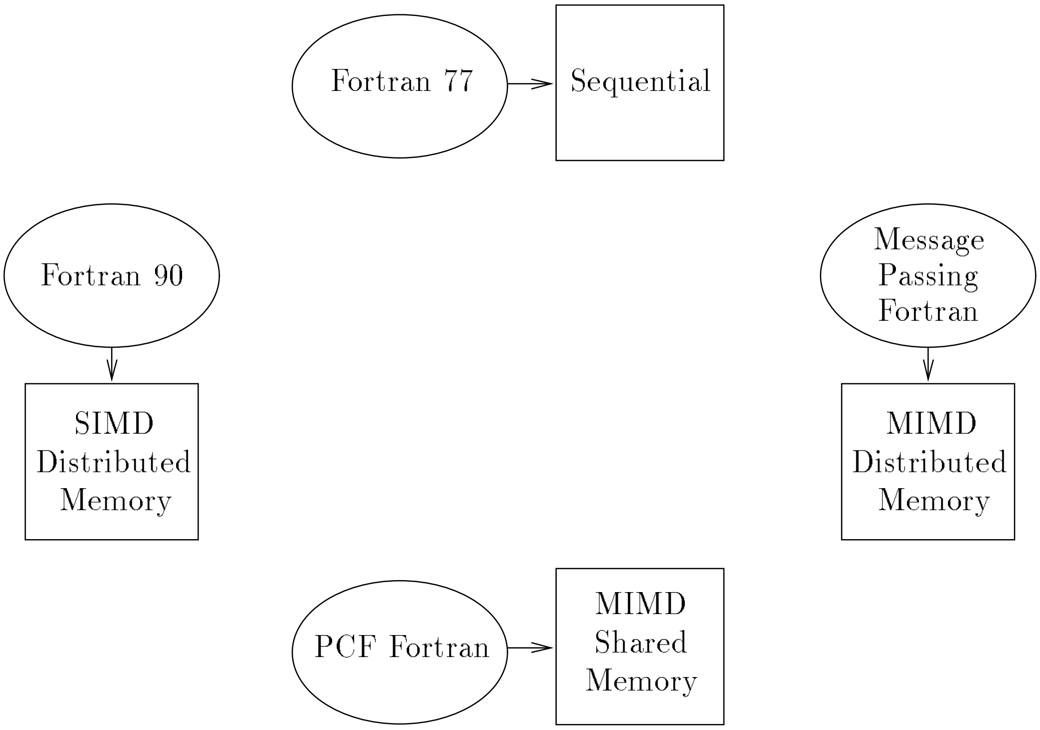


Figure 1.1 Fortran Dialects and Machine Architectures

for all programs. To be successful, the compiler needs additional information not present in Fortran 77, Fortran 90, or PCF Fortran. This conviction forms the basis for our thesis.

1.2 Thesis

We believe that a fundamental ingredient required for compiling programs to distributed-memory machines is a specification of the data decomposition of the program. Our thesis is that:

> An advanced compiler can generate efficient parallel programs for MIMD distributed-memory machines, *if* data decompositions are provided for well-written data-parallel numeric programs.

In other words, we believe that when data decompositions are provided for sequential programs written in a data-parallel programming style, an advanced compiler can automatically generate parallel programs that execute efficiently on MIMD distributed-memory machines.

1.3 Fortran D

Unfortunately, most current parallel programming languages concentrate on constructs to express parallelism; they provide little support for data decomposition [165]. For these reasons, we have developed an enhanced version of Fortran that introduces data decomposition specifications. We call the extended language Fortran D, where "D" suggests data, decomposition, or distribution.

As shown in Figure 1.2, we believe that if a Fortran D program is written in a data-parallel programming style with reasonable data decompositions, it can be implemented efficiently on a variety of parallel architectures. We should note that our goal in designing Fortran D is not to support the most general data decompositions possible. Instead, our intent is to provide data decompositions that are both powerful

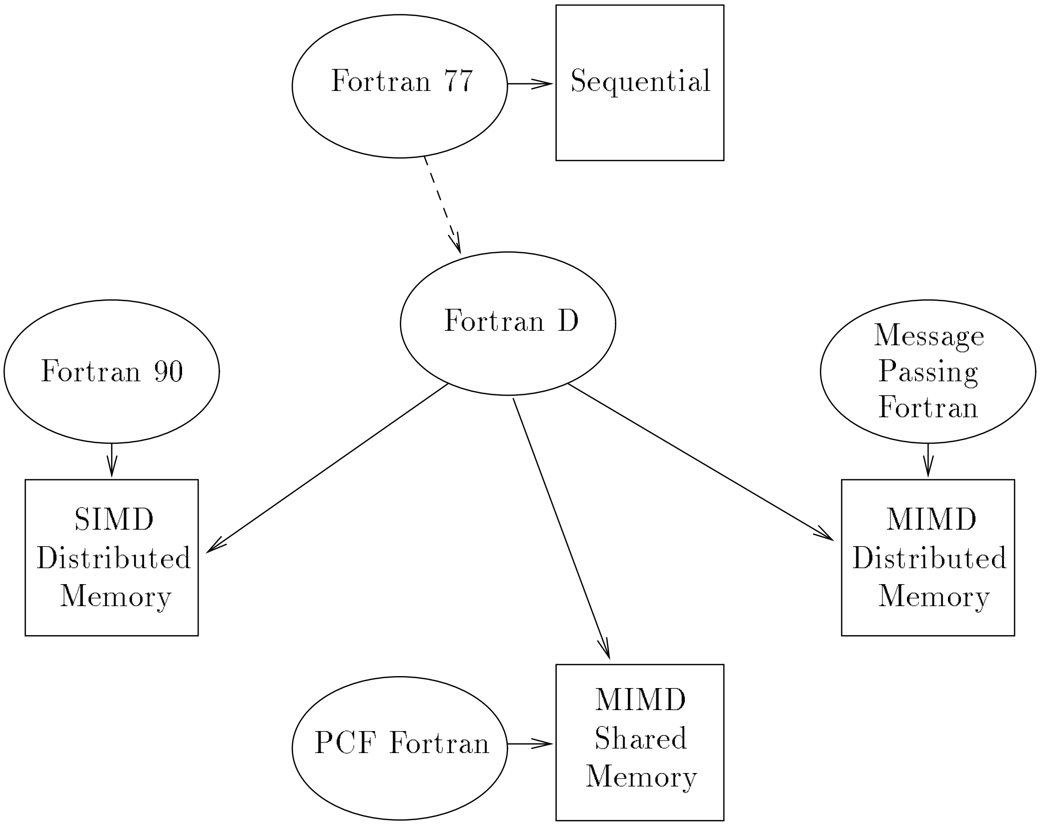


Figure 1.2 Machine-Independent Programming Strategy Using Fortran D

enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs.

A Fortran D program is a Fortran program augmented with a set of data decomposition specifications. If these specifications are ignored the program can be run without change on a sequential machine. Hence, the meaning of the program is exactly the meaning of the Fortran program contained within it—the specifications do not affect the meaning, they simply advise the compiler. Compilers for parallel machines can use the specifications not only to decompose data structures but also to infer parallelism, based on the principle that only the owner of a datum computes its value. In other words, the data decomposition also implicitly specifies the distribution of the work in the program.

1.4 Contributions

Despite its importance, a language such as Fortran D merely selects one avenue to machine-independent parallel programming. Only the development of advanced compilation technology can make this approach truly efficient and practical. The core of this thesis is devoted to demonstrating that Fortran D programs written in a data-parallel programming style can be *compiled* into efficient parallel programs for MIMD distributed-memory machines. It makes contributions in three areas: new and extended compilation techniques, experimental evaluation and verification of several design decisions, and implementation and empirical validation of a prototype Fortran D compiler.

The Fortran D compiler is distinguished by its reliance on advanced compile-time analysis and optimization to generate efficient programs. In comparison, most other distributed-memory compilers rely on extensive language features or run-time support, placing greater strain on the programmer and run-time system. The Fortran D compiler also directly generates the optimized program, rather than first generating a naive program and applying optimizations through a series of program transformations.

1.4.1 New Compilation Techniques

The Fortran D compiler incorporates a number of novel compilation techniques that are important for efficient generation of high-performance code for distributed-memory machines. They include:

- The DECOMPOSITION or TEMPLATE language feature, which allows users to map different arrays to the same logical group (Chapter 2).
- *Vector message pipelining*, a technique that combines message vectorization and message pipelining to hide communication overhead (Chapter 5).
- Detecting *pipelined* computations via *cross-processor loops*. Exploiting pipeline parallelism while balancing communication costs through *coarse-grain pipelining* (Chapter 5).
- Guiding communication and parallelism optimizations using cost models for communication and computation (Chapter 6).
- Strategies for efficient one-pass interprocedural compilation and interprocedural optimization of communication, partitioning, and dynamic data decomposition (Chapter 7).

As the Fortran D compiler is a second generation research project, many of the compilation techniques and optimizations found in the Fortran D compiler have been discussed or implemented by other researchers. However, previous researchers tend to apply each technique in isolation, without evaluating their effectiveness or considering their interaction with other elements of the compiler. In the Fortran D compiler we adapt, integrate, and extend a number of compilation techniques, including the following:

- Combine ALIGN and DISTRIBUTE language features to provide data decomposition for both SIMD and MIMD systems (Chapter 2).
- Extend the FORALL language feature to handle multiple statements and reductions through deterministic merge of multiple writes to the same location (Chapter 2).
- A formal compilation model based on the *owner computes* rule. The model employs translations of index and iteration sets through invertible *distribution functions* derived from data decomposition specifications (Chapter 3).
- Algorithms to partition data and computation at compile-time and generate code to efficiently instantiate the partition at run-time (Chapter 4).
- Extend *message vectorization* from a method of extracting element-wise messages out of loop nests into a complete code-generation strategy for guiding communication placement (Chapter 4).
- Introduce *message coalescing* and *message aggregation*, enhancements to message vectorization that reduce the number of messages generated (Chapter 5).
- Adapt methods for relaxing the owner computes rule for private variables and reductions detected by the dependence analyzer (Chapter 5).
- Generalize the application of *iteration reordering* to complete loop nests and individual statements in order to enhance *unbuffered* messages (Chapter 5).
- Discover new profitability criteria for applying program transformations such as loop interchange, fusion, distribution, and strip-mining in the context of distributed-memory machines (Chapter 5).
- Design a unified Fortran 77D and 90D compilation framework by integrating loop fusion, partitioning, and scalarization (Chapter 8).

1.4.2 Experimental Evaluation and Verification

In addition to developing new compilation techniques, we performed empirical experiments to verify some design choices. These experiments are used to:

- Evaluate the impact of individual optimizations on overall performance, as well as interactions between optimizations. Optimizations are classified, integrated in an overall optimization scheme, and ranked in order of effectiveness (Chapter 6).
- Verify the importance of interprocedural optimization using DGEFA, a linear algebra computation in Linpack (Chapter 7).
- Establish the need for a unified compilation framework for both Fortran 77D and 90D. First, determine the importance of loop fusion and pipelining for ADI integration, a technique for solving partial difference equations. Second, measure the performance of scalarization optimizations such as loop fusion and data prefetching (Chapter 8).

1.4.3 Implementation and Validation

Finally, we validated the overall effectiveness of our approach by designing and implementing a prototype Fortran D compiler for MIMD distributed-memory machines. We evaluated the compiler in two ways, by comparing the performance of its output code against:

- Hand-optimized message-passing kernels and programs on the Intel iPSC/860 (Chapter 9).
- Kernels and programs compiled by the CM Fortran compiler for the TMC CM-5 (Chapter 9).

These results proved very helpful in assessing prospects for automatic parallelization for distributed-memory machines.

1.5 Overview

We close this chapter by providing an overview of this thesis. Our research focuses on four areas—the Fortran D language, basic compilation, advanced optimizations, interprocedural compilation. In addition, we also present a unified compilation framework for both Fortran 77D and 90D, and elements of a complete Fortran D programming system. Other compiler projects will establish that Fortran D can also be compiled onto SIMD and shared-memory machines. The Fortran D compiler is designed to support both regular and irregular computations, but for this thesis we concentrate on compiler techniques for regular computations.

We are implementing the prototype Fortran D compiler in the context of the ParaScope programming environment in order to utilize its analysis and transformation capabilities [35, 117]. The prototype compiler automatically derives from the data decomposition node programs for MIMD distributed-memory machines; its goal is to minimize both load imbalance and communications costs. This thesis describes the design, implementation, and evaluation of the prototype Fortran D compiler. Here we present a brief overview of the remainder of the thesis.

Fortran D Language

Fortran D is a version of Fortran enhanced with data decomposition specifications. It is designed to support two fundamental stages of writing a data-parallel program: *problem mapping* using sophisticated array alignments, and *machine mapping* through a rich set of data distribution functions. The DECOMPOSITION statement declares an abstract problem or index domain. The ALIGN statement maps each array element onto the decomposition. The DISTRIBUTE statement groups elements of the decomposition and aligned arrays, mapping them to the parallel machine. Each dimension is distributed in a block, cyclic, or block-cyclic manner. Because the alignment and distribution statements are executable, dynamic data decomposition is possible. The FORALL loop specifies parallelism in a deterministic manner.

Compilation Model

The Fortran D compiler utilizes a code generation strategy based on the "owner computes" rule—where each processor only computes values of data it owns. Fortran D data decomposition specifications are translated into mathematical *distribution functions* that determine the ownership of local data. By composing these with subscript functions or their inverses, the Fortran D compiler can partition the computation and determine nonlocal accesses at compile-time. This information is used to generate the program for execution on the nodes of the distributed-memory machine.

Basic Compilation

The basic structure of the Fortran D compiler is organized around three major functions—program analysis, program optimization, and code generation. New analysis techniques are required to compile shared-memory programs for distributed memory machines. Internal data structures used in the compilation process are described. The Fortran D compiler utilizes a compilation strategy based on the concept of *data dependence* [130] that unifies and extends previous techniques. The major step of the compilation process are:

1. **Analyze Program.** Symbolic and data dependence analysis is performed.
2. **Partition data.** Fortran D data decomposition specifications are analyzed to determine the decomposition of each array in a program.
3. **Partition computation.** The compiler partitions computation across processors using the "owner computes" rule.
4. **Analyze communication.** Based on the work partition, references that result in nonlocal accesses are marked.
5. **Optimize communication.** Nonlocal references are examined to determine optimization opportunities. The key optimization, message vectorization, uses the level of loop-carried true dependences to combine element messages into vectors [16, 212].
6. **Manage storage.** "Overlaps" [212] or buffers are allocated to store nonlocal data.
7. **Generate code.** The compiler *instantiates* the communication, data and work partition determined previously, generating a Fortran 77 program with explicit message-passing. This *single-program multiple-data* (SPMD) program can then be compiled and executed directly on the nodes of the distributed-memory machine [111].

In the Fortran D compiler, collections of data and work are referred to as *index sets* and *iteration sets*, respectively. Both are represented by *regular section descriptors* (RSDs) [98], which we describe using Fortran 90 triplet notation.

Compiler Optimizations

A number of Fortran D compiler optimizations are introduced and classified. Program transformations modify the program execution order to enable optimizations. Communication optimizations can be separated into two classes, those that reduce communication overhead by decreasing the number of messages, and those that hide communication overhead by overlapping the cost of remaining messages with local computation. Parallelism optimizations restructure the computation or communication to increase the amount of useful computation that may be performed in parallel. Optimizations improve computation partitioning by eliminating explicit guards. Storage requirements are reduced by partitioning data across processors and sending messages in smaller blocks.

Evaluation of Optimizations

Communication and parallelism optimizations are analyzed and empirically evaluated for stencil computations. Profitability formulas are derived for each optimization. Results show that exploiting parallelism for pipelined computations, reductions, and scans is vital. Message vectorization, coarse-grain pipelining, and collective communication significantly improve performance because they eliminate large numbers of messages. Remaining optimizations help hide communication overhead by overlapping communication with computation.

Interprocedural Compilation

Algorithms for compiling Fortran D for MIMD distributed-memory machines are significantly restricted in the presence of procedure calls. Interprocedural analysis, optimization, and code generation algorithms for Fortran D are presented that limit compilation to only one pass over each procedure. This is accomplished by collecting summary information after edits, then compiling procedures in reverse topological order to propagate necessary information. Delaying instantiation of the computation partition, communication, and dynamic data decomposition is key to enabling interprocedural optimization. Empirical results show that this can be crucial in achieving acceptable performance for common applications.

Fortran 77D and 90D

An integrated approach to compiling Fortran 77D and Fortran 90D programs is presented. The integrated Fortran D compiler relies on two key observations. First, array constructs may be *scalarized* into FORALL loops without loss of information. Second, *loop fusion*, *partitioning*, and *sectioning* optimizations are essential for both Fortran D dialects.

Preliminary Experiences

Case studies are used to illustrate its strengths and weaknesses of the prototype Fortran D compiler. When compared against hand-optimized message-passing code on the Intel iPSC/860, the output of the prototype is slower than hand-optimized kernels by 50–100% due to unimplemented optimizations. However, it closely matches the performance of parallel stencil programs since computation dominates execution time. By making more decisions at compile-time, the Fortran D compiler outperforms the CM Fortran compiler by a factor of 2–17 on the Thinking Machines CM-5. Despite these successes, the prototype requires better symbolic analysis, greater flexibility, and improved optimization of linear algebra and pipelined codes to be truly useful. Overall, we find that the success of automatic parallelization is dependent on the amount of communication and parallelism inherent in the application.

Programming System

We describe the elements of a complete Fortran D programming system. A static performance estimator based on training sets provides machine-dependent details that fine-tune the compilation process. The automatic data partitioner derives Fortran D data decomposition specifications based on the original Fortran program. A shared-memory parallelizing and vectorizing compiler exploits multiple processors on a single node. The data locality optimizer restructures the node program for improved use of the memory hierarchy, backed up by an optimizing scalar compiler. The programming environment provides program profiling, performance measurement and visualization, as well as support for debugging and accepting user feedback. A portable lightweight communication library improves performance.

Chapter 2

Fortran D Language

Fortran D is a version of Fortran enhanced with data decomposition specifications. It is designed to support two fundamental stages of writing a data-parallel program: *problem mapping* using sophisticated array alignments, and *machine mapping* through a rich set of data distribution functions. We believe that Fortran D provides a simple machine-independent programming model for most data-parallel computations.

2.1 Introduction

High-level parallel languages such as Delirium [147], Linda [41], and Strand [70] are valuable when used to *coordinate* coarse-grained functional parallelism. However, these languages do not meet the needs of computational scientists because they do not elegantly describe data-parallel computations of the type described by Hillis and Steele [101] and Karp [111]. Parallelism must be explicitly specified because these languages do not provide compilers that can automatically detect and exploit parallelism. In addition, these languages also lack both language and compiler support to assist in efficient *data placement*, the partitioning and mapping of data to individual processors [165].

To overcome this deficiency, we have designed Fortran D, a version of Fortran enhanced with a rich set of data decomposition specifications. Fortran D is targeted at data-parallel numeric applications that are not supported by existing parallel languages. The extensions proposed in Fortran D are compatible with both Fortran 77 and Fortran 90, a version of Fortran with explicit manipulation of high-level array structures. Fortran 90D can be viewed as a refinement of CM Fortran [196] consistent with a parallel Fortran 77.

We consider Fortran D to be one of the first of a new generation of *data-placement* programming languages. Its design was inspired by the observation that modern high-performance architectures demand that careful attention be paid to data placement by both the programmer and compiler. As one measure of its relevance, we note that many features from Fortran D have been adopted by High Performance Fortran (HPF), a new proposed Fortran standard [99]. HPF is being used by Cray Research, DEC, IBM, and Thinking Machines for programming their newest generation of parallel machines.

Our goal with Fortran D is to provide a simple yet efficient machine-independent parallel programming model. By shifting much of the burden of machine-dependent optimization to the compiler, we allow the programmer to easily write data-parallel programs that can be compiled and executed with good performance on a variety of parallel architectures.

We believe that Fortran D is powerful enough to express most fine-grain parallel computations, but also simple enough that a sophisticated compiler can produce efficient programs for different parallel architectures. In particular, Fortran D is well suited for supporting compiler techniques for automatic data decomposition and communication generation, two crucial problems in programming distributed-memory machines. Fortran D programs also have the advantage of being deterministic, unlike programs written in most explicitly parallel languages.

This chapter presents the design of Fortran D, especially its strategy for expressing data parallelism and mapping it to the underlying parallel architecture. A number of language issues are discussed and solutions presented. Language features for partitioning data are crucial. Because of the Fortran D compilation model, they ultimately determine the shape of the resulting code by defining the computation partitioning and the resulting inter-processor communication. We start by describing our view of data parallelism.

2.2 Data-Parallel Programming Model

The data decomposition problem can be approached by noting that there are two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is dependent on the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. Data distribution provides opportunities to reduce data movement, but must also maintain load balance. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D requires the user to specify data decompositions in terms of these two levels of data parallelism. First, the ALIGN statement is used to describe a problem mapping. Second, the DISTRIBUTE statement is used to map the problem and its associated arrays to the physical machine. We believe that our two phase strategy for specifying data decomposition is natural for the computational scientist, and is also conducive to modular, portable code. Previous projects also include a third intermediate level of parallelism representing a coarse-grain "virtual machine". We do not think this is necessary for our work, although it may be helpful for explicit message-passing programs.

2.3 N$PROC

Fortran D reserves the variable N$PROC to indicate the number of processors available. It may be evaluated at run-time or passed as a compile-time option to the compiler.

2.4 DECOMPOSITION Statement

In Fortran D, the DECOMPOSITION statement may be used to declare a name for each problem mapping. Arrays in the program are mapped to the decomposition with the ALIGN statement. The result represents an abstract high level specification of the fine-grain parallelism of a problem. There may be multiple decompositions representing different problem mappings, but an array may be mapped to only one decomposition at a time. All scalars and arrays not mapped to a decomposition are allocated locally.

Decompositions are designed to enable users to easily group data arrays associated with solving a single problem. The decomposition statement declares the name, dimensionality, and size of a decomposition for later use. A decomposition is simply an abstract problem or index domain. No storage is allocated for a decomposition.

```
DECOMPOSITION A(N)
DECOMPOSITION B(N,N)
```

In this example, A is declared as an one-dimensional decomposition of size N, with elements indexed from 1 to N. B is a two-dimensional $N \times N$ decomposition.

We have also adopted a feature proposed in High Performance Fortran back into Fortran D. We have made the DECOMPOSITION statement optional, allowing arrays to be aligned or distributed directly. We do this by assuming arrays in Fortran D are implicitly aligned with a decomposition of the same size and shape. Any Fortran D statements applied to an array affects the underlying decomposition in the standard way.

2.5 ALIGN Statement

The ALIGN statement is used to map arrays with respect to a decomposition. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or between dimensions.

The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition. I, J, K, etc... are canonical placeholders indicating the location of dimensions in a decomposition. Array subscripts are fixed; they always consist of the placeholders in alphabetical order beginning with I. The decomposition subscripts can be functions of the placeholders; they specify the alignment of the array with respect to the decomposition.

2.5.1 Exact Match

The simplest alignment occurs when the array is exactly mapped onto the decomposition. In the following example, the arrays X1 and X2 are mapped exactly onto the equivalent dimensions in the decompositions A and B.

```
REAL X1(N), X2(N,N), X3(N,N)
DECOMPOSITION A(N), B(N,N)
ALIGN X1(I) with A(I)
ALIGN X2(I,J) with B(I,J)
ALIGN X3(I,J) with B(I,J)
```

For convenience, placeholders are not required where the mapping is exact. Multiple arrays may also be aligned with the same statement. For instance, the alignments in the previous example could also have been specified with the following syntax.

```
REAL X1(N), X2(N,N), X3(N,N)
DECOMPOSITION A(N), B(N,N)
ALIGN X1 with A
ALIGN X2, X3 with B
```

2.5.2 Intra-dimension Alignment

Intra-dimension alignment determines the data decomposition within each dimension. This section describes how offset and stride may be specified.

Alignment Offsets

In Fortran D, the user can specify an alignment offset for any dimension of an array. The simplest case occurs when the array and decomposition have the same number of dimensions. Constants are added to the placeholders in the decomposition to indicate the offset in that dimension.

```
REAL X1(N), X2(N)
DECOMPOSITION A(N)
ALIGN X1(I) with A(I+1)
ALIGN X2(I) with A(I-1)
```

In this example, X1 and X2 are aligned with respect to decomposition A by 1 and -1. X1(I) is thus always mapped to the same element of the decomposition as X2(I+2); *e.g.,* X1(1) is mapped together with X2(3).

```
REAL X3(N,N), X4(N,N)
DECOMPOSITION B(N,N)
ALIGN X3(I,J) with B(I,J-1)
ALIGN X4(I,J) with B(I-1,J+2)
```

Similarly, in this example the alignment of X3 and X4 with respect to decomposition B means that X3(I,J) is mapped to the same element of B as X4(I+1, J-3).

Alignment Strides

Fortran D also allows a stride to be specified when performing intra-dimensional alignment. Alignment strides are used to determine the density of an array mapped to a dimension. They are introduced as

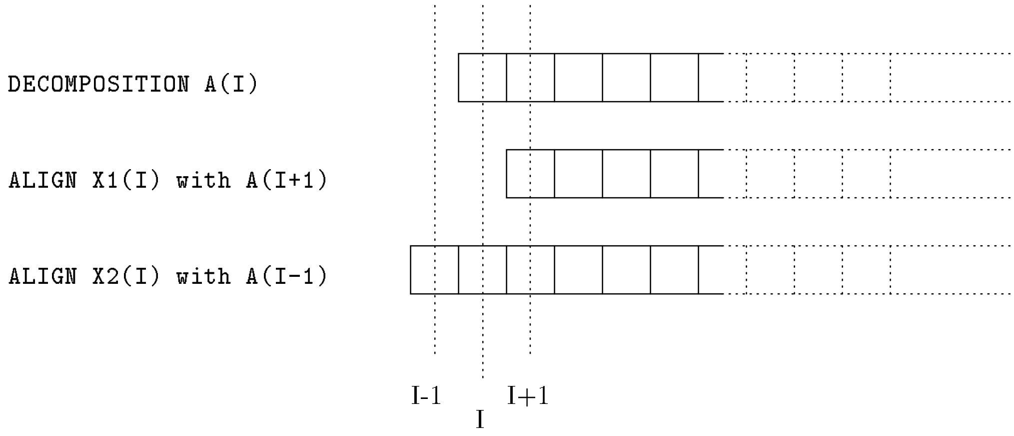


Figure 2.1 1-D Alignment Offsets

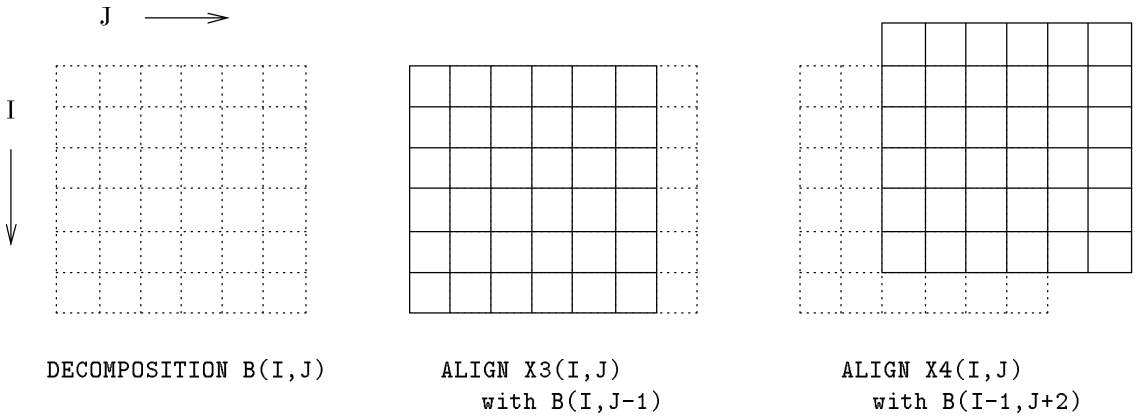


Figure 2.2 2-D Alignment Offsets

coefficients of placeholders in the subscript expressions of decompositions in an ALIGN statement. Strides may be also used in combination with offsets.

```
REAL X1(N), X2(N)
DECOMPOSITION A(N)
ALIGN X1(I) with A(2*I)
ALIGN X2(I) with A(2*I-1)
```

In this example, array X1 has a stride of 2 with respect to decomposition A. It is thus mapped to the even elements of A. Array X2 also has a stride of 2, but the alignment offset of -1 causes it to be mapped to every odd element of A. Alignment strides are easily extended to higher order arrays and decompositions, as in the following example.

```
REAL X1(N,N), X2(N,N+N)
DECOMPOSITION B(N,N)
ALIGN X1(I,J) with B(2*I,2*J)
ALIGN X2(I,J) with B(2*I-1,J)
```

Alignment strides with negative values are also allowed; they correspond mapping the reflection of the array dimension onto the decomposition.

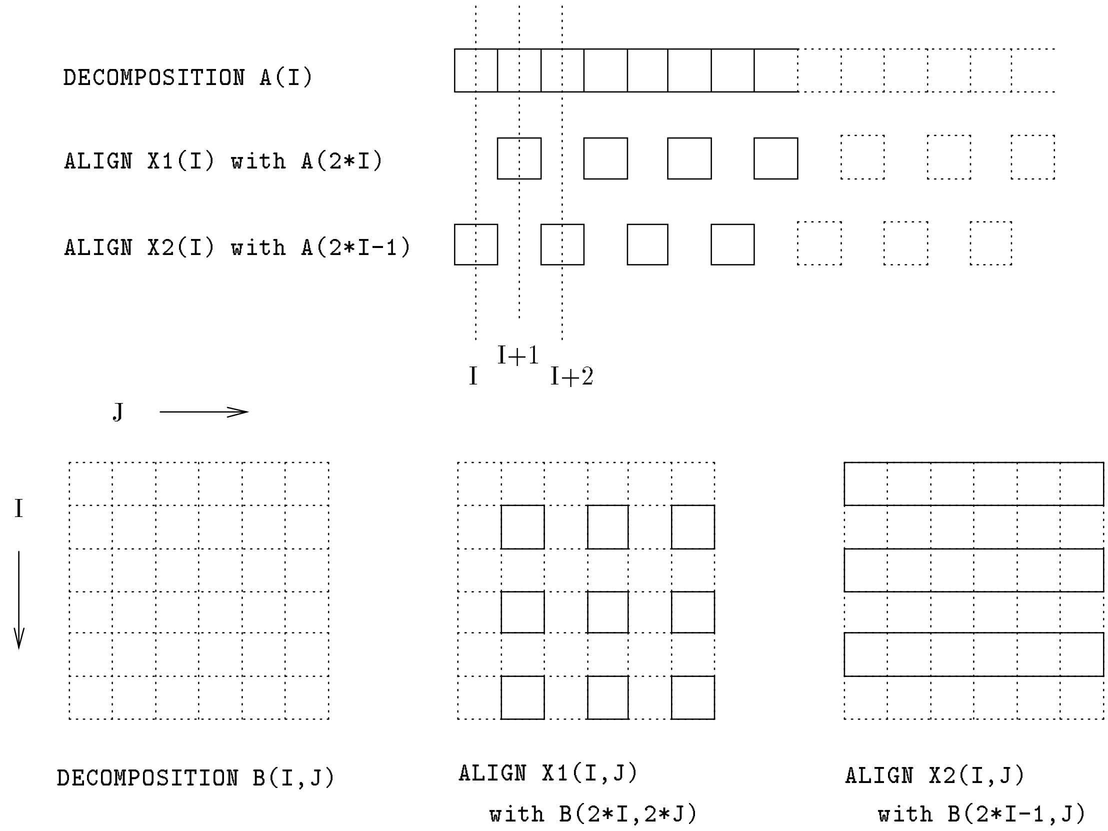


Figure 2.3 Alignment Stride

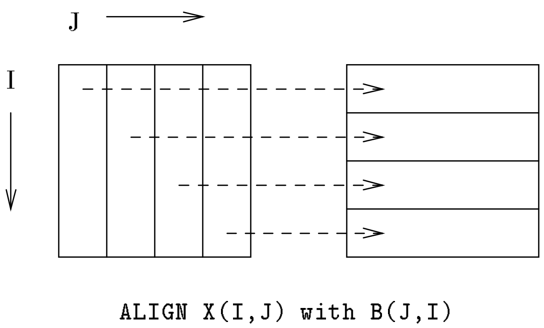


Figure 2.4 Alignment Permutation

2.5.3 Inter-dimension Alignment

Inter-dimension alignment determines the data decomposition between dimensions. This section describes how permutation, collapse, and embedding may be specified.

Permutation

In Fortran D, the user can arbitrarily permute the dimensional alignment between arrays and decompositions. A common application would be to perform array transpositions. Canonical placeholders must be used to mark the aligned dimensions.

```
REAL X1(N,N), X2(N,N,N)
DECOMPOSITION B(N,N), C(N,N,N)
ALIGN X1(I,J), with B(J,I)
ALIGN X2(I,J,K), with C(K,I,J)
```

In this example, the transpose of X1 is mapped to the decomposition B, as indicated by the reversed placeholders I and J. Similarly, the third and first dimensions of X2 are mapped to the first and second dimensions of decomposition B.

Collapse

It is sometimes convenient to ignore certain dimensions of the array when mapping an array to a decomposition. All data elements in the unassigned dimensions are collapsed and mapped to the same location in the decomposition. An array dimension may be collapsed in the ALIGN statement simply by excluding its placeholder from the decomposition subscripts, as this demonstrates that the dimension has no effect on the actual alignment.

```
REAL X1(N,N), X2(N,N), X3(N,N,N), X4(N,N,4)
ALIGN X1(I,J) with A(I)
ALIGN X2(I,J), X3(I,J,K) with A(J)
ALIGN X4(I,J,K) with B(I,J)
```

In this example, the first dimension of array X1 is mapped onto the decomposition A. The second dimension of X1 is collapsed and stored on the same processor. In other words, each row of X1 is mapped to an individual element in decomposition A. Similarly, each column of array X2 is mapped to A. For array X3, the second dimension is mapped onto the decomposition A, with the first and third dimensions local. Array collapse frequently occurs when an array dimension is used to store multiple data fields per problem element, such as for array X4 in the example.

Embedding

Conversely, it may be necessary to map arrays with fewer dimensions onto the decomposition. In these cases it is necessary to specify both the mapping for each dimension of the array and the actual position of the array in the unmapped dimensions of the decomposition. This determines the embedding of the array in the decomposition.

```
REAL X1(N), X2(N), X3(N), X4(N)
ALIGN X1(I) with B(I,2)
ALIGN X2(I) with B(1,I)
ALIGN X3(I) with B(I-1,2)
ALIGN X4(I) with B(1,I+2)
```

In this example, array X1 is mapped to the first dimension of decomposition B, a column. It is necessary to specify the actual column position with a constant remaining unmapped dimension. In this case the constant "2" in the second dimension indicates that X1 should be mapped to the second column of decomposition B. Similarly, array X2 is mapped to the first row of decomposition B. In a more complex example, arrays X3 and X4 are both aligned and mapped to decomposition B.

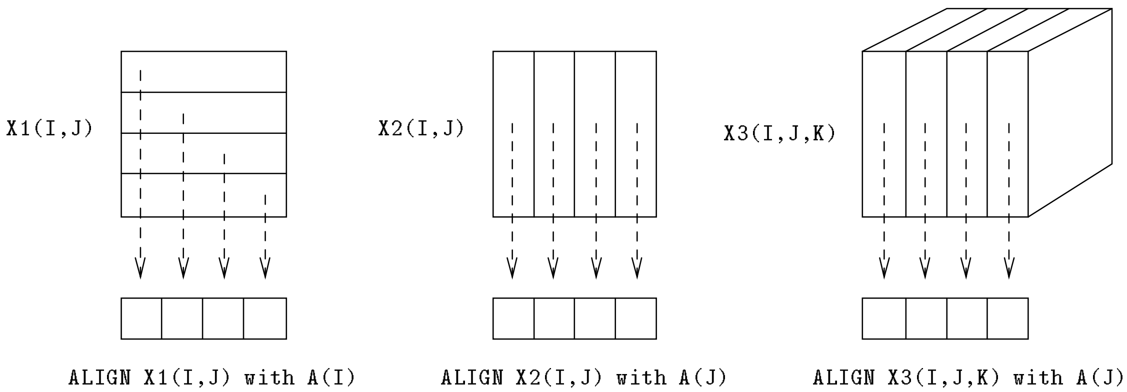


Figure 2.5 Array Collapse

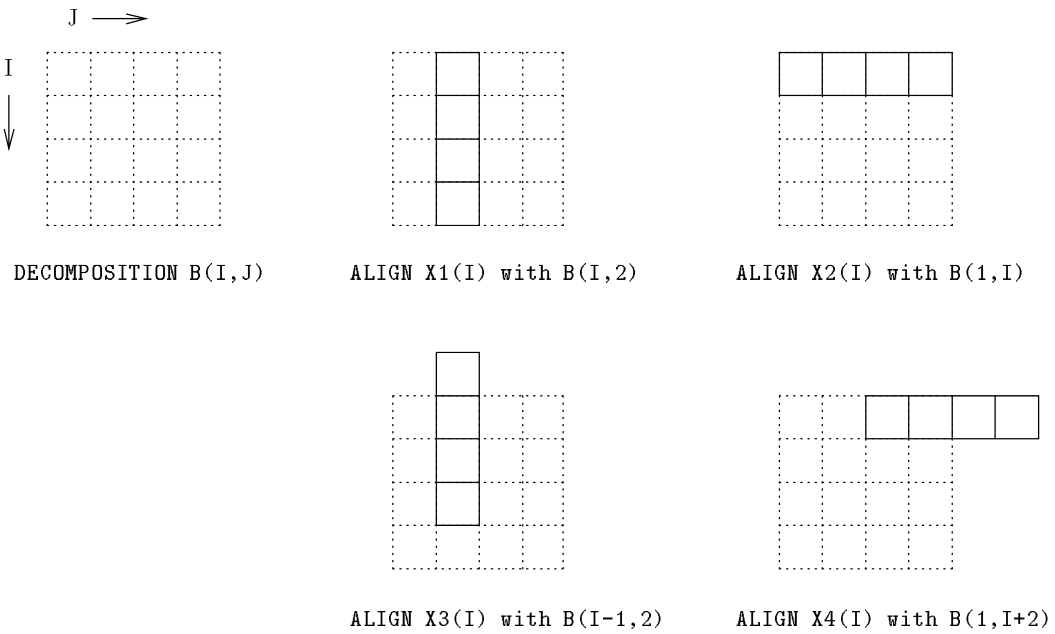


Figure 2.6 Array Embedding

This scheme can be extended to higher order arrays and decompositions. Note that when arrays are mapped only to part of a decomposition, the array may not be mapped to all the processors in the machine, depending on the actual distribution.

2.5.4 Combinations

The user can apply any combination of intra-dimensional and inter-dimensional alignments when mapping arrays to decompositions.

```
REAL X1(N,N), X2(N,N)
ALIGN X1(I,J) with B(J+2,I-1)
ALIGN X2(I,J) with B(4,I-2)
```

In this example, array X1 is both aligned and transposed with respect to decomposition B. Array X2 is collapsed into its first dimension (forming a single column), mapped to the fourth row of decomposition B, and aligned by -2.

2.5.5 Alignment Options

The ALIGN statement also supports options to specify actions for overflows, mapping parts of arrays to a decomposition, and either totally or partially replicating arrays. These options are discussed in this section.

Array Overflow

It is possible that the array to be aligned does not fit completely within the decomposition, causing an *overflow*. In these cases, an *optional* `overflow` clause may be used to select one of three options, `ERROR`, `TRUNC`, and `WRAP`, described below.

The default choice, `ERROR`, considers elements overflowing the decomposition to be unmapped. Any attempt to access such elements will be considered to be an error. Alternatively, the user may choose to truncate the array with the `TRUNC` option. All elements overflowing the decomposition are then mapped to the element on the edge of the decomposition in that dimension. `WRAP`, the last choice, wraps overflowing array elements back to the opposite end of the decomposition. Systolic algorithms in particular may benefit from this feature.

```
REAL X1(N), X2(N), X3(N), X4(N,N,N)
DECOMPOSITION A(N), C(N,N,N)
ALIGN X1(I) with A(I-1)
ALIGN X2(I) with A(I-1) overflow (TRUNC)
ALIGN X3(I) with A(I-1) overflow (WRAP)
ALIGN X4(I,J,K) with C(I-1,J-1,K-1) overflow (ERROR,TRUNC,WRAP)
```

In the previous example, attempting to reference X1(1) would be illegal since it maps to the undeclared decomposition element A(0), which by default is defined as type `ERROR`. Because X2 is truncated, the array elements X2(1) and X2(2) map to the same decomposition element A(1). Wrapping X3 causes the array element X3(1) to map to the decomposition element A(N). The alignment statement for X4 shows how overflow options may be specified for multidimensional decompositions.

Restrictions A problem exists with detecting array overflow that occur when attempting to access array elements that have not been mapped to a decomposition. Most cases may be detected at compile-time, but irregular or complex computations require run-time support. To restrict complexity the Fortran D compiler may either provide a special compile-time option that generates code that assumes no array overflows, or an option that generates code to aid run-time detection of such overflows.

Array Range

By default, the ALIGN statement maps the entire array to the decomposition. However, Fortran D also allows just part of an array to be mapped onto a decomposition. This may be done by specifying a section of the

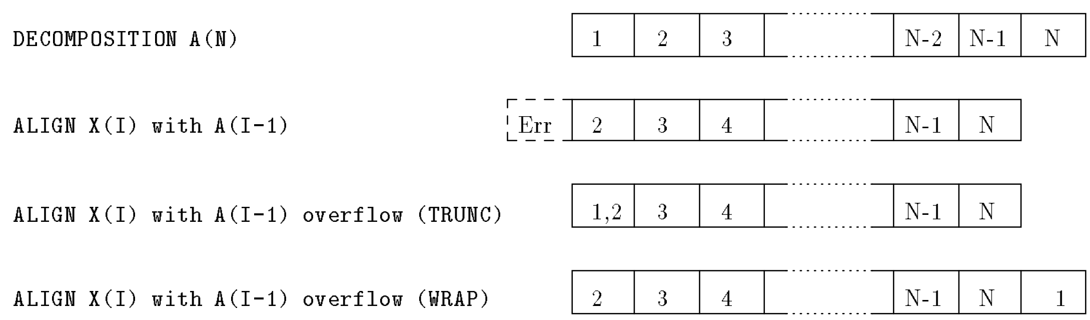


Figure 2.7 Array Overflow

array to be mapped using the RANGE clause. The RANGE clause specifies a range for each dimension of the form <from>:<to>. The : symbol indicates that the entire array dimension should be mapped. A subarray can thus be selected and aligned with a decomposition. This partial alignment feature is useful when one large work array is subdivided into several smaller logical arrays at run-time.

```
REAL X1(N+N), X2(N+N), X3(N+N,N+N), X4(N,N+N)
DECOMPOSITION A(N), B(N,N)
ALIGN X1(I) with A(I) range (1:N)
ALIGN X2(I) with A(I-N) range (N+1:N+N)
ALIGN X3(I,J) with B(I-N,J-N) range (N+1:N+N,N+1:N+N)
ALIGN X4(I,J) with B(I,J-N) range (:,N+1:N+N)
```

In the previous example, the RANGE clause is used to map elements 1 to N of array X1 to decomposition A and elements N+1 to 2N of array X2 to decomposition A, starting at decomposition element 1. Similarly, the subarray of X3 beginning at (N+1,N+1) is aligned with decomposition B. Finally, half of array X4 is aligned with decomposition B, with the : symbol indicating that the entire first dimension of X4 is mapped to the decomposition.

2.5.6 Replication

The ALIGN statement may also be used a means to replicate distributed variables in Fortran D. This can be done by assigning a range for a dimension rather than a position or placeholder. Ranges may be specified as <from>:<to>, or simply as : if the entire dimension is desired. If an assignment is made to a replicated value, all replicated values would be updated. Note that all variables not aligned to a decomposition are considered to be totally replicated on all processors. The compiler will label scalar and array variables as local, distributed, or replicated.

```
REAL X1(N), X2(N), X3(N)
DECOMPOSITION B(N,N)
ALIGN X1(I) with B(I,1:2)
ALIGN X2(I) with B(I,:)
ALIGN X3(I) with B(I-1:I,:)
```

In the first ALIGN statement in this example, a range from 1 to 2 is specified in the second dimension of B. This causes each of the first two columns of decomposition B to each get a copy of array X1, in effect replicating every element of X1 among the first five elements of each row of B. In the second ALIGN statement, the : symbol in the second dimension of decomposition B specifies that each element of array X2 is replicated across all elements of B in the same row. The two modes may also be combined, as in the third statement, where each row of B gets a copy of the element of array X3 in that row, as well the element of X3 from the previous row.

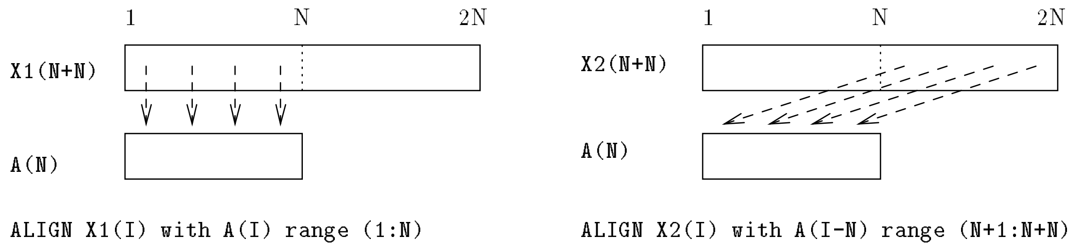


Figure 2.8 Array Range

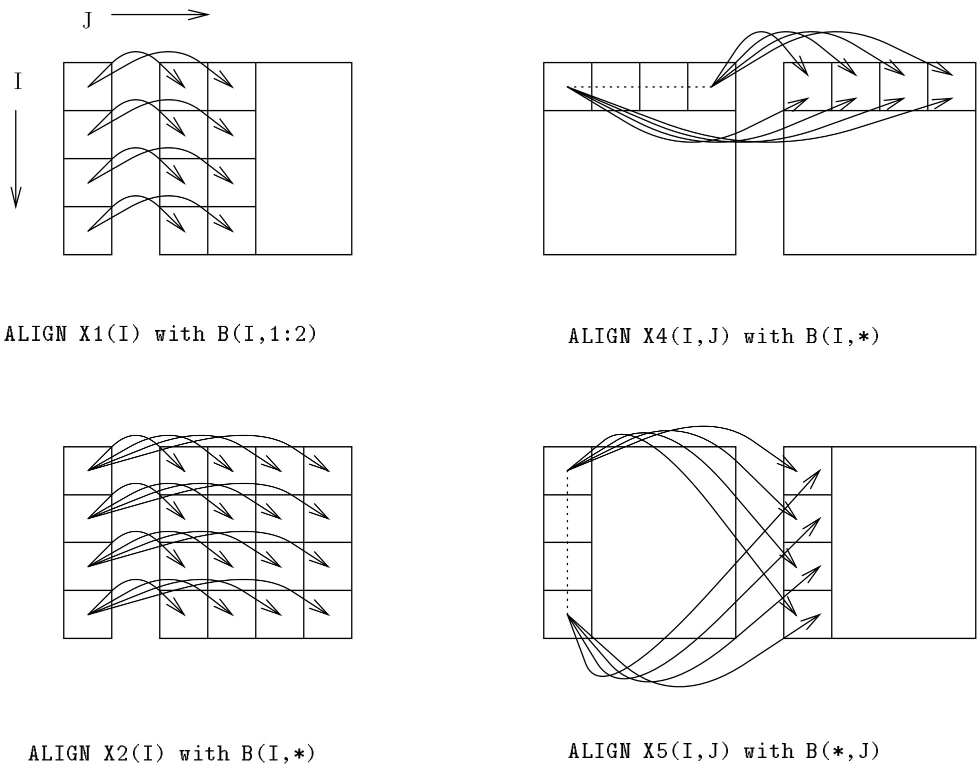


Figure 2.9 Array Replication

```
REAL X4(N,N), X5(N,N), X6(N,N)
DECOMPOSITION B(N,N)
ALIGN X4(I,J) with B(I,:)
ALIGN X5(I,J) with B(:,J)
ALIGN X6(I,J) with B(:,:)
```

Replication can also be extended to higher dimension arrays. In this example, the first ALIGN statement causes each row of array X4 to be mapped to each element in the corresponding row of decomposition B. Similarly, the second ALIGN statement causes each column of X5 to be mapped to each element in the corresponding column of B. Finally, each element of X6 is totally replicated for each element of decomposition B; *i.e.,* each processor is guaranteed to have a copy of X6. This is exactly the default case for unaligned arrays.

2.6 DISTRIBUTE Statement

In Fortran D, we use the DISTRIBUTE statement to specify the mapping of the decomposition to the physical parallel machine. The distribution selected will affect the ability of the compiler to minimize communications and load imbalance for the resulting program. Physical machine characteristics such as the number of processors, amount of memory per processor, and communication costs between processors must all be taken into account since they affect which distributions are feasible and efficient. Program characteristics such as the size of distributed arrays and computation structure may also be crucial in determining a good distribution.

In addition, data parallelism may either be regular or irregular. Regular parallelism can be effectively exploited through relatively simple data distributions. Irregular data parallelism, on the other hand, may require irregular data distributions and run-time preprocessing to manage the parallelism.

In Fortran D, a distribution specifies the machine mapping for exactly one decomposition. The compiler then applies the distribution to all the arrays mapped to the decomposition. The user does not need to specify a distribution for each array. It is illegal to access any element of a distributed array before it has been mapped to the machine with a DISTRIBUTE statement.

The DISTRIBUTE statement takes the name of a decomposition and assigns an *attribute* to each dimension of the decomposition. Each attribute describes the mapping of the data in that dimension of the decomposition. Attributes in each dimension are independent, and may specify regular or irregular distributions, as described in later sections. The symbol : is used to denote dimensions which are assigned locally; *i.e.,* these dimensions are not distributed.

```
DISTRIBUTE A(attribute)
DISTRIBUTE B(attribute, attribute)
```

In this example, the decompositions A and B are assigned an attribute for each dimension of the decomposition. Distributions in effect describe the assignment of data to an underlying processor array.

2.6.1 Regular Distributions

The three types of attributes for regular distributions in Fortran D are BLOCK, CYCLIC, and BLOCK_CYCLIC. Suppose there are P processors and N elements in a decomposition. We assume for simplicity that P divides N evenly. If this is not the case, the resulting distribution will be slightly unbalanced. The Fortran D distributions can then be described as follows:

- BLOCK divides the decomposition into contiguous chunks of size N/P, assigning one block to each processor.
- CYCLIC specifies a round-robin division of the decomposition, assigning every P^{th} element to the same processor. CYCLIC distributions are useful for load balancing.
- BLOCK_CYCLIC is similar to CYCLIC but takes a parameter M. It first divides the dimension into contiguous chunks of size M, then assigns these chunks in the same fashion as CYCLIC.

Only one attribute may be assigned for each dimension of the decomposition. However, multidimensional decompositions may have different combinations of distribution patterns. For these decompositions, processors are allocated as evenly as possible to each distributed dimension. This creates an implicit processor array. The following examples show some common Fortran D distributions.

```
DISTRIBUTE A(BLOCK), B(CYCLIC), C(BLOCK_CYCLIC(2))
DISTRIBUTE A(BLOCK,:), B(:,BLOCK), C(BLOCK,BLOCK)
DISTRIBUTE A(CYCLIC,:), B(:,CYCLIC), C(CYCLIC,CYCLIC)
DISTRIBUTE A(BLOCK_CYCLIC(2),:), B(:,BLOCK_CYCLIC(3))
DISTRIBUTE C(BLOCK_CYCLIC(2),BLOCK_CYCLIC(4))
DISTRIBUTE A(CYCLIC,BLOCK), B(BLOCK,CYCLIC), C(BLOCK,BLOCK_CYCLIC(2))
```

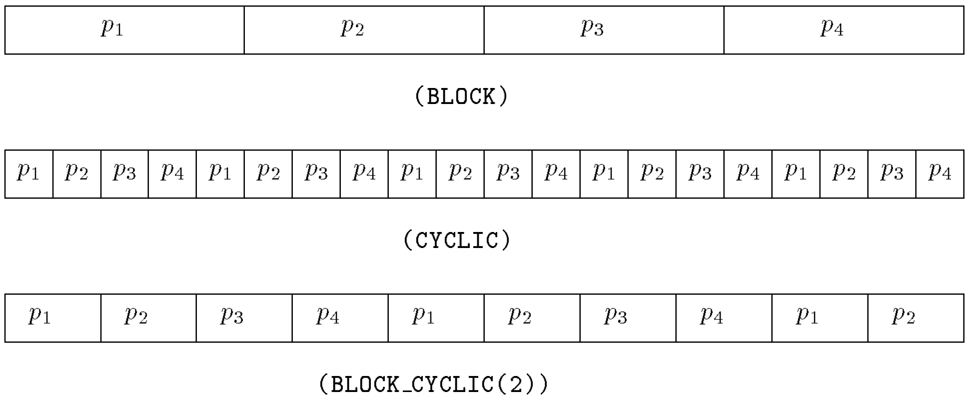


Figure 2.10 1-D Distributions

2.6.2 Processor Allocation

Fortran D also provides the capability of specifying processor allocations, where the allocation specifies the number of processors assigned to each dimension of the decomposition. Users can thus define their own uneven processor allocation, instead of using the even processor allocations the compiler generates by default.

Processor allocations are specified by adding an additional parameter indicating the number of processors for each dimension of the distribution. The multiplicand of the processors in each dimension must be less than or equal to the total number of processors defined by N$PROC, since Fortran D does not support virtual processors. If BLOCK_CYCLIC is passed two parameters, the first parameter specifies the block size and the second specifies the number of processors. The following are some examples of uneven distributions:

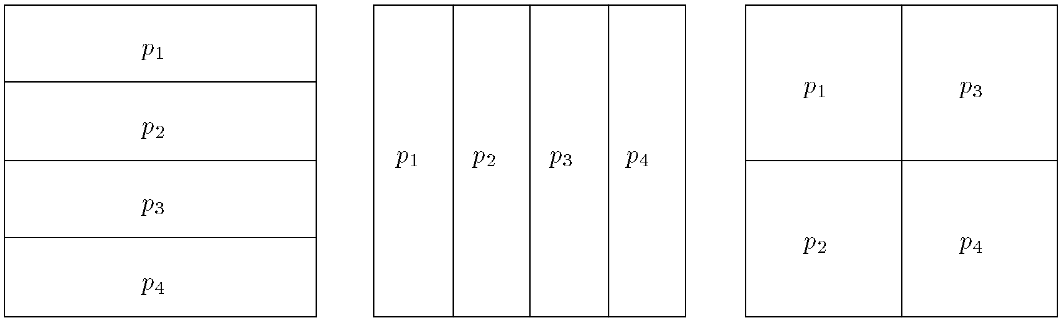
```
DISTRIBUTE A(BLOCK(4),BLOCK(2)), B(BLOCK(2),BLOCK(4))
DISTRIBUTE A(BLOCK(4),CYCLIC(2)), B(BLOCK(2),CYCLIC(4))
DISTRIBUTE C(CYCLIC(4),BLOCK(2)), D(BLOCK_CYCLIC(2,2),BLOCK(4))
```

2.6.3 Unsupported Distributions

Though Fortran D supports several regular distribution patterns, our intention is to keep the distribution attributes relatively simple to allow straightforward communications generation by the compiler. As a result, Fortran D distributions obey these simple rules:

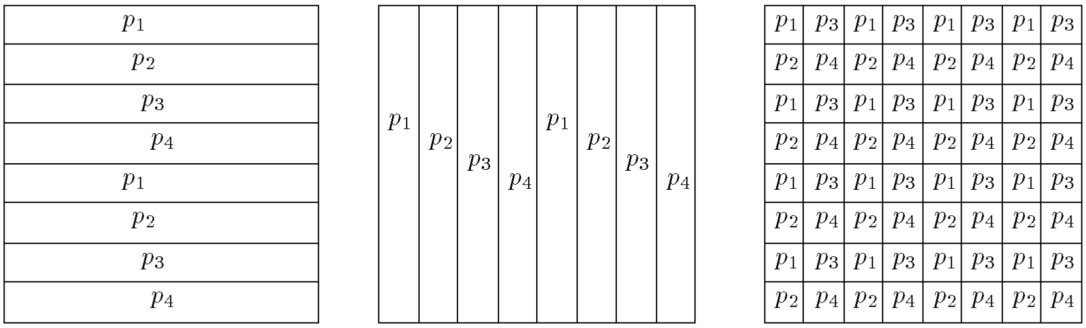
- decomposition dimensions are distributed independently
  (no diagonal distributions are possible)
- decomposition segments have uniform size and shape (except for boundary conditions)
- processor assignments are regular

Figure 2.17 shows some distributions we do not plan to support in Fortran D. We do not believe there will be a significant loss of performance caused by using the regular distributions provided in Fortran D.



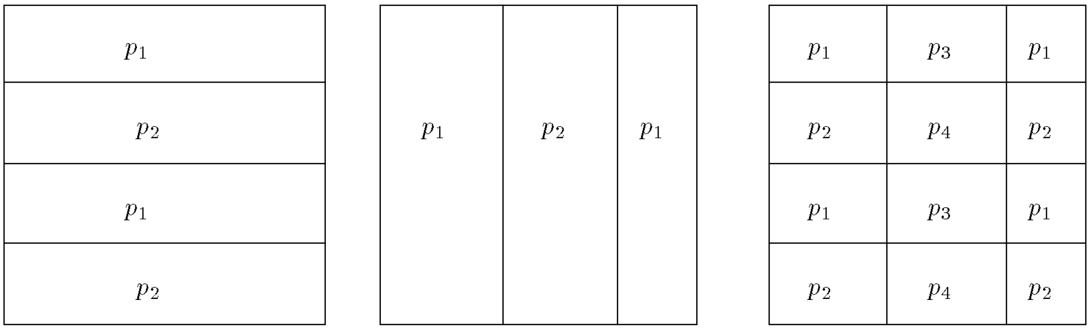
`(BLOCK,:)` `(:,BLOCK)` `(BLOCK,BLOCK)`

Figure 2.11 2-D Block Distributions



`(CYCLIC,:)` `(:,CYCLIC)` `(CYCLIC,CYCLIC)`

Figure 2.12 2-D Cyclic Distributions

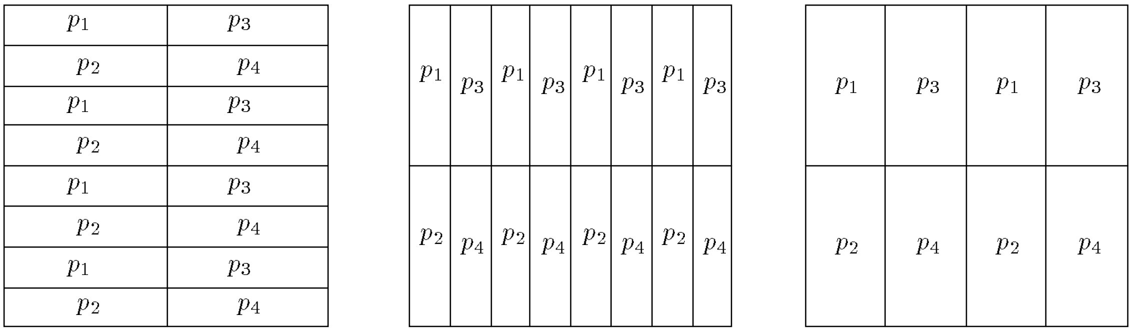


`(BLOCK_CYCLIC(2),:)`
`n$proc = 2`

`(:,BLOCK_CYCLIC(3))`
`n$proc = 2`

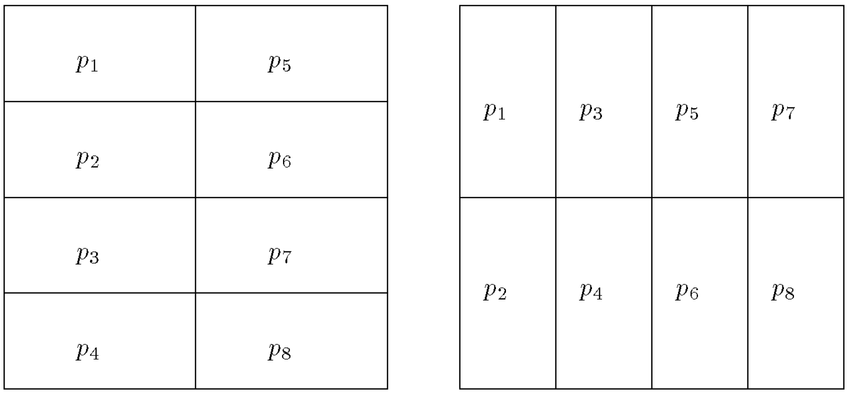
`(BLOCK_CYCLIC(2),`
`BLOCK_CYCLIC(3))`
`n$proc = 4`

Figure 2.13 2-D Block_cyclic Distributions



(CYCLIC,BLOCK) (BLOCK,CYCLIC) (BLOCK,BLOCK_CYCLIC(2))

Figure 2.14 2-D Combination Distributions



(BLOCK(4),BLOCK(2)) (BLOCK(2),BLOCK(4))

Figure 2.15 2-D Uneven Block Distributions

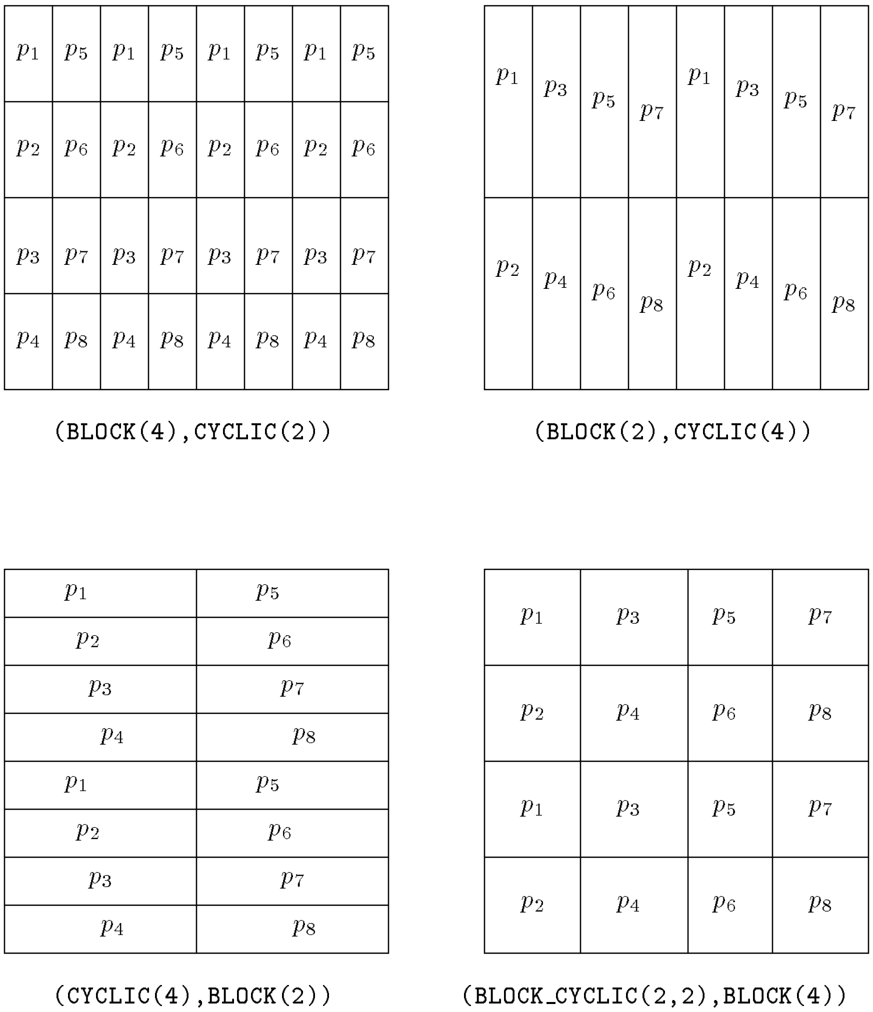


Figure 2.16 2-D Uneven Combination Distributions

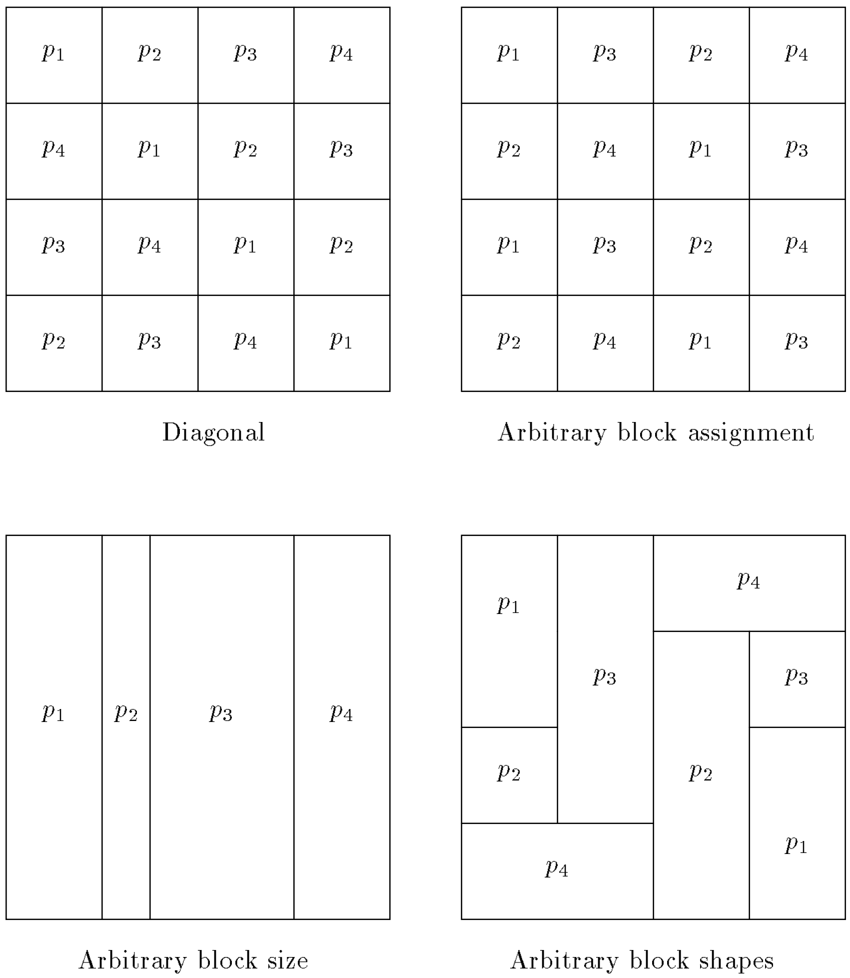


Figure 2.17 Unsupported Distributions

2.6.4 Irregular Distributions

For problems with irregular data parallelism, regular distributions may not be efficient. For these cases, Fortran D allows user specified irregular distributions through the use of a mapping array, which will itself usually be distributed. An example for implementing an irregular distribution in this manner is as follows:

```
n$proc = 4
REAL X(16)
INTEGER MAP(16)
DECOMPOSITION REG(16), IRREG(16)
ALIGN MAP with REG
ALIGN X with IRREG
DISTRIBUTE REG(BLOCK)
...set values of MAP array by some algorithm...
DISTRIBUTE IRREG(MAP)
```

In this example, the elements of MAP must be set to integers between 1 and 4 (the number of processors). IRREG(i) will then be stored on the processor value in MAP(i), as shown in Figure 2.18.

If an element of MAP is not a valid processor number, then that element of decomposition IRREG will not be mapped to any processor; accessing such an element is an error. This is the case with X(15) in the figure. Changes to MAP made after the DISTRIBUTE statement is executed do not affect the distribution. MAP may be either distributed or replicated; distributed MAP arrays will consume less memory, but may require more communication steps to access elements.

2.6.5 Combined Regular and Irregular Distribution

A mixture of regular and irregular distributions may also be used.

```
n$proc = 16
INTEGER MAP(16)
REAL X(16,16)
DECOMPOSITION A(16), B(16,16)
ALIGN MAP with A
ALIGN X with B
DISTRIBUTE A(BLOCK)
...set values of MAP array by some algorithm...
DISTRIBUTE B(MAP,BLOCK(4))
```

In this example, the map array is block distributed among all processors. The array X (aligned with decomposition B) is distributed irregularly in the first dimension according to the map array, and block distributed in the second dimension. Since only four processors are available in the first dimension, the map array must only provide a distribution for processors 1–4.

2.7 Dynamic Alignment and Distribution

Data mappings may change between different computation phases, thereby requiring dynamic realignment and/or redistribution to reduce data movement. We term this *dynamic data decomposition*. To support this, in Fortran D both ALIGN and DISTRIBUTE may be interpreted as executable statments rather than declarations, depending on their location. In the following example, the second set of data specifications cause dynamic realignment of arrays X and Y. This is done to reduce communications for the second loop.

```
REAL X(N), Y(N)
DECOMPOSITION A(N)
ALIGN X, Y with A
DISTRIBUTE A(BLOCK)
DO I = 1,N
   X(I)=Y(I)
ENDDO
```

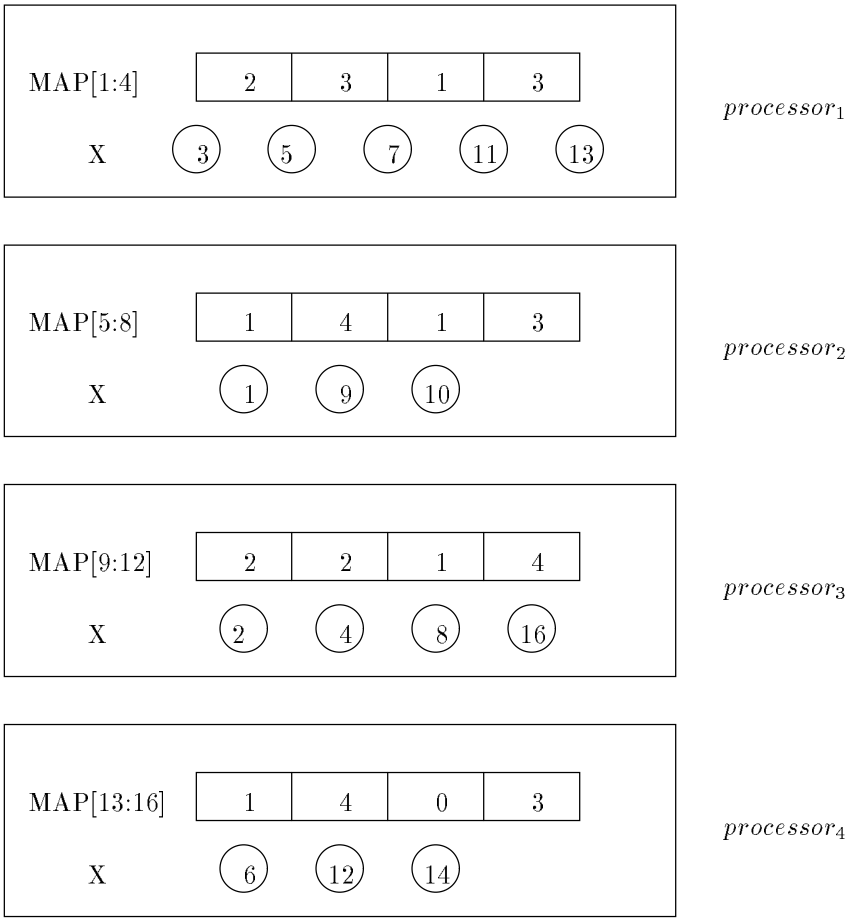


Figure 2.18 Irregular Distribution Example

```
...
ALIGN X(I) with A(I+1)
DO I = 1,N
   X(I)=Y(I+1)
ENDDO
```

Another reason to employ dynamic data distributions is to is to configure a program for greater efficiency, based on the problem size or number of available processors.

```
REAL X(N,N)
DECOMPOSITION B(N,N)
ALIGN X with B
IF (n$proc .GT. 20) THEN
   DISTRIBUTE B(BLOCK,BLOCK)
ELSE
   DISTRIBUTE B(BLOCK,:)
ENDDO
```

In this example, the program is configured so that the data distribution chosen is dependent on the total number of processors available. The Fortran D compiler will require additional sophistication in order to handle dynamic data decompositions.

Restrictions To reduce implementation complexity, the prototype may require that only one local data mapping can "reach" any reference to a distributed variable. This will simplify the job of communication generation. The compiler can ensure this by requiring all ALIGN and DISTRIBUTE statements to be unguarded.

```
REAL X(N)
DECOMPOSITION A(N)
ALIGN X with A
DISTRIBUTE A(BLOCK)
IF ( I .EQ. 1 ) THEN
   DISTRIBUTE A(CYCLIC)
ENDIF
X(I) = 1.0
```

For instance, this example will not be supported, because the program determines at run-time whether X has a BLOCK or CYCLIC distribution.

2.8 Procedures

There are a number of issues concerning procedures in Fortran D. First, it is permitted to call procedures with distributed arrays as arguments. The array formal parameters in the procedure will inherit the data decomposition of the actual parameters in the caller. The Fortran D compiler is responsible for performing all the analysis required to generate the correct code.

Second, in Fortran D the effect of all DECOMPOSITION, ALIGN, and DISTRIBUTE statements are limited to the scope of the enclosing procedure. This provides users with a structured method to limit the scope of their decompositions, and simplifies the problem of dealing with dynamic decompositions.

```
REAL X(N), Y(N)
DECOMPOSITION A(N)
ALIGN X, Y with A
DISTRIBUTE A(BLOCK)
CALL FOO(X)
X(1) = ...
```

For instance, in this example the scoping rule in Fortran D ensures that the array X will be BLOCK distributed at the assignment to X(1), even if procedure FOO redistributes X locally. However, procedures do inherit data decompositions from their callers. Upon entry of procedure FOO, the array X will be BLOCK distributed. Array X may be dynamically redistributed in procedure FOO, but Fortran D ensures that it will not affect the decomposition of X of the parent procedure.

2.8.1 Restrictions

To reduce interprocedural analysis, the prototype Fortran D compiler may place restrictions on procedures. First, to avoid calculating interprocedural reaching data decompositions, the prototype may require procedures to locally declare data decompositions for all distributed arrays accessed. Second, procedure calls may also be barred from FORALL loops.

2.9 FORALL Loops

Certain programming constructs, such as the use of index arrays, make compile-time detection of data dependences impossible. This is especially true for irregular computations, since many parallel loops cannot be detected by the compiler. The compiler is forced to assume *loop-carried* or inter-iteration dependences that force synchronization to be inserted [10].

In a separate situation, the user may wish to design computations such as Jacobi iteration where existing values in an array are used to calculate new values. However, the user is forced to explicitly copy the old values of the array to a separate location in order to avoid inadvertently overwriting the old values before they have been used. This requires both additional effort and storage.

As a remedy, Fortran D defines FORALL to be a loop such that each iteration can *only use values defined before the loop or within the current iteration.* When a statement in an iteration of the FORALL loop accesses a memory location, it will not get any value written by a different iteration of the loop. Instead, it will get the *old* value at that memory location (*i.e.,* the value at that location before the execution of the FORALL loop) or it will get some new value written on the current iteration.

Another way of viewing the FORALL loop is that it has copy-in/copy-out semantics. In other words, each iteration gets its own copy of the entire data space that exists before the execution of the loop, and writes its results to a new data space at the end of the loop [6, 148, 22].

At the end of a FORALL loop, any variables that are assigned *new* values by different iterations have these values *merged* at the end of the loop. Merges are performed deterministically, by using the value assigned from the latest sequential iteration.

The major benefit of a FORALL loop is that since no values depend on other iterations, the loop may be executed in parallel without communication. However, communication may still be required before the loop to acquire non-local values, and after the loop to update or merge non-local values. Another advantage of the FORALL loop is that it has deterministic semantics, provided that the underlying system merges values in a deterministic manner. Finally, the user can use the FORALL loop to preserve values in a *rhs* array without the need for explicit copies or storage.

2.9.1 Example FORALL Loop

The syntax of the FORALL loop is shown in the following example:

```
FORALL I = 1,N
   X(IDX(I)) = ...
   ... = X(IDX(I+1))
ENDDO
```

In this example, the FORALL loop may be executed in parallel without communication or synchronization, even though loop-carried dependences cannot be eliminated by compile-time analysis. Instead, the compiler and run-time system will ensure that statements in the loop body access old values of X instead of new values written on other iterations.

Here we provide a more detailed example. In the following FORALL loop, there are three dependences caused by the assignment to X(I) at statement S_1—a loop-carried antidependence to X(I+1) at S_2, a loop-independent true dependence to X(I) at S_3, and a loop-carried true dependence to X(I-1) at S_4.

```
      FORALL I = 1,N
S1       X(I) = ...
S2       ... = X(I+1)
S3       ... = X(I)
S4       ... = X(I-1)
      ENDDO
```

Both sequential Fortran and FORALL semantics specify that the reference to X(I+1) at S_2 uses its old value; *i.e.,* the value of X(I+1) before it is assigned at statement S_1. Similarly, both sequential and FORALL semantics require the reference to X(I) at S_3 to use the new value; *i.e.,* the value assigned to X(I) at S_1. This is because the new value is assigned in the current iteration.

On the other hand, sequential and FORALL semantics differ for the loop-carried true dependence between X(I) and X(I-1). Sequential Fortran semantics require that the reference to X(I-1) at statement S_4 use the new value computed at S_1. However, FORALL semantics cause statements in the loop body to use new values *only if they are calculated in the current loop iteration.* All other values are old values from before the FORALL loop. The reference to X(I-1) thus uses the old value of X(I-1), before it is assigned to at statement S_1. In effect, all loop-carried true dependences are converted to antidependences in a FORALL loop.

2.9.2 Nested FORALL Loops

Multiple nested FORALL loops may be used to specify more than one level of data parallelism. A nested FORALL loop has exactly the same semantics as the standard FORALL loop—no value may be computed and used on different iterations of the FORALL loop. In most cases, all communications can be moved entirely out of several nested FORALL loops.

```
FORALL I = 1,N
  FORALL J = 1,N
    X(...) = ...
  ENDDO
ENDDO

FORALL I = 1,N
  FORALL J = 1,N
    X(...) = ...
    ... = X(...)
  ENDDO
ENDDO
```

For instance, consider the two example loop nests. Standard FORALL semantics allow all communications resulting from values assigned to X in the inner J loop to be moved outside the J loop. What is less clear is that the communications can actually be moved outside the outer I loop as well. This is because the semantics of the FORALL loop guarantee that the values of X produced (by the J loop) in one iteration of I cannot be used until the entire I loop is complete. Since there is no possible use of these values in the same iteration of loop I, the communications may be delayed to the end of the entire loop nest. However, a different situation exists in this example:

```
FORALL I = 1,N
  FORALL J = 1,N
    X(...) = ...
  ENDDO
  ... = X(...)
ENDDO
```

In this loop nest, there is actually a possible of use of X following the inner J loop. The difference here is that the values generated in the inner J loop may be used in the same iteration of the outer I loop. If the compiler cannot eliminate possible dependence between the definition and use of X, communications may be necessary at the end of the J loop to update values of X.

Our intent in providing the FORALL loop in Fortran D is to provide an *optional* method for users to aid the compiler in generating efficient codes for irregular or sparse computations. FORALL loops are unnecessary for regular computations—we believe that a sophisticated compiler can readily extract the parallelism from normal `do` loops for regular computations.

We have defined semantics of the FORALL loop to be quite close to sequential Fortran. In particular, FORALL loops are deterministic. As a result we believe that it will be easy to understand and use for scientific programmers. The FORALL loop possesses similar semantics to the CM Fortran FORALL statement [196, 6] and the Myrias `PARDO` loop [22]. In fact, the FORALL statement in CM Fortran is simply a special form of the FORALL loop—one that has only one statement in the loop body.

2.9.3 Restrictions

With the assistance of dependence analysis, FORALL loops with regular computation patterns may be compiled into efficient code. When compile-time analysis is insufficient to guarantee deterministic results for a FORALL loop, run-time support is required. This is particularly true for irregular computations over sparse arrays involving index arrays, as in the following example.

```
FORALL I = 1, N
   X(IDX(I)) = X(IDX(I+1))
ENDFOR
```

Because the index array *IDX* is used, it is not possible to determine at compile-time whether there are any loop-carried true or output dependences that must be handled. There are two possible run-time approaches.

First, the compiler may conservatively save all *rhs* values potentially accessed by the FORALL loop, then modify the loop to use those values. This results in the code below.

```
DO I = 1, N
   XOLD(I) = X(I)
ENDDO
FORALL I = 1, N
   X(IDX(I)) = XOLD(IDX(I+1))
ENDFOR
```

Buffering data eliminates the possibility of nondeterminism, but may require significant amounts of storage. Additional code must be inserted to ensure deterministic merge of output dependences. Alternatively, the compiler may generate code to preprocess values of IDX(I) at run-time, in order to determine whether old values of X need to be saved.

For ease of implementation the prototype compiler may wish to simply assume that no loop-carried true or output dependences exist for FORALL loops containing irregular computations. A compile-time warning should then list all references that may cause violations of FORALL semantics. In particular, nondeterministic merges of values produced by output dependences carried by the FORALL loop are rare and easily detected. Alternatively, special compile-time options may be provided to instruct the compiler to either generate code to ensure the proper results through some form of run-time resolution, or to generate debugging code that will detect when such violations occur.

2.10 Reductions

A reduction is an operation on a collection of data that results in new data of lesser dimensionality, usually a single scalar value. Simple but common examples of reductions include calculating the sum or maximum of a vector or array of numbers. Fortran D provides the REDUCE statement as an *optional* method of specifying reductions that the compiler may find difficult to detect. It can also be used to specify reductions in FORALL loops that bypass its copy-in/copy-out semantics. The syntax of the REDUCE statement is as follows:

REDUCE(*function*, LHS, RHS)

Where the *function* is the reduction function to be performed, the *lhs* is the target data, and the RHS is the source data. The following standard reduction functions are provided in Fortran D:

SUM	sum of a list of numbers
PROD	product of a list of numbers
MIN	minimum of a list of numbers
MAX	maximum of a list of numbers
AND	logical AND of a list of booleans
OR	logical OR of a list of booleans

Programmers may also define their own reduction functions, providing much greater flexibility in performing reductions. Fortran D will assume any function passed to a REDUCE statement to be user-defined if it does not match the name of a standard reduction function. In such cases, all other arguments to the REDUCE statement are passed as arguments to the user-defined reduction function. The following example shows reductions performed with both standard and user-defined reduction functions:

```
REAL X(N), S, P, M1, M2, Z
BOOLEAN B(N), T1, T2
DO I = 1,N
   REDUCE(SUM, S, X(I))
   REDUCE(PROD, P, X(I))
   REDUCE(MIN, M1, X(I))
   REDUCE(MAX, M2, X(I))
   REDUCE(AND, T1, B(I))
   REDUCE(OR, T2, B(I))
   REDUCE(USER_FUNCTION, Z, X(I))
ENDDO
```

Reductions in DO loops may be automatically recognized by the Fortran D compiler, even if the REDUCE statement is not employed. However, use of the REDUCE statement is required for reductions in FORALL loops, since FORALL semantics change the meaning of standard user-programmed reductions. Reductions in essence provide appropriate merge functions for FORALL loops.

2.10.1 Restrictions

Reductions provide a means for executing otherwise sequential computations in parallel. However, several restrictions must be observed in order to avoid nondeterministic results when using reductions in FORALL loops. First, because reductions change the order of operations in a reduction, all user-defined reduction functions must be both associative and commutative. Otherwise, any change in the actual evaluation order of the reduction may affect the final value returned by the reduction operation. Note that this requirement has to be relaxed for floating point operations, which may prove unstable because of rounding errors.

Second, since intermediate values of the LHS variables in reductions are undefined, they must not be used within the loop. However, they may be employed in other REDUCE statements in the same loop, provided that the reduction functions are identical. The following shows some examples of variables involved in multiple reductions.

```
REAL X(N), Y(N), S, M
FORALL I = 1,N
   REDUCE(SUM, S, X(I))
   REDUCE(SUM, S, Y(I))
   IF (...) THEN
     REDUCE(MAX, M, X(I))
   ELSE
     REDUCE(MAX, M, Y(I))
   ENDIF
   ... = S
ENDFOR
```

In the previous example, the variables S and M serve as the LHS of several reductions. Variable S is used to sum up the values in both arrays X and Y, and variable M is set to the maximum value in some subset of arrays X and Y. In both cases the reductions are legal since the same reduction function is used. On the other hand, the Fortran D compiler will mark the last statement in the loop as illegal, because it attempts to use an intermediate value of variable S during execution of the FORALL loop.

2.10.2 Location Reductions

Fortran D also provides additional support for determining the location of minimum or maximum values. For the MIN and MAX reductions, the REDUCE statement will accept additional pairs of arguments of the form <LHS,RHS>. In the course of the reduction, the values of the RHS will be assigned to that of the LHS when the minimum or maximum element is found. This provides a mechanism for determining the location of the minimum or maximum value.

```
INTEGER I, J, IDX1, IDX2, IDX3, IDX4, IDX5, IDX6
REAL X1(N), X2(N,N), M1, M2, M3, M4
DO I = 1,N
   REDUCE(MIN, M1, X1(I), IDX1, I)
   REDUCE(MAX, M2, X1(I), IDX2, I)
   DO J = 1,N
     REDUCE(MIN, M3, X2(I,J), IDX3, I, IDX4, J)
     REDUCE(MAX, M4, X2(I,J), IDX5, I, IDX6, J)
   ENDDO
ENDDO
```

In the previous example, additional arguments of the form <var, current index> are passed to the REDUCE statements for to find the index of the minimum or maximum element of the array. If there are multiple elements with the minimum or maximum value, the assignment is performed only for the first such value found.

2.11 On Clause

Fortran D provides a feature from KALI [127], an *optional* ON clause. The ON clause is used to specify the processor which will execute each iteration of a loop. This allows user greater control of where the computation is performed for load-balancing and reducing communications.

```
n$proc = 4
REAL X(1024), Y(1024), Z(1024)
DECOMPOSITION A(1024)
ALIGN X, Y, Z with A
DISTRIBUTE A(BLOCK)
FORALL I = 1,512 on HOME(A(I))
   X(I+512) = F(X(I),Y(I),Z(I))
ENDFOR
```

In this example, it may be advantageous to perform the computation on the processor where the data is stored (where X(I) is) rather than where the results are to be sent (where X(I+512) is). This is precisely what the ON clause specifies. There are three forms of the ON clause.

```
n$proc = 4
REAL X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I+1)
DISTRIBUTE A(CYCLIC)
FORALL I = 1,N on HOME(A(I))
FORALL I = 1,N on HOME(X(I))
FORALL I = 1,N on MOD(I, n$proc) + 1
```

In all cases, the expression in the ON clause names the processor to execute a given iteration of the FORALL loop. `HOME` is used to derive the identifier of the actual processor assigned ownership. Referencing the `HOME` of a decomposition or array element in the ON clause will cause the iteration to be assigned to the processor where that element is mapped. This is the case for the first two FORALL loops. Otherwise the ON clause takes an expression to calculate a processor identifier between 0 and N$PROC-1, and directly assigns each iteration of the loop to a specific processor. Arbitrary expressions are allowed in the processor, decomposition, or array subscripts. However, the user should be aware that complex expressions will be difficult for the compiler to implement efficiently.

2.12 Discussion

Programming languages lack support to efficiently exploit fine-grain data parallelism on distributed-memory machines. We believe that explicit data alignment and distribution specifications provide programmers and compiler writers with the correct paradigm for specifying data decompositions. We have designed Fortran D to be powerful enough to express most fine-grain parallel computations, but also simple enough that a sophisticated compiler can produce efficient programs for different parallel architectures. To make it usable for computational scientists, we have also made the meaning of Fortran D deterministic and quite close to sequential Fortran. In fact, any Fortran program is also a valid Fortran D program.

Chapter 3

Compilation Model

The Fortran D compiler utilizes a code generation strategy based on the "owner computes" rule—where each processor only computes values of data it owns. Fortran D data decomposition specifications are translated into mathematical distribution functions that determine the ownership of local data. By composing these with subscript functions or their inverse, the Fortran D compiler can partition the computation and determine nonlocal accesses at compile-time. This information is used to generate efficient SPMD program for execution on the nodes of the distributed-memory machine.

3.1 Introduction

Writing efficient parallel message-passing programs for MIMD distributed-memory machines is a difficult and machine-dependent task. The Fortran D compiler relieves users of this responsibility by automatically translating sequential Fortran 77 or 90 programs into SPMD Fortran 77 programs containing explicit calls to message-passing routines. In this thesis we refer to this process as "compiling" Fortran D, even though it is actually a source-to-source translation at the Fortran level.

There are two major concerns in compiling Fortran D for MIMD distributed-memory machines.

- Partition data and computation across processors.
- Generate communications where needed to access nonlocal data.

Our philosophy is to use the "owner computes" rule, where every processor only performs computation for data it owns [212, 38, 178]. The owner compute rule may be relaxed depending on the structure of the computation; however, in this paper we concentrate on deriving a functional decomposition and communication generation by applying the owner computes rule.

3.2 Compilation Example

We begin by using the simple example program in Figure 3.1 to compare two different approaches to compiling Fortran D programs. For clarity, we assume that the compiler targets a machine with four processors.

3.2.1 Run-time Resolution

A simple compilation technique known as *run-time resolution* yields code that explicitly calculates the ownership and communication for each reference at run-time [38, 178, 212]. For instance, for the previous example it generates the code shown in Figure 3.2. Run-time resolution does not require much compiler analysis, but the resulting programs are likely to extremely inefficient. In fact, they may execute much slower than the original sequential code.

There are several reasons for the poor performance of run-time resolution. First, parallelism is mostly lost because each processor must execute the entire program. Worse still, not only does the program have to explicitly check every variable reference, it generates a message for each nonlocal access. , Without adequate compile-time information, the compiler is forced to rely on run-time techniques. Fortunately, few programs inherently require run-time resolution.

```
PROGRAM P1
  REAL X(100)
  PARAMETER (n$proc = 4)
  DECOMPOSITION D(100)
  ALIGN X with D
  DISTRIBUTE D(BLOCK,:)
  do i = 1,95
    X(i) = F(X(i+5))
  enddo
end
```

Figure 3.1 Simple Fortran D Program

```
PROGRAM P1
  REAL X(100)
  my$p = myproc() {* 0...3 *}
  do i = 1,95
    if (my$p .EQ. owner(X(i+5))) then
      send X(i+5) to owner(X(i))
    endif
    if (my$p .EQ. owner(X(i))) then
      recv X(i+5) from owner(X(i+5))
      X(i) = F(X(i+5))
    endif
  enddo
end
```

Figure 3.2 Run-time Resolution

```
PROGRAM P1
  REAL X(30)
  my$p = myproc() {* 0...3 *}
  if (my$p .GT. 0) send X(1:5) to my$p-1
  if (my$p .LT. 3) recv X(16:30) from my$p+1
  ub$1 = min((my$p+1)*25,95)-(my$p*25)
  do i = 1,ub$1
    X(i) = F(X(i+5))
  enddo
end
```

Figure 3.3 Compile-time Analysis and Optimization

3.2.2 Compile-time Analysis and Optimization

In comparison, when extensive compile-time analysis is performed, the Fortran D compiler can produce highly efficient code. We demonstrate by examining the compilation process for the example program. During dependence analysis, we determine that i is the index variable of a loop iterating from 1 to 95, and that there are no true dependences for statement S_1 (though a loop-carried anti-dependence exists). During data partitioning, we find that decomposition D has size 100 and is distributed blockwise across four processors, assigning to each a contiguous block of 25 elements. Throughout the program array X is perfectly aligned with D, so each processor also has 25 elements of X.

During computation partitioning, we apply the inverse of the subscript function for $X(i)$ to determine that each processor executes 25 iterations of loop i. Intersection with the actual loop bounds shows that boundary conditions exist for the last processor. During communication analysis, we determine that the reference $X(i+5)$ causes nonlocal accesses on the last five loop iterations. Checking boundary conditions, we find all accesses on the last processor are local. During communication optimization, we compare the subscript expressions and choose to use calls to *send* and *recv* primitives for the simple shift communication pattern. The lack of true dependences allows messages to be vectorized outside of the i loop. During storage management, we choose to provide storage for nonlocal data using overlaps, expanding the local bounds of X.

During code generation, we first eliminate Fortran D statements, then introduce an assignment to set my$p to the local processor number, an integer between 0 and 3. We instantiate the data partition by reducing the array bounds for X. We instantiate the computation partition by reducing the bounds of the i loop to 1 through 25. An expression is generated for the upper bound to handle boundary conditions for the last processor. We then introduce communication by inserting calls to *send* and *recv* routines preceding the i loop. The location and owners of the data are calculated at compile time and are used as arguments to the communication routines. Because boundary conditions are present, guards are inserted so that only the correct processors communicate.

The final compiler generated code is shown in Figure 3.3. As can be seen, it is much more efficient than the program generated using run-time resolution. Computation has been statically partitioned efficiently to exploit parallelism, and communication overhead has been greatly reduced by combining all nonlocal accesses in a single message. This and the next few chapters describe the compilation process in greater detail.

3.3 Formal Model

In this section we provide a formal description of the Fortran D compilation model. Terminology and notation are introduced. They are used to show conceptually how the Fortran D compilation process progresses from a program with data decompositions to a SPMD node program with explicit message-passing. Actual design and structure of the Fortran D compiler are presented later in Chapter 4.

We begin by examining the algorithm used to compile a simple loop nest using the owner computes rule. Correct application of the rule requires knowledge of the data decomposition for a program. In Fortran D information concerning the ownership of a particular decomposition or array element is provided by the ALIGN and DISTRIBUTE statements.

3.3.1 Distribution Functions

Data distribution functions specify the mapping of data arrays. The ALIGN and DISTRIBUTE statements in Fortran D specify how distributed arrays are mapped to the physical machine. The Fortran D compiler uses the information contained in these statements to construct *distribution functions* that can be used to calculate the mapping of array elements to processors. Distribution functions are also created for decompositions and are used during the actual distribution of arrays onto processors.

The distribution function μ, defined below,

$$\mu_A(\vec{i}) \;=\; (\delta_A(\vec{i}), \alpha_A(\vec{i})) \;=\; (p, \vec{j})$$

is a mapping of the global index $\vec{i}$ of a decomposition or array A to a local index $\vec{j}$ for a unique processor p. Each distribution function has two component functions, δ and α. These functions are used to compute ownership and location information. For a given decomposition or array A, the owner function δ_A maps the global index $\vec{i}$ to its unique processor owner p, and the local index function α_A maps the global index $\vec{i}$ to a local index $\vec{j}$.

Regular Distributions

The formalism described for distribution functions are applicable for both regular and irregular distributions. An advantage of the simple regular distributions supported in Fortran D is that their corresponding distribution functions can be easily derived at compile-time. For instance, given the following regular distributions,

$$\begin{aligned}
\mu_A^{(block,:)}(i,j) &= (\lceil i/BlockSize \rceil, ((i-1) \bmod BlockSize + 1, j)) \\
\mu_B^{(cyclic,:)}(i,j) &= ((i-1) \bmod P + 1, (\lceil i/P \rceil, j)) \\
\mu_X(i,j) &= (\lceil i/BlockSize \rceil, ((i-1) \bmod BlockSize + 1, j+1)) \\
\mu_Y(i,j) &= ((j-1) \bmod P + 1, (\lceil j/P \rceil, i))
\end{aligned}$$

Figure 3.4 Distribution Functions

```
REAL X(N, 0:N-1), Y(N,N)
DECOMPOSITION A(N,N), B(N,N)
ALIGN X(I,J) with A(I, J+1)
ALIGN Y(I,J) with B(J, I)
DISTRIBUTE A(BLOCK, :)
DISTRIBUTE B(CYCLIC, :)
```

the compiler automatically derives the distribution functions in Figure 3.4. In the figure, the 2-D decompositions A and B are declared to have size (N, N). The number of processors is P. For a block distribution, $BlockSize = \lceil N/P \rceil$.

Irregular Distributions

For an irregular distribution, we use an integer array to explicitly represent the component functions $\delta_A(\vec{i})$ and $\alpha_A(\vec{i})$. This is the most general approach possible since it can support any arbitrary irregular distribution. Unfortunately, the distribution must now be evaluated at run-time. In the following 1-D example,

```
INTEGER MAP(N), X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I)
DISTRIBUTE A(MAP)
```

the irregular distribution for decomposition A is stored in the integer array **MAP**. The distribution functions for decomposition A and array X are then computed through run-time preprocessing techniques [188, 153]. Researchers are examining more sophisticated methods of specify irregular distributions for Fortran D programs [47, 172, 209].

3.3.2 Computation

We continue to describe some additional notation we will employ later in this paper. We assume the computation is represented by the following simple loop nest:

DO $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$
  $X(g(\vec{k})) = Y(h(\vec{k}))$
enddo

In the example loop nest, $\vec{k}$ is the set of loop iterations. It is also displayed as as the triplet $[\vec{l} : \vec{m} : \vec{s}]$. In addition, X and Y are distributed arrays, and g and h are the array subscript functions for the left-hand side (lhs) and right-hand side (rhs) array references, respectively.

3.3.3 Image, Local Index Sets

We define the *image* of an array X on a processor p as follows:

$$image_X(p) \;=\; \{\vec{i} \mid \delta_X(\vec{i}) = p\}$$

The *image* for a processor p is constructed by finding all array indices that cause a reference to a local element of array X, as determined by the distribution functions for the array. As a result, *image* describes all the elements of array X assigned to a particular processor p. This may also referred to as the *local index set* of the array. In addition, we define t_p as *this processor*, a unique processor identification representing the local processor. Thus the expression $image_X(t_p)$ corresponds to the set of all elements of X owned locally.

3.3.4 Iteration Sets

We define the *iteration set* of a reference R for a processor p to be the set of loop iterations $\vec{j}$ that cause R to access data owned by p. Each element of the iteration set corresponds to a point in the iteration space, and is represented by a vector containing the iteration number for each loop in the loop nest.

The iteration set of a statement can be constructed in a very simple manner. Our example loop contains two references, $X(g(\vec{k}))$ and $Y(h(\vec{k}))$. The iteration set for processor p with respect to reference $X(g(\vec{k}))$ is simply $g^{-1}(image_X(p))$, the inverse subscript function g^{-1} applied to the image of the array X on processor p. Similarly, the iteration set with respect to reference $Y(h(\vec{k}))$ can be calculated as $h^{-1}(image_Y(p))$.

This property will be used in several algorithms later in the paper. In particular, notice that when using the owner computes rule, the iteration set of the *lhs* of an assignment statement for processor p is exactly the iterations in which that statement must be executed on p. For example, in the simple loop above, the function $g^{-1}(image_X(t_p))$ may be used to determine when t_p, the local processor, should execute the statement.

3.3.5 Computation Partitioning

The computation partitioning phase of the compiler ensures that computations in a program are divided correctly among the processors according to the owner computes rule. This may be accomplished by a combination of reducing loop bounds and guarding individual statements. Both approaches are based on calculating *iteration sets* for statements in a loop.

Loop Bounds Reduction

Since evaluating guards at run-time increases execution cost, the Fortran D compiler strategy is to reduce loop bounds where possible for each processor to avoid evaluating guard expressions.

Figure 3.5 presents a straightforward algorithm for performing simple loop bounds reduction. The algorithm works as follows. First, the iteration sets of all the *lhs* are calculated for the local processor t_p. These

for each loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ **do**
    $reduced\_iter\_set = \emptyset$
    for each statement$_i$ in loop with $lhs = X_i(g_i(\vec{k}))$
        $iter\_set = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
        $reduced\_iter\_set = reduced\_iter\_set \cup iter\_set$
    endfor
    reduce bounds of loop nest to those in *reduced_iter_set*
endfor

Figure 3.5 Reducing Loop Bounds Using Iteration Sets

for each loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ **do**
    $previous\_iter\_set = [\vec{l} : \vec{m} : \vec{s}]$
    for each statement$_i$ in order **do**
        if statement$_i$ = assignment AND lhs = global array $X_i(g_i(\vec{k}))$ **then**
            $iter\_set = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
        else
            $iter\_set = [\vec{l} : \vec{m} : \vec{s}]$
        endif
        if $iter\_set = previous\_iter\_set$ **then**
            insert statement$_i$ after statement$_{i-1}$
        else
            terminate previous mask if it exists
            create new mask for *iter_set* and insert statement$_i$ inside mask
            $previous\_iter\_set = iter\_set$
        endif
    endfor
endfor

Figure 3.6 Generating Statement Masks Using Iteration Sets

sets are then unioned together. The result represents all the iterations on which a assignment will need to be executed by the processor. The loop bounds are then reduced to the resulting iteration set.

Guard Introduction

In the case where all assignment statements have the same iteration set, loop bounds reduction will eliminate any need for masks since all statements within the reduced loop bounds always execute. However, loop bound reduction will not work in all cases. For instance, loop nests may contain multiple assignment statements to distributed data structures. The iteration set of each statement for a processor may differ, limiting the number of guards eliminated through bounds reduction. The compiler will need to introduce masks for the statements that are conditionally executed.

Figure 3.6 presents a simple algorithm to generate masks for statements in a loop nest. Each statement is examined in turn and its iteration set calculated. If it is equivalent to the iteration set of the previous statement, then the two statements may be guarded by the same mask. Otherwise, any previous masks must be terminated and a new mask created. We assume the existence of functions to generate the appropriate guard/mask for each statement based on its iteration set.

3.3.6 Communication Generation

Once guards have been introduced, the Fortran D compiler must generate communications for nonlocal accesses to preserve the meaning of the original program. This can be accomplished by calculating SEND and RECEIVE iteration sets. For simple loop nests which do not contain *loop-carried* (inter-iteration) true dependences [10], These iteration sets may also be used to generate IN and OUT array index sets that combine messages to a single processor into one message. We describe the formation and use of these sets in more detail in the following sections.

LOCAL, SEND, and RECEIVE Iteration Sets

We describe as *regular computations* those computations which can be accurately characterized at compile-time. In these cases the compiler can exactly calculate all communications and synchronization required without any run-time information. The first step is to calculate the following iteration sets for each reference R in the loop with respect to the local processor t_p:

- LOCAL – Set of iterations in which R results in an access to data local to t_p.
- SEND – Set of iterations in which R results in an access to data local to t_p, but the statement containing R is executed on a different processor.
- RECEIVE – Set of iterations in which the statement containing R is executed on t_p, but R results in an access to data not local to t_p.

The LOCAL, SEND, and RECEIVE iteration sets can be generated using the owner computes rule. Figure 3.7 shows the algorithm for regular computations. It starts by first calculating the iteration set for the *lhs* of each assignment statement with respect to the local processor t_p; this determines the LOCAL iteration set.

---

for each statement$_i$ with $lhs = X_i(g_i(\vec{k}))$ in loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ **do**
  $local\_iter\_set^{t_p}_{X_i} = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
  for each *rhs* reference to a distributed array $Y_i(h_i(\vec{k}))$ **do**
    $local\_iter\_set^{t_p}_{Y_i} = h_i^{-1}(image_{Y_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    $receive\_iter\_set^{t_p}_{Y_i} = local\_iter\_set^{t_p}_{X_i} - local\_iter\_set^{t_p}_{Y_i}$
    $send\_iter\_set^{t_p}_{Y_i} = local\_iter\_set^{t_p}_{Y_i} - local\_iter\_set^{t_p}_{X_i}$
  endfor
endfor

Figure 3.7 Generating SEND/RECEIVE Iteration Sets (for Regular Computations)

The iteration sets for each *rhs* of the statement are then constructed with respect to the t_p. Any element of the LOCAL iteration set that does not also belong to the iteration set for the *rhs* will need to access nonlocal data; it is put in the RECEIVE iteration set. Conversely, any elements in the iteration set for the *rhs* not also in the LOCAL iteration are needed by some other processor; it is put into the SEND iteration set. These iteration sets complete specify all communications that must be performed.

IN and OUT Index Sets

For loop nests which do not contain loop-carried true dependences, communications may be moved entirely outside of the loop nest and blocked together. In addition, messages to the same processor may also be combined to form a single message. These steps are desirable when communication costs are high, as is the case for most MIMD distributed-memory machines. The following array index sets are utilized for these optimizations:

- IN – Set of array indices that correspond to nonlocal data accesses. These data elements must be received from other processors in order to perform local computations.
- OUT – Set of array indices that correspond to local data accessed by other processors. These data elements must be sent to other processors in order to permit them to perform their computations.

The calculation of IN and OUT index set for regular computations is depicted in Figure 3.8. The algorithm works as follows. Each element in the SEND and RECEIVE iteration sets is examined. Some combination of the subscript, mapping, alignment, and distribution functions and their inverses are applied to the element to determine the source or recipient of each message. The message to that processor is then stored in the appropriate IN or OUT index set, effectively combining it with all other messages to the same processor.

More complicated algorithms are needed for loops with loop-carried dependences, since not all communication can be moved outside of the entire loop nest. To handle loop-carried dependences, IN and OUT index sets need to be constructed at each loop level. Dependence information may be used to calculate the appropriate loop level for each message, using the algorithms described by Balasundaram *et al.* and Gerndt [16, 80]. Messages in SEND and RECEIVE sets can then be inserted in the IN or OUT set at that loop level.

---

for each statement$_i$ with $lhs = X_i(g_i(\vec{k}))$ in loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ **do**
  for each *rhs* reference to a distributed array $Y_i(h_i(\vec{k}))$ **do**
    {∗ initialize IN and OUT index sets ∗}
    for $proc = 1$ to $numprocs$ **do**
      $in\_index\_set_{Y_i}^{(t_p,proc)} = \emptyset$
      $out\_index\_set_{Y_i}^{(t_p,proc)} = \emptyset$
    endfor
    {∗ compute OUT index sets ∗}
    for each $\vec{j} \in send\_iter\_set_{Y_i}^{t_p}$ **do**
      $send_p = \delta_{X_i}(g_i(h_i^{-1}(\mu_{Y_i}^{-1}(t_p, \alpha_{Y_i}(h_i(\vec{j}))))))$
      $out\_index\_set_{Y_i}^{(t_p,send_p)} = out\_index\_set_{Y_i}^{(t_p,send_p)} \cup \{\alpha_{Y_i}(h_i(\vec{j}))\}$
    endfor
    {∗ compute IN index sets ∗}
    for each $\vec{j} \in receive\_iter\_set_{Y_i}^{t_p}$ **do**
      $recv_p = \delta_{Y_i}(h_i(\vec{j}))$
      $in\_index\_set_{Y_i}^{(t_p,recv_p)} = in\_index\_set_{Y_i}^{(t_p,recv_p)} \cup \{\alpha_{Y_i}(h_i(\vec{j}))\}$
    endfor
  endfor
endfor

Figure 3.8 Generating IN/OUT Index Sets (for Regular Computations)

---

for each statement$_i$ with *lhs* $= X_i(g_i(\vec{k}))$ in loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ **do**
  $local\_iter\_set^{t_p}_{X_i} = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
  $receive\_iter\_set^{t_p}_{Y_i} = \emptyset$
  for each *rhs* reference to a distributed array $Y_i(h_i(\vec{k}))$ **do**
    $\{*$ calculate IN index sets for this processor $*\}$
    for each $\vec{j} \in local\_iter\_set^{X_i}_{t_p}$ **do**
      if $(\delta_{Y_i}(h_i(\vec{j}))) \neq t_p)$ **then**
        $receive\_iter\_set^{t_p}_{Y_i} = receive\_iter\_set^{t_p}_{Y_i} \cup \{\vec{j}\}$
        $recv_p = \delta_{B_i}(h_i(\vec{j}))$
        $in\_index\_set^{(t_p, recv_p)}_{Y_i} = in\_index\_set^{(t_p, recv_p)}_{Y_i} \cup \{\alpha_{Y_i}(h_i(\vec{j}))\}$
      endif
    endfor
    $\{*$ send IN index sets to all other processors $*\}$
    for $recv_p = 1$ to $numprocs$ **do**
      if $(recv_p \neq t_p)$ **then**
        $send(in\_index\_set^{(t_p, recv_p)}_{Y_i},\ recv_p)$
      endif
    endfor
    $\{*$ receive IN index sets, convert into OUT index sets $*\}$
    for $send_p = 1$ to $numprocs$ **do**
      if $(send_p \neq t_p)$ **then**
        $receive(out\_index\_set^{(t_p, send_p)}_{Y_i},\ send_p)$
      endif
    endfor
  endfor
endfor

Figure 3.9 Inspector to Generate IN/OUT Index Sets (for Irregular Computations)

Irregular Computations

Irregular computations are computations that cannot be accurately characterized at compile-time. Note that irregular computations are different from irregular distributions, which are irregular mappings of data to processors. It is not possible to determine the SEND, RECEIVE, IN, and OUT sets at compile-time for these computations. However, an *inspector* [155, 125] may be constructed to preprocess the loop body at run-time to determine what nonlocal data will be accessed. This in effect calculates the IN index set for each processor. A global transpose operation between processors can then be used to calculate the OUT index sets as well.

An inspector is the most general way to generate IN and OUT sets for loops without loop-carried dependences. Despite the expense of additional communications, experimental evidence from several systems [127, 209] proves that it can improve performance by combining communications to access nonlocal data outside of the loop nest. In addition it also allows multiple messages to the same processor to be combined. The Fortran D compiler plans to automatically generate inspectors where needed for irregular computations.

The structure of an inspector loop is shown in Figure 3.9. For compatibility with our treatment of regular computations, the Fortran D inspector also generates the LOCAL and RECEIVE iteration sets. In the first part of the inspector, the LOCAL iteration set is calculated for each statement based on the *lhs*. The *rhs* is examined for each element in the LOCAL iteration set. Any nonlocal references cause the iteration to be added to the RECEIVE iteration set. The owner and local index of the nonlocal reference are then calculated and added to the IN index set.

After the local IN sets have been calculated, a global transpose is performed in the remainder of the inspector. Each processor sends its IN index set for a given processor to that processor. Upon receipt, they become OUT index sets for the receiving processor.

{$*$ original loop to be transformed into send, receive, and compute loops $*$}
DO $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$
  $X(g(\vec{k})) = Y(h(\vec{k}))$
enddo
{$*$ send loop $*$}
for $send_p = 1$ to $numprocs$ **do**
  if $(out\_index\_set_Y^{(t_p,send_p)} \neq \emptyset)$ **then**
    $buffer\_values(out\_value\_set_Y^{(t_p,send_p)}, out\_index\_set_Y^{(t_p,send_p)})$
    $send(out\_value\_set_Y^{(t_p,send_p)}, send_p)$
  endif
endfor
{$*$ local compute loop $*$}
for each $\vec{j} \in \{local\_iter\_set_X^{t_p} - receive\_iter\_set_Y^{t_p}\}$ **do**
  $X(\alpha_A(g(\vec{j}))) = Y(\alpha_B(h(\vec{j})))$
endfor
{$*$ receive loop $*$}
for $recv_p = 1$ to $numprocs$ **do**
  if $(in\_index\_set_Y^{(t_p,recv_p)} \neq \emptyset)$ **then**
    $receive(in\_value\_set_Y^{(t_p,recv_p)}, recv_p)$
    $store\_values(in\_value\_set_Y^{(t_p,recv_p)}, in\_index\_set_Y^{(t_p,recv_p)})$
  endif
endfor
{$*$ nonlocal compute loop $*$}
for each $\vec{j} \in receive\_iter\_set_Y^{t_p}$ **do**
  $X(\alpha_X(g(\vec{j})))) = get\_value(Y(h(\vec{j})))$
endfor

Figure 3.10 Send, Receive, and Compute Loops Resulting from In/Out Index Sets

3.3.7 Resulting Program

Once the Send and Receive sets have been calculated, the example loop nest is transformed into the loops pictured in Figure 3.10 [125]. In the *send* loop, every processor sends data they own to processors that need the data. The Out index set for *rhs* of the statement in the example loop has already been calculated. However, the function *buffer_values*() must be used to actually collect the values at each index and the Out set. The resulting values are then sent to the appropriate processor.

Next, in the *local compute* loop, loop iterations that assign and use only local data may be executed. These are elements that are in the Local but not Receive iteration sets. These iterations are executed immediately following the send loop to take advantage of communication latency.

In the *receive* loop, every processor receives nonlocal data sent from their owners in the send loop. The values received are mapped to their designated storage locations using the function *store_values*(). The indices corresponding to these values have already been calculated and stored in the In index sets. Finally, in the *nonlocal compute* loop every processor performs computations for loop iterations that also require nonlocal data. The function *get_value*() is used to fetch nonlocal data from their designated storage locations.

3.4 Discussion

We have shown how the Fortran D compiler utilizes compile-time analysis to avoid the inefficiencies of run-time resolution. Its code generation strategy is based on the owner computes rule. Fortran D data decomposition specifications are translated into mathematical functions that determine the ownership of local data. By composing these with subscript functions or their inverse, the Fortran D compiler can partition the computation and determine nonlocal accesses at compile-time. This information is used to guide a source-to-source translation that generates the SPMD output program with explicit message passing.

Chapter 4

Basic Compilation

The basic structure of the Fortran D compiler is organized around three major functions—program analysis, program optimization, and code generation. New analysis techniques are required to compile shared-memory programs for distributed memory machines. Internal data structures used in the compilation process are described. The Fortran D compiler utilizes a compilation strategy based on the concept of *data dependence* that unifies and extends previous techniques.

4.1 Introduction

The chapter describes specific details of the implementation of the Fortran D compiler, including choice of internal data structures and how they are used during compilation. The overall structure and sequence of compilation phases of the Fortran D compiler is shown in Figure 4.1. It can be roughly divided into three areas: program analysis, optimization, and code generation.

4.2 Program Analysis

4.2.1 Dependence Analysis

Dependence analysis is the compile-time analysis of control flow and memory accesses to determine a statement execution order that preserves the meaning of the original program. A *data dependence* between two references R_1 and R_2 indicates that they read or write a common memory location in a way that requires their execution order to be maintained [130, 132]. We call R_1 the *source* and R_2 the *sink* of the dependence if R_1 must be executed before R_2. There are four types of data dependence:

True (flow) dependence occurs when S_1 writes a memory location that S_2 later reads.

Anti dependence occurs when S_1 reads a memory location that S_2 later writes.

Output dependence occurs when S_1 writes a memory location that S_2 later writes.

Input dependence occurs when S_1 reads a memory location that S_2 later reads. Input dependences do not restrict statement order.

Dependences may be either loop-independent or loop-carried. *Loop-independent* dependences occur on the same loop iteration; *loop-carried* dependences occur on different iterations of a particular loop. The *level* of a loop-carried dependence is the depth of the loop carrying the dependence [10]. Loop-independent dependences have infinite depth. The number of loop iterations separating the source and sink of the loop-carried dependence may be characterized by a dependence *distance* or *direction* [205]. The level of a dependence is determined by the first non-zero entry in its distance or direction vector.

For instance, dependence testing applied to the references in the following loop nest shows that a true dependence exists between the definition and use of array `A` with distance and direction vectors of $(1, 0, -1)$ and $(<, =, >)$, respectively. Since the outermost loop contributes the first nonzero element, it carries the true dependence.

1. Program analysis
   (a) Dependence analysis
   (b) Array section analysis
   (c) Data decomposition analysis
   (d) Partitioning analysis
   (e) Communication analysis
2. Program optimization
   (a) Program transformations
   (b) Parallelism optimizations
   (c) Communication optimizations
   (d) Run-time processing
3. Code generation
   (a) Initialization insertion
   (b) Program partitioning
   (c) Index translation
   (d) Message generation
   (e) Forall scalarization
   (f) Storage management

Figure 4.1 Fortran D Compiler Structure

```
do i = 1,n
   do j = 1,m
     do k = 1,l
       A(i+1,j,k-1) = A(i,j,k) + C
     enddo
   enddo
enddo
```

Dependence analysis is vital to shared-memory vectorizing and parallelizing compilers. We show that it is also highly useful for guiding compiler optimizations for distributed-memory machines. The prototype Fortran D compiler is being developed in the context of the ParaScope programming environment and incorporates the following analysis capabilities [35, 117].

Scalar data-flow analysis

Control flow, control dependence, and live range information are computed during the scalar data-flow analysis phase. In addition, scalar variables are labeled *private* with respect to a loop if their values are used only within the current loop iteration; this is useful for eliminating unnecessary computation and communication.

Symbolic analysis

Constant propagation, auxiliary induction variable elimination, expression folding, and loop invariant expression recognition are performed during the symbolic analysis phase of the Fortran D compiler. The goal of symbolic analysis is to provide a simplified program representation for the Fortran D compiler that improves program analysis and optimization. Consider the example below:

```
do ij = 1,len
  F(ij,n) = (F(ij,n)-TOT(ij)) / B(n)
enddo
```

If constant propagation is able to produce a constant value for n, or if n is identified as a loop-invariant expression, the Fortran D compiler can communicate $B(n)$ with an efficient *broadcast* preceding the loop.

Symbolic analysis also recognizes *reductions*, operations such as SUM, MIN, or MAX that are both commutative and associative. Once identified, reductions may be executed locally in parallel and the results combined efficiently using collective communication routines. Reduction operations are tagged during symbolic analysis for later use.

Dependence testing

Dependence testing determines the existence of data dependences between pairs of array references by examining their subscript expressions. Dependences exist only if it is possible for subscripts in corresponding dimensions simultaneously assume the same value. All data dependences found are characterized by their dependence level, as well as by distance and direction vectors. This information is used to guide subsequent compiler analysis and optimization.

4.2.2 Array Section Analysis

In addition to detecting data dependences between pairs of array references, the Fortran D compiler also performs analysis to summarize the sections accessed by each individual reference. Array sections are represented as regular sections and used to calculate array definitions and kills.

Regular section descriptors (RSDs)

Regular section descriptors (RSDs) are widely used in the Fortran D compiler as an internal representation. Originally developed to summarize array side effects across procedure boundaries, RSDs are compact representations of rectangular or right-triangular array sections and their higher dimension analogs [18, 37, 98]. They may also possess some constant step. The union and intersection of RSDs can be calculated inexpensively, making them highly useful for the Fortran D compiler. RSDs have also proven to be quite precise in practice, due to the regular memory access patterns exhibited by scientific programs. Figure 4.2 shows some examples of regular section descriptors.

In this thesis RSDs will be represented as $[\,l_i{:}u_i{:}s_i, \ldots]$, where l_i, u_i, and s_i indicate the lower bound, upper bound, and step of the ith dimension of the RSD, respectively. A default unit step is assumed if not explicitly stated. In loop nests or multidimensional arrays, the leftmost dimension of the RSD corresponds to the outermost loop or the leftmost array dimension. The other dimensions are listed in order.

Array Definitions

The principal use of RSDs in array section analysis is to compute the array sections defined by an assignment statement at each loop level. This information is useful for optimizations such as vector message pipelining, described later in Chapter 5. For instance, in the following program array section analysis builds an RSD for reference $X(j,i)$ at each loop level.

```
do i = 1,100
  do j = 1,100
    X(j,i) = ...
  enddo
enddo
```

At the innermost level [j,i] is produced. At the level of the j loop the RSD [1:100,i] results. At the outermost loop level, the compiler produces [1:100,1:100]. The RSDs are stored for use during array kill analysis and future optimizations.

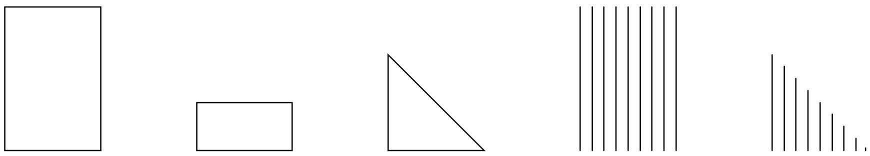


Figure 4.2 Regular Section Descriptors (RSDs)

Array Kills

Scalar data-flow analysis can detect private scalar variables. By combining control-flow information with array section information for array definitions, the Fortran D compiler can also calculate array kills [87]. This aids the compiler in detecting private arrays, and can significantly improve communication.

4.2.3 Data Decomposition Analysis

The Fortran D compiler requires a new type of program analysis to generate the proper program—it must determine the data decomposition for each reference to a distributed array.

Reaching decompositions

Because data access patterns may change between program phases, Fortran D provides dynamic data decomposition by permitting executable ALIGN and DISTRIBUTE statements to be inserted at any point in a program. This complicates the job of the Fortran D compiler, since it must know the decomposition of each array.

We define *reaching decompositions* to be the set of decomposition specifications that may reach an array reference aligned with the decomposition; it may be calculated in a manner similar to *reaching definitions*. The Fortran D compiler will apply both intra- and interprocedural analysis to calculate reaching decompositions for each reference to a distributed array. If multiple decompositions reach a procedure, run-time or node splitting techniques such as *cloning* may be required to generate the proper code for the program. Reaching decompositions is discussed in greater detail in Chapter 7.

4.2.4 Partitioning Analysis

After data decomposition analysis is performed, the program partitioning analysis phase of the Fortran D compiler divides the overall data and computation among processors. This is accomplished by first partitioning all arrays onto processors, then using the *owner computes* rule to derive the functional decomposition of the program. We begin with some useful definitions.

Iteration & index sets, RSDs

As described in the previous chapter, an *iteration set* is simply a set of loop iterations—it describes a section of the work space. An *index set* is a set of locations in an array—it describes a section of the data space. For the sake of efficiency, when generating communications the Fortran D compiler represents iteration and index sets in the form of regular section descriptors (RSDs).

Global vs. local indices

Because the Fortran D compiler creates SPMD node programs, all processors must possess the same array declarations. This forces all processors to adopt *local indices*. For instance, consider the following program and the node program produced when array A is block-distributed across four processors.

```
{* Original program *}      {* SPMD node program *}
REAL A(100)                 REAL A(25)
do i = 1, 100               do i = 1, 25
  A(i) = 0.0                   A(i) = 0.0
enddo                       enddo
```

The local indices for A on each processor are all [1:25], even though the equivalent global indices for A are [1:25], [26:50], [51:75], and [76:100] on processors 1 through 4, respectively. A similar conversion of loop indices may also occur, with the global loop indices [1:100] translated to the local loop indices [1:25].

Local index sets

As the first step in partitioning analysis, the Fortran D compiler uses the Fortran D statements associated with the reaching decomposition to calculate the *local index set* of each array—the local array section owned by every processor. This creates the data partition used in the program.

We illustrate the analysis required to partition the Jacobi code in Figure 4.3. For this and all future examples we will be compiling for a four processor machine. In the example, both arrays A and B are aligned identically with decomposition D, so they have the same distribution as D. Because the first dimension of D is local and the second dimension is block-distributed, the local index set for both A and B on each processor (in local indices) is [1:100,1:25].

Local iteration sets

Once the local index set for each array has been calculated, the Fortran D compiler uses it to derive the functional decomposition of the program. We define the *local iteration set* of a reference R on a processor to be the set of loop iterations that cause R to access data owned by the processor. It can be calculated by applying the inverse of the array subscript functions to the local index set of R, then intersecting the result with the iteration set of the enclosing loops.

The calculation of local index and iteration sets is vital to the partitioning analysis of the Fortran D compiler. When applying the *owner computes* rule, the set of loop iterations on which a processor must execute an assignment statement is exactly the local iteration set of the left-hand side (*lhs*). The Fortran D compiler can thus partition the computation by assigning iteration sets to each statement based on its *lhs*.

To demonstrate the algorithm, we will calculate the local iteration set for the assignment statement S_1 in the Jacobi example. Remember that the local index set of A is [1:100,1:25]. First we apply to it the inverse of the subscript functions of the *lhs*, $A(i,j)$. This yields the unbounded local iteration set [:,1:25,1:100]. The first entry is ":" since all iterations of the k loop access local elements of A. The inverse subscript functions cause the j and i loops to be mapped to [1:25] and [1:100], respectively.

Next we intersect the unbounded iteration set with the actual bounds of the enclosing loops, since these are the only iterations that actually exist. The iteration set of the loop nest (in global indices) is [1:$time$,2:99,2:99]. Converting it into local indices for each processor and performing the intersection yields the following local iteration sets for each processor (in local indices):

$$\begin{aligned} Proc(1) &= [\,1 : time,\ 2 : 25,\ 2 : 99\,] \\ Proc(2:3) &= [\,1 : time,\ 1 : 25,\ 2 : 99\,] \\ Proc(4) &= [\,1 : time,\ 1 : 24,\ 2 : 99\,] \end{aligned}$$

Similar analysis produces the same local iteration sets for statement S_2. Note how the local indices calculated for the local index set of each array have been used to derive the local indices for the local iteration set, using techniques described in Chapter 3.

Handling boundary conditions

Because alignment and distribution specifications in Fortran D are fairly simple, local index sets and their derived iteration sets may usually be calculated at compile time. In fact, in most regular computations local index and iteration sets are identical for every processor except for boundary conditions. When boundary conditions for each array dimension or loop are independent, as in the Jacobi example, the Fortran D compiler can store each boundary condition separately. This avoids the need to calculate and store a different result for each processor.

We may summarize independent boundary conditions for iteration or index sets as *pre, mid*, and *post* sets for each loop or array dimension. The mid set describes the interior uniform case. The pre and post iteration sets describe the boundary conditions encountered and their positions. These sets are represented in the Fortran D compiler by *augmented* iteration sets. Instead of a single section, each dimension of the augmented iteration set contains three component sections for the pre, mid, and post sets as well as their positions.

```
REAL A(100,100), B(100,100)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(:,BLOCK)
do k = 1,time
   do j = 2,99
     do i = 2,99
S1       A(i,j) = (B(i,j-1)+B(i-1,j)+B(i+1,j)+B(i,j+1))/4
     enddo
   enddo
   do j = 2,99
     do i = 2,99
S2       B(i,j) = A(i,j)
     enddo
   enddo
enddo
```

Figure 4.3 Jacobi

Because boundary conditions for iteration and index sets can be handled in the same manner, we will just discuss an example case for iteration sets. When partitioning the Jacobi example, the following *pre, mid*, and *post* iteration sets are calculated by the Fortran D compiler:

$$\left[\ 1 : time,\ \left\{ \begin{array}{l} pre = [2:25]\ @p_1 \\ mid = [1:25] \\ post = [1:24]\ @p_4 \end{array} \right\},\ 2:99\ \right]$$

In the *augmented* RSD representing the pre, mid, and post iteration sets, "@" indicates the position for each pre or post set. If an interior processor is causing a boundary condition, processors between it and the edge will not be assigned loop iterations. A pre or post iteration set may also be empty if that boundary condition does not exist.

The iteration set for each processor is calculated by taking the Cartesian product of the pre, mid, and post iteration sets for each dimension of the augmented iteration set. Unfortunately not all boundary conditions may be succinctly represented by augmented iteration sets. In the worst case the Fortran D compiler is forced to derive and store an individual index or iteration set for each processor.

Statement Groups

Whole programs and linear algebra codes tend to possess large diverse loop nests, with many imperfectly nested statements and triangular/trapezoidal loops. These complex loops increase the difficulty of partitioning the computation and calculating appropriate local and global loop indices and bounds. We found it useful in the Fortran D compiler to partition statements into *statement groups* during partitioning analysis. Statements are put into the same group for a given loop if their iteration sets for that loop and enclosing loops are the same. We mark a loop as *uniform* if all its statements belong to the same statement group. Uniform loop nests are desirable because they may be partitioned by reducing loop bounds; no explicit guards need to be inserted in the loop. Calculating statement groups can determine whether a loop nest is uniform and guide code generation for non-uniform loops.

An immediate application of statement groups is for *loop distribution*, a program transformation that separates independent statements inside a single loop into multiple loops with identical headers. If the Fortran D compiler detects a nonuniform loop nest, it attempts to distribute the loop around each statement group, producing smaller uniform loop nests. If loop distribution is prevented due to recurrences carried by the loop, the Fortran D compiler must insert explicit guards for each statement group to ensure they are executed only by the appropriate processor(s) on each loop iteration. Here statement groups also help because they identify groups of statements that can share the same guard expression. Statement groups also help guide the compiler in generating loop indices and bounds, as we later shown.

4.2.5 Communication Analysis

Once partitioning analysis determines how data and work are partitioned across processors, communication analysis determines which variable references cause nonlocal data accesses.

Computing nonlocal index sets

In this phase, all *rhs* references to distributed arrays are examined. For each *rhs*, the Fortran D compiler constructs the index set accessed by each processor. The index set is computed by applying the subscript functions of the *rhs* to the local iteration set assigned to the statement. The local index set is subtracted from the resulting RSD to check whether the reference accesses nonlocal array locations. If only local accesses occur, the *rhs* reference may be discarded. Otherwise the RSD representing the *nonlocal index set* accessed by the *rhs* is retained.

If boundary conditions exist for the local iteration set of the statement, the Fortran D compiler must compute the index set for each group of processors assigned different iteration sets. In the worst case the index set for each processor must be calculated separately.

We show how index sets are computed for the Jacobi example. We first consider the four *rhs* references to B in statement S_1. The iteration set boundary conditions cause processors to be separated into three groups. The group of interior processors, $Proc(2{:}3)$, have the local iteration set [1:*time*,1:25,2:99]. This derives the following index sets:

$$\begin{aligned} B(i, j-1) &= [\,2:99,\ 0:24\,] \\ B(i-1, j) &= [\,1:98,\ 1:25\,] \\ B(i+1, j) &= [\,3:100,\ 1:25\,] \\ B(i, j+1) &= [\,2:99,\ 2:26\,] \end{aligned}$$

Since the local index set for B is [1:100,1:25], $B(i-1,j)$ and $B(i+1,j)$ cause only local accesses and may be ignored. However, $B(i,j-1)$ and $B(i,j+1)$ access nonlocal locations [2:99,0] and [2:99,26], respectively. Both references are marked and their nonlocal index sets stored. Computing the index sets using the local iteration sets for the other two groups, $Proc(1)$ and $Proc(4)$, does not yield additional nonlocal references. Examination of the index sets for the *rhs* reference to $A(i,j)$ in statement S_2 show that only local accesses occur.

4.3 Program Optimization

The program optimization phase of the Fortran D compiler utilizes the results of program analysis to improve program performance. Its two primary goals are to exploit parallelism and reduce communication overhead. In this section, we present message vectorization, the key communication optimization performed by the Fortran D D compiler. Other communication and parallelism optimizations are deferred until Chapter 5.

4.3.1 Message Vectorization

A naive but workable algorithm known as *run-time resolution* inserts guarded *send* and/or *recv* operations directly preceding each nonlocal reference [38, 178, 212]. Unfortunately, this simple approach generates many small messages that prove extremely inefficient due to communication overhead [178].

The most basic communication optimization performed by the Fortran D compiler is *message vectorization*. It uses the level of loop-carried data dependences to calculate whether communication may be legally performed at outer loops. This replaces many small messages with one large message, reducing both message startup cost and latency. Message vectorization forms the basis of our algorithm for introducing and placing communication for nonlocal accesses.

Algorithm

We use the message vectorization algorithm developed by Balasundaram *et al.* and Gerndt to calculate the appropriate loop level to insert messages for nonlocal references [16, 80]. We define the *commlevel* for loop-carried dependences to be the level of the dependence. For loop-independent dependences we define it to be the level of the deepest loop common to both the source and sink of the dependence.

We also extend the algorithm to guide message generation by classifying messages for each reference as one of the following message types:

- *independent*
- *carried-all*
- *carried-part*

The message type will determine the placement of communication, and will be discussed in greater detail in Section 4.4.5.

To vectorize messages for a *rhs* reference R with a nonlocal index set, we examine all cross-processor true dependences with R as the sink. The deepest commlevel of all such dependences determines the loop level at which the message may be vectorized. If the deepest commlevel is for a dependence carried by loop L, we insert a message tag for R marked *carried* at the header for loop L. This tag indicates that nonlocal data accessed by R must be communicated between iterations of loop L.

Otherwise the deepest commlevel is for a loop-independent dependence with loop L as the deepest loop enclosing both the source and sink. We insert a tag for R marked *independent* at the header of the next deeper loop enclosing R at level $L+1$, or at R itself if no such loop exists. This tag indicates that nonlocal data accessed by R must be communicated at this point on each iteration of loop L. Additionally, the Fortran D compiler may move this tag to any statement in loop L between the source and the sink of the dependence in order to combine messages arising from different references. We illustrate the message vectorization algorithm with three examples.

Example 1: Jacobi

First we examine the Jacobi code in Figure 4.3. In the communication analysis phase, we have already determined that for the given data decomposition only the *rhs* references $B(i,j-1)$ and $B(i,j+1)$ from S_1 access nonlocal locations. The only cross-processor true dependences incident on these references are from the definition to B in S_2. These dependences are carried on the k loop, so we insert their tags (labeled carried) at the header of the k loop. The code generation phase will later insert messages for those references inside the k loop.

Example 2: Successive over-relaxation (SOR)

In the code for SOR in Figure 4.4, communication analysis discovers that the *rhs* references $A(i+1,j)$ and $A(i-1,j)$ have nonlocal index sets. Dependence analysis shows that the reference $A(i+1,j)$ has a cross-processor true dependence carried on the k loop, so we insert its tag (labeled carried) at the k loop header. The deepest loop-carried true dependence for reference $A(i-1,j)$ is carried on the i loop, so we insert its tag (also labeled carried) at the i loop header.

```
REAL A(100,100)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(BLOCK,:)
do k = 1,time
   do j = 2,99
     do i = 2,99
       A(i,j) = (ω/4)*(A(i,j-1)+A(i-1,j)+
           A(i+1,j)+A(i,j+1))+(1-ω)*A(i,j)
     enddo
   enddo
enddo
```

Figure 4.4 Successive Over-Relaxation (SOR)

```
   REAL V(N,N)
   DECOMPOSITION D(N,N)
   ALIGN V with D
   DISTRIBUTE D(BLOCK,BLOCK)
   do k = 1,time
      {* compute red points *}
      do j = 3,N-1,2
        do i = 3,N-1,2
S1         V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
              V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
        enddo
      enddo
      do j = 2,N-1,2
        do i = 2,N-1,2
S2         V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
              V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
        enddo
      enddo
      {* compute black points *}
      do j = 3,N-1,2
        do i = 2,N-1,2
S3         V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
              V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
        enddo
      enddo
      do j = 2,N-1,2
        do i = 3,N-1,2
S4         V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
              V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
        enddo
      enddo
   enddo
```

Figure 4.5 Pointwise Red-black SOR

Example 3: Red-black SOR

In the code in Figure 4.5, communication analysis discovers that all *rhs* references except $V(i,j)$ possess nonlocal index sets. However, dependence analysis shows that the only cross-processor true dependences incident on the *rhs* references for statements S_1 and S_2 are carried on the k loop from S_3 and S_4. The tags for these references (labeled as carried) are inserted at the header of the k loop. During code generation phase they will generate messages in the k loop.

For statements S_3 and S_4, dependence analysis shows that the only cross-processor true dependences incident on their *rhs* references are loop-independent dependences from S_1 and S_2. Their commlevel is set to the k loop because it is the deepest loop enclosing both the source and sink of these dependences. We insert tags (labeled independent) for all *rhs* references in S_3 at its enclosing j loop, since it is the next loop deeper than k enclosing S_3.

Similar analysis causes us to insert tags (labeled independent) for all *rhs* references in S_4 at its enclosing j loop. As an additional optimization, we can move these tags to the j loop enclosing S_3 to combine these messages. This is legal since we are moving tags to a statement that is at the same loop level and between the source and sink of the dependence. In the code generation phase these tags will cause vectorized messages to be generated before the j loop, to be executed on each iteration of the k loop.

4.4 Code Generation

Once program analysis and optimization is complete, the code generation phase of the Fortran D compiler performs the source-to-source translation that actually *instantiates* the program partitioning and com-

```
REAL A(100,25), B(100,0:26)
my$p = myproc() {* 0...3 *}
{* if (my$p .eq. 0) lb1 = 2 else lb1 = 1 *}
{* if (my$p .eq. 3) ub1 = 24 else ub1 = 25 *}
lb1 = max((my$p*25)+1,2)-(my$p*25)
ub1 = min((my$p+1)*25,99)-(my$p*25)
do k = 1,time
   if (my$p .gt. 0) send(B(2:99,1), my$p-1)
   if (my$p .lt. 3) recv(B(2:99,26), my$p+1)
   if (my$p .lt. 3) send(B(2:99,25), my$p+1)
   if (my$p .gt. 0) recv(B(2:99,0), my$p-1)
   do j = lb1,ub1
     do i = 2,99
       A(i,j) = (B(i,j-1)+B(i-1,j)+B(i+1,j)+B(i,j+1))/4
     enddo
   enddo
   do j = lb1,ub1
     do i = 2,99
       B(i,j) = A(i,j)
     enddo
   enddo
enddo
```

Figure 4.6 Generated Jacobi

munication. It utilizes the results of previous analysis (local index and iteration sets, RSDs, collective communication) to transform the program text, modifying array and loop bounds, inserting guards and communication, loop and array indices, in order to generate the actual SPMD node program with explicit message-passing.

4.4.1 Initialization Insertion

First, the Fortran D compiler inserts a number of initialization statements that are used to gather information at run-time. The statements initialize values for *n$proc*, the total number of processors; *my$p*, the local processor number, from 0 to *n$proc*-1, etc.

4.4.2 Program Partitioning

Most optimizations increase the amount of temporary storage required by the program. Compile-time partitioning of the data so that each processor allocates memory only for array sections owned locally is fundamental. Otherwise the problem size is limited by the amount of data that can be place on a single processor. The Fortran D compiler instantiates the data partition calculated during program analysis by modifying the array declarations, reducing the size of the array on each processor.

4.4.3 Computation Partitioning

During partitioning analysis, the Fortran D compiler applied the *owner computes* rule to calculate the local iteration set for each statement. One of the goals for code generation is to modify the program to ensure that each processor only executes a statement on loop iterations in its local iteration set.

Loop bounds reduction and *guard introduction* are the two program transformations used to instantiate the computation partition. Loop bounds reduction is the most common and efficient case. The Fortran D compiler first reduces the loop bounds so that each processor only executes iterations in the unioned local iteration sets of all statements in the loop. It then inserts code to calculate boundary conditions, as shown the bounds generated for the j loop in Figure 4.6. In these cases the loop index variables have also been localized.

	LowerLoopBound()	*UpperLoopBound*()
BLOCK	MAX((my$p*bk)+1,L)–my$p*bk	MIN((my$p+1)*bk,U)–my$p*bk
CYCLIC	lb$ = ((L–1)/n$p)+1 if (my$p < MOD(L–1,n$p)) lb$1 = lb$1 + 1	ub$ = ((U–1)/n$p)+1 if (my$p > MOD(U–1,n$p)) ub$1 = ub$1 – 1

Table 4.1 Generating Loop Bounds

	LocalLoopIndex()	*GlobalLoopIndex*()
BLOCK	i – my$p*bk	i + my$p*bk
CYCLIC	(i–1)/bk+1	((i–1)*bk)+1+my$p

Table 4.2 Generating Local and Global Indices

Generating loop index variables and bounds are more complex for cyclic than block distributions. Note that in the Fortran D compiler this process is derived from the data decomposition, so arrays with different data distributions may require distinct local index variables. The compiler assigns the new indices and bounds to compiler-generated variables, and places the assignments where they can be reused. Table 4.1 shows the formulae used by the Fortran D compiler to compute indices and bounds, where:

n$p	=	total number of processors
my$p	=	the local processor number (0 ... n$p-1)
bk	=	block size of the local array section (array size/n$p)
L,U	=	original lower & upper loop bounds

For simplicity, we assume that Fortran arrays begin at 1 and processor IDs start at 0.

Not all loops may be partitioned in this manner. With multiple statements in the loop, the local iteration set of a statement may be a subset of the reduced loop bounds. For these statements the compiler needs to add explicit guards based on membership tests for the local iteration set of the statement [38, 178, 212].

In other cases, the compiler may not be able to localize loop bounds and indices because a processors executes some statement on all iterations of the loop. Statement groups formed during partitioning analysis help detect this situation. To decide whether it may localize indices and bounds for a given loop, the Fortran D compiler examines all statement groups in the loop. If any statement group is executed by all processors, the loop indices and bounds remain global. Explicit guards and index translation will then be necessary for the other statement groups.

4.4.4 Index Translation

After partitioning the data and computation in the output program, The compiler converts global loop indices and loop bounds into local values where necessary so that local array sections are accessed. Occasions also arise where both global and local indices are required. In these cases the compiler will convert from one or the other as appropriate, using statement groups to help guide generation of loop index variables. Table 4.2 displays the formulae employed by the Fortran D compiler to convert between local and global indices.

For instance, in Figure 4.7 array B is partitioned blockwise across n$p processors. Statements S_1 and S_2 possess different iteration sets and are put into separate statement groups. In this code the i loop cannot be localized because on each iteration all processors execute S_2. The indices for the outer i loop must thus remain global. Since B is distributed, only one processor executes S_1. The compiler must insert an explicit guard and generate a local index i$ for indexing into B. In comparison, each processor executes only some iterations of the inner j loop. The compiler can thus localize j, reducing its bounds to lb$1 and ub$1 so that each processor only executes local iterations. However, since the global value of j is used by S_2, the compiler must generate the global index value j$ from the local index j. In the following sections we examine some cases where explicit index translation is required.

	{∗ Original Program ∗}	{∗ Compiler Output ∗}
	`REAL B(n)`	`REAL B(n/n$p)`
	`do i =` L_i, U_i	`lb$1 =` *LowerLoopBound*(L_j)
S_1	`B(i) =` $\mathcal{F}_1$`(i)`	`ub$1 =` *UpperLoopBound*(U_j)
	`do j =` L_j, U_j	`do i =` L_i, U_i
S_2	`B(j) =` $\mathcal{F}_2$`(j)`	`i$ =` *LocalLoopIndex*`(i)`
	`enddo`	`if (Owner(B(i))) B(i$) =` $\mathcal{F}_1$`(i)`
	`enddo`	`do j = lb$1,ub$1`
		`j$ =` *GlobalLoopIndex*`(j)`
		`B(j) =` $\mathcal{F}_2$`(j$)`
		`enddo`
		`enddo`

Figure 4.7 Loop Indices and Bounds Generation

Local to Global Conversion

In most cases encountered by the Fortran D compiler, loops are localized, partitioned during compilation, and translated into local indices. If their index variables appear in positions other than subscripts in distributed array dimensions, they must be translated back into global indices. This is achieved by applying the inverse of the distribution function applied to the loop. For loops iterating over block-distributed arrays, simply adding an offset based on the processor number is sufficient.

For instance, consider how the Fortran D compiler would translate the indices in the example program in Figure 4.8. Both arrays X and Y originally possess 256 elements. They are distributed blockwise across four processors, assigning a contiguous block of 64 elements to each of the four processors. Both i loops iterating over X and Y are converted to local indices as well.

At statement S_1, the local index variable i appears outside the subscript of a distributed array. Instead, it is used to calculate a value for $X(i)$. To convert i back to its global value is simple because i was derived from a block-distributed array. Since the translation from global to local indices was subtraction of the processor offset, the compiler can just calculate the offset and add it back to the local index value to construct the global index value. In the program, the Fortran D compiler inserts code to calculate the offset *i$off* for the given processor. This is then added to the occurrence of i in the *rhs* expression. The result is shown in Figure 4.9.

Global to Local Conversion

If a loop has not been localized and its index variable appears in the subscript of a distributed array dimension, it must be explicitly converted into an local index. Another situation that requires performing the same conversion is the presence of constant subscripts in distributed array dimensions of the *lhs* of an assignment statement. Because these *lhs* references will not be changed into references to the message buffer, they need to be translated into local indices. This is accomplished by applying the distribution function to the subscript value at either compile-time or run-time. Constant subscripts in *rhs* array references do not need to be checked because they will be replaced by references to some message buffer.

Consider how the Fortran D compiler translates the program in Figure 4.8. At statements S_2 and S_3, constants are subscripts in the distributed dimension of Y, the *lhs* array reference. Because the subscripts in the distributed dimension of the array Y are constant for S_2 and S_3, the owner computes rule implies that only one processor will execute each statement. The compiler determines which processor owns the *lhs*, then converts the global index into the local index for that processor.

For S_2, processor 0 owns $Y(1)$, so the conversion leaves the index unchanged. For S_3, processor 3 owns $Y(256)$, so the conversion transforms the global index 1 into the local index 64 on processor 3. The result is shown in Figure 4.9. As can be seen, constant subscripts in the *rhs* may be ignored since they have been replaced by references to message buffers.

```
     {* Fortran D Program *}
     REAL X(256), Y(256)
     PARAMETER (n$proc = 4)
     DECOMPOSITION D(256)
     ALIGN X, Y with D
     DISTRIBUTE D(BLOCK)
     do i = 1,256
S1     X(i) = F1(i)
     enddo
S2   Y(1) = F2(X(2),X(256))
S3   Y(256) = F2(X(1),X(255))
     do i = 2,255
       Y(i) = F2(X(i+1),X(i-1))
     enddo
```

Figure 4.8 Index Translation

```
     {* Compiler Output *}
     REAL X(64), Y(0:65)
     my$p = myproc() {* 0...3 *}
     lb1 = max((my$p*25)+1,2)-(my$p*25)
     ub1 = min((my$p+1)*25,99)-(my$p*25)
     i$off = my$p*25
     do i = 1,64
       X(i) = F1(i+i$off)
     enddo
     if (my$p .eq. 3) send(X(64),0)
     if (my$p .eq. 0) send(X(1),3)
     if (my$p .eq. 0) then
       recv(r$buf(1),3)
       Y(1) = F2(X(2),r$buf(1))
     endif
     if (my$p .eq. 3) then
       recv(r$buf(1),3)
       Y(64) = F2(r$buf(1),X(63))
     endif
     do i = lb1,ub1
       Y(i) = F2(X(i+1),X(i-1))
     enddo
```

Figure 4.9 Index Translation—Compiler Output

4.4.5 Message Generation

The Fortran D compiler uses information calculated in the communication analysis and optimization phases to guide message generation. Non-blocking *send* and blocking *recv* are inserted for the following types of messages:

Independent Messages

Messages for references tagged at loop headers for loop-independent cross-processor dependences are labeled as *independent*. For these messages the Fortran D compiler inserts calls to *send* and *recv* primitives preceding the loop header. For messages tagged at individual references, the Fortran D compiler inserts *send* and *recv* in the body of the loop preceding the reference. All messages are guarded so that the owners execute *send* and recipients execute *recv*. To calculate the data that must be sent, the Fortran D compiler builds the RSD for the reference at the loop level that the message is generated. This represents data sent on each loop iteration.

```
      {* Example Computation *}
      do k = 1,M
        do i = 1,N
δ∞        A(i) = B(i+1)
δk              + A(i+1)
δi              + A(i-1)
        enddo
      enddo
```

```
{* Communication Placement *}
send & recv B(i+1)
do k = 1,M
  send & recv A(i+1)
  recv A(i-1)
  do i = 1,N/P
    A(i) = B(i+1) + A(i+1) + A(i-1)
  enddo
  send A(i-1)
enddo
```

Figure 4.10 Message Vectorization According To Message Type

Carried-all and Carried-part Messages

The situation is more complex for messages representing loop-carried dependences. To calculate the data that must be communicated, we build the RSD for each *rhs* reference at the level of the loop L carrying the dependence. If iterations of L are executed by all processors, the reference is labeled as *carried-all*. The Fortran D compiler inserts calls to *send* and *recv* primitives inside the loop header for L, at the beginning of the loop body.

If the iterations of L are partitioned across processors, the reference is labeled as *carried-part*. In this case loop-carried messages represent data synchronization. The compiler inserts calls to *recv* preceding loop L, since they occur before the local iterations of L. Similarly, calls to *send* are inserted after L, since they are executed after the local iterations of L.

If both independent and carried-part messages are generated at the same loop header, *send* and *recv* primitives for independent messages are inserted first further away from the header. Communication for carried-part messages are inserted afterwards, immediately preceding the loop header in order to avoid deadlock. This ordering allows communication for independent messages to take place in parallel, before communicating data corresponding to carried-part messages.

Communication Placement

Figure 4.10 illustrates the communication generated for these three message types. Assume that A and B are distributed block-wise, causing the iterations of the i loop to be partitioned among processors. Messages are required for all three *rhs* references and classified based on the level and type of true dependences. The message for $B(i+1)$ is of type independent at the k loop, causing communication to be inserted preceding the k loop header. The message for $A(i+1)$ is of type carried-all at the k loop, so communication is inserted at the head of the loop body. Finally, the message for $A(i-1)$ is of type carried-part at the i loop, so communication is inserted before and after the i loop.

Additional Examples

We illustrate message generation for several additional examples from previous sections. First, messages are labeled independent in Red-black SOR. They thus cause communication to be inserted preceding the loop nests, such as the loops enclosing statements S_3 and S_4 in Red-black SOR, displayed in Figure 4.5.

For the Jacobi code in Figure 4.3, recall that the k loop carries true dependences for the *rhs* references in S_1. These messages were tagged at the k loop header as carried. We first compute RSDs for the data that need to be communicated. Boundary conditions cause three RSDs to be generated for each *rhs* reference. Below are the RSDs for the reference $B(i, j+1)$ at the k loop level.

$$
\begin{aligned}
Proc(1) &= [\,2:99,\ 3:26\,] \\
Proc(2:3) &= [\,2:99,\ 2:26\,] \\
Proc(4) &= [\,2:99,\ 2:25\,]
\end{aligned}
$$

```
REAL A(0:26,100)
my$p = myproc() {* 0...3 *}
lb₁ = max((my$p*25)+1,2)-(my$p*25)
ub₁ = min((my$p+1)*25,99)-(my$p*25)
do k = 1,time
   if (my$p .gt. 0) send(A(1,2:99), my$p-1)
   if (my$p .lt. 3) recv(A(26,2:99), my$p+1)
   do j = 2,99
     if (my$p .gt. 0) recv(A(0,j), my$p-1)
     do i = lb₂, ub₂
       A(i,j) = (ω/4)*(A(i,j-1)+A(i-1,j)+
           A(i+1,j)+A(i,j+1))+(1-ω)*A(i,j)
     enddo
     if (my$p .lt. 3) send(A(25,j), my$p+1)
   enddo
enddo
```

Figure 4.11 Generated SOR

We subtract the local index set from these RSDs to determine the RSDs for the nonlocal index set. The nonlocal RSDs for $Proc(1)$ and $Proc(2{:}3)$ are both [2:99, 26] and are therefore combined. The RSD for $Proc(4)$ consists of only local data and is discarded.

The sending processor is determined by computing the owners of the section [2:99,26] @ $Proc(1{:}3)$, resulting in $Proc(2{:}4)$ sending data to their left processors. To compute the data that must be sent, we translate the local indices of the receiving processors to that of the sending processors, obtaining the section [2:99,26−25] = [2:99,1]. Since loop k is executed by all processors, the messages are inserted at the beginning of the loop body. Messages for $B(i, j-1)$ are calculated in a similar manner. The communication generated is shown in Figure 4.6.

Now consider the SOR code depicted in Figure 4.4. Dependences for $A(i+1, j)$ are carried on the k loop, causing vectorized messages to be inserted at the beginning of the k loop body as in Jacobi. The compilation of $A(i-1, j)$ is more complicated. Boundary conditions and dependences carried by the i loop cause the following three RSDs to be generated at the level of the i loop.

$$\begin{aligned} Proc(1) &= [\,1:24,\ j\,] \\ Proc(2:3) &= [\,0:24,\ j\,] \\ Proc(4) &= [\,0:23,\ j\,] \end{aligned}$$

The local index set is subtracted from these RSDs to determine the RSDs for the nonlocal index set, producing the empty set for $Proc(1)$. The nonlocal RSDs for both $Proc(2{:}3)$ and $Proc(4)$ are [0,j] and are combined. This shows that processors 2 through 4 require data from their left neighbor. Iterations of the i loop are partitioned, alerting the Fortran D compiler to the fact that the *send* for $A(i-1, j)$ occurs after the last local i loop iteration, and the *recv* occurs before the first local i loop iteration. It thus inserts the *recv* before the i loop and the *send* after the i loop, resulting in the code shown in Figure 4.11.

Collective Communication

During communication optimization, opportunities for reductions and collective communication have been marked separately. Instead of individual calls to *send* and *recv*, the Fortran D compiler inserts calls to the appropriate collective communication routines. Additional communication is also appended following loops containing reductions to accumulate the results of each reduction.

Run-time Processing

Run-time processing is applied to computations whose nonlocal data requirements are not known at compile time. An *inspector* [155] is constructed to preprocess the loop body at run-time to determine what nonlocal data will be accessed. This in effect calculates the *receive* index set for each processor. A global transpose operation between processors is then used to calculate the *send* index sets. Finally, an *executor* is built to actually communicate the data and perform the computation.

An inspector is the most general way to generate *send* and *receive* sets for references without loop-carried true dependences. Despite the expense of additional communication, experimental evidence from several systems show that it can improve performance by grouping communication to access nonlocal data outside of the loop nest, especially if the information generated may be reused on later iterations [123, 155].

The inspector strategy is not applicable for unanalyzable references causing loop-carried true dependences. In this case the Fortran D compiler inserts guards to resolve the needed communication and program execution at run-time [38, 178, 212].

4.4.6 Forall Scalarization

Another responsibility of the Fortran D compiler is to convert FORALL loops into DO loops, inserting additional code where necessary to maintain the semantics of the FORALL loop. This process, known as scalarization, generally involves making temporary copies of *rhs* values where necessary to prevent them from being overwritten before they are used. Optimizations known as *sectioning* may be used to reduce the amount of buffering needed [8, 11]. FORALL scalarization is discussed in greater detail later in Chapter 8.

4.4.7 Storage Management

One of the major responsibilities of the Fortran D compiler is to select and manage storage for all nonlocal array references. The simplest approach is to allocate full-sized arrays on each processor. This requires the least change to the program, but wastes most of the available memory and limits the total problem size. More sophisticated storage management techniques manipulate both the location and lifetimes of nonlocal storage in order to reduce memory use and code complexity. Storage management may be separated into two phases, selection of the desired storage types, and coordinating their usage. The Fortran D compiler analyzes nonlocal references and selects one of the following three storage schemes.

Overlaps

Overlaps are expansions of local array sections to accommodate neighboring nonlocal elements [80]. For regular stencil computations, overlaps are useful because they allow the generation of clear and readable code. For other computations overlaps are inefficient, because all array elements between the local section and the one accessed must also be part of the overlap, even if they are not used. Storing nonlocal data in overlaps also requires more storage than other means. This is because overlaps are permanent and specific to individual arrays, and thus cannot be reused in other parts of the computation.

Buffers

Buffers avoid the contiguous and permanent nature of overlaps. They are useful when storage for nonlocal data must be reused, or when the nonlocal area is bounded in size but not near the local array section. For instance, a buffer can be used to store the pivot column for Gaussian elimination with partial pivoting, without requiring the entire array as an overlap. Buffers have the advantage that nonlocal array elements do not have to be copied to a data array, but can be accessed directly by modifying nonlocal references in the program. The lifetime of a buffer depends on how quickly it will be reused for different arrays or later parts of the computation.

Hash tables

Hash tables may be used when the set of nonlocal elements accessed is sparse, as for many irregular computations. They also provide a quick lookup mechanism for arbitrary sets of nonlocal values [108, 156].

Maintenance

Selecting storage types is fairly simple. Stencil computations that result in nonlocal accesses at boundaries of local array sections are satisfied by providing overlaps. The compiler allocates buffers for nonstencil computations or stencils that are result in nonlocal accesses not contiguous to the local array section. Hash tables are selected for nonlocal accesses made by sparse computations.

Once the type of storage is chosen, the compiler needs to perform analysis to determine the total amount of storage needed as overlaps, persistent buffers, or temporary buffers. The extent of all RSDs representing nonlocal accesses produced during message generation are examined to select the appropriate storage type for each array. If overlaps have been selected, array declarations are modified to take into account storage for nonlocal data. For instance, array declarations in the generated code in Figures 4.6 and 4.11 have been extended for overlap regions.

If buffers are used, additional buffer array declarations are inserted. Finally, all nonlocal array references in the program are modified to reflect the actual data location selected so that they access the appropriate buffer instead. This may require also linearizing the nonlocal reference to the buffer.

4.5 Discussion

We have presented the basic structure of the Fortran D compiler. It performs three major functions—program analysis, program optimization, and code generation. New analysis techniques are required to compile shared-memory programs for distributed memory machines. We also described several internal data structures used in the compilation process. The Fortran D compiler utilizes a compilation strategy based on the concept of data dependence that unifies and extends previous techniques. The partitioning and communication are derived from the Fortran D data decomposition specifications via the owner computes rule. This information is used to perform a source-to-source translation, generating an efficient message-passing SPMD Fortran 77 program for execution on the nodes of the distributed-memory machine.

Chapter 5

Compiler Optimizations

A number of Fortran D compiler optimizations are introduced and classified. Program transformations modify the program execution order to enable optimizations. Communication optimizations can be separated into two classes, those that reduce communication overhead by decreasing the number of messages, and those that hide communication overhead by overlapping the cost of remaining messages with local computation. Parallelism optimizations restructure the computation or communication to increase the amount of useful computation that may be performed in parallel. Optimizations improve computation partitioning by eliminating explicit guards. Storage requirements are reduced by partitioning data across processors and sending messages in smaller blocks.

5.1 Introduction

The goal of the Fortran D compiler is to generate a parallel program with low communication overhead and storage requirements for MIMD distributed-memory machines. This chapter introduces and classifies a number of advanced compiler optimizations to achieve this goal. These optimizations are shown in Figure 5.1. Communication optimizations can reduce communication overhead by eliminating messages or hide communication overhead by overlapping communication and computation. Parallelism optimizations exploit the parallelism in fully parallel and pipelined computations. Optimizations improve partitioning by reducing the number of run-time tests required to partition computation. Optimization may also reduce the amount of storage required for nonlocal data.

As the Fortran D compiler is a second generation research project, many of its optimizations have been discussed in the literature. However, previous researchers tend to develop each optimization in isolation, without evaluating their effectiveness or considering their interaction with other elements of the compiler. We find that integrating these optimizations in a single compiler is difficult but feasible. In the Fortran D compiler we design a framework for classifying an integrating optimizations, as well as adapt and extend a number of existing optimizations. In Figure 5.1, we use $\circ$, $\bullet$, and $\star$ to mark the extent of our contribution to each optimization technique.

Shared-memory parallelizing compilers apply program transformations to expose or enhance parallelism in scientific codes, using dependence information to determine their legality and profitability [10, 117, 132, 205]. We have found that transformations such as loop interchange, fusion, distribution, alignment, and strip-mining to be also quite useful for optimizing Fortran D programs. The legality of each transformation is determined in exactly the same manner as for shared-memory programs, since the same execution order must be preserved in order to retain the meaning of the original program. However, their profitability criteria are now totally different.

In the remainder of this chapter, we describe each optimization and provide motivating examples using a small selection of scientific program kernels adapted from the Livermore Kernels and finite-difference algorithms [151]. They contain *stencil computations* and reductions, techniques commonly used by scientific programmers to solve partial differential equations (PDEs) [31, 74]. For clarity we ignore boundary conditions and use constant loop bounds and machine size in the examples, though this is not required by the optimizations. We have also equalized sizes for two dimensional problems used in our Livermore examples (*e.g.,* $n \times n$ instead of $7 \times n$ data arrays).

1) Reduce Communication Overhead
   - • Message Vectorization
   - • Message Coalescing
   - • Message Aggregation
   - ◦ Collective Communication
   - ◦ Run-time Processing
   - • Relax Owner Computes Rule
   - • Replicate Computation

2) Hide Communication Overhead
   - ◦ Message Pipelining
   - ⋆ Vector Message Pipelining
   - • Iteration Reordering
   - • Unbuffered Messages

3) Exploit Parallelism
   - ◦ Partition Computation
   - • Private Variables
   - • Reductions and Scans
   - • Dynamic Data Decomposition
   - ⋆ Pipeline Computations

4) Improve Partitioning
   - • Loop Bounds Reduction
   - • Loop Distribution
   - • Loop Align

5) Reduce Storage
   - ◦ Partition Data
   - • Message Blocking

Where:
- ◦ indicates existing technique
- • indicates improvement or adaptation of existing technique
- ⋆ indicates new technique

Figure 5.1 Fortran D Compiler Optimizations

5.2 Reducing Communication Overhead

We begin our description of compiler optimizations with those that attempt to reduce communication overhead. Many compiler optimizations focus on reducing communication overhead. Computer architects usually characterize communication by latency and bandwidth. For evaluating compiler optimizations, we find it convenient to divide communication overhead into three components:

- T_{start}, the startup time to send & receive messages.
- $T_{copy}(n)$, the time to copy a message of size n into & out of the program address space.
- $T_{transit}(n)$, the transit time for a message of size n between processors.

We assume T_{start} is relatively fixed with respect to message size, but that both T_{copy} and $T_{transit}$ grow with n. Using this communication model we can cast latency as $T_{start} + T_{copy}(1) + T_{transit}(1)$ and bandwidth as $n/(T_{copy}(n) + T_{transit}(n) - T_{transit}(1))$.

We begin with optimizations to reduce T_{start}, the startup cost incurred to access nonlocal data. For most MIMD distributed-memory machines, the cost to send the first byte is significantly higher than the cost for additional bytes. For instance, the Intel iPSC/860 requires approximately 95 μsec to send one byte versus .4 μsec for each additional byte [26]. The following optimizations seek to reduce T_{start} by *eliminating* messages, reducing the total number of messages sent. In the next section we describe optimizations that try to *hide* T_{copy} and $T_{transit}$ by overlapping communication with computation.

5.2.1 Message Vectorization

Message vectorization has been discussed in the previous chapter. Basically, it uses the results of data dependence analysis [10, 132] to combine element messages into vectors. Message vectorization is a loop-based optimization. It extracts communication from within loops, replacing sending a message per loop iteration to one vectorized message preceding the loop.

Recall that message vectorization first calculates *commlevel*, the level of the deepest loop-carried true dependence or loop enclosing a loop-independent true dependence. This determines the outermost loop where element messages resulting from the same array reference may be legally combined [16, 80]. Vectorized nonlocal accesses are represented as RSDs and stored at the loop at commlevel. They eventually generate messages at loop headers for loop-carried RSDs and in the loop body for loop-independent RSDs.

5.2.2 Message Coalescing

Once nonlocal accesses are vectorized at outer loops, the compiler applies *message coalescing* to avoid communicating redundant data. It compares RSDs from different references to the same array, merging RSDs that contain overlapping or contiguous elements, if the RSDs possess the same communication type (*e.g.,* shift, broadcast). If two overlapping RSDs cannot be be coalesced without loss of precision, they are split into smaller sections in a manner that allows the overlapping regions to be merged precisely. The resulting RSDs are aggregated.

For instance, consider the kernel in Figure 5.2. Communication analysis discovers that references U(k+1)...U(k+6) access nonlocal data. These references do not cause loop-carried true dependences, so their nonlocal RSDs are vectorized outside both the *l* and *k* loops, resulting in the RSDs [26:26]...[26:31]. These RSDs may be coalesced without loss of precision, resulting in [26:31]. Calls to copy routines are inserted during code generation to pack noncontiguous data into message buffers, but is not needed for this example since the data is contiguous. Finally, explicit *send* and *recv* statements are placed at loops headers to communicate nonlocal data.

Message coalescing can also be applied across loop nests. RSDs representing nonlocal data to be communicated for a loop nest may be combined with RSDs at previous loop nests at the same level, if the data represented in the RSD has not be redefined by an intervening write. During dependence analysis, the Fortran D compiler summarizes array definitions using RSDs. During optimization, the compiler steps backwards through the program for each RSD, seeking targets for message coalescing and aggregation. The search halts if a merge is found, the RSD intersects an RSD representing an intervening definition, or if no more statements exist at the appropriate loop level.

```
{* Fortran D Program *}
REAL U(100), X(100), Y(100), Z(100), R, T
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN U, X, Y, Z with D
DISTRIBUTE D(BLOCK)
do l = 1,time
  do k = 1,94
    X(k) = F(Z(k),Y(k),U(k)...U(k+6))
  enddo
enddo

{* Compiler Output *}
REAL U(31), X(25), Y(25), Z(25), R, T
my$p = myproc() {* 0...3 *}
if (my$p .gt. 0) send(U(1:6),my$p-1)
if (my$p .lt. 3) recv(U(26:31),my$p+1)
do l = 1,time
  do k = 1,25
    X(k) = F(Z(k),Y(k),U(k)...U(k+6))
  enddo
enddo
```

Figure 5.2 Livermore 7–Equation of State Fragment

```
{* Fortran D Program *}
REAL ZP(100,100), ZQ(100,100), ZM(100,100)
REAL ZR(100,100), ZZ(100,100), ZA(100,100)
REAL ZU(100,100), ZV(100,100), ZB(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN ZA, ZB, ZM, ZP, ZQ, ZR, ZU, ZV, ZZ with D
DISTRIBUTE D(BLOCK,:)
do l = 1,time
  do k = 2,99
    do j = 2,99
      ZA(j,k) = F1(ZP(j-1,k),ZQ(j-1,k),ZM(j-1,k),ZR(j-1,k),...)
      ZB(j,k) = F2(ZP(j-1,k),ZQ(j-1,k),...)
    enddo
  enddo
  do k = 2,99
    do j = 2,99
      ZU(j,k) = F3(ZZ(j-1,k),ZZ(j+1,k),ZA(j-1,k),...)
      ZV(j,k) = F4(ZR(j-1,k),ZR(j+1,k),ZA(j-1,k),...)
    enddo
  enddo
  do k = 2,99
    do j = 2,99
      ZR(j,k) = F5(ZR(j,k),ZU(j,k))
      ZZ(j,k) = F6(ZZ(j,k),ZV(j,k))
    enddo
  enddo
enddo
```

Figure 5.3 Livermore 18—Explicit Hydrodynamics

5.2.3 Message Aggregation

Message coalescing ensures that each data value is sent to a processor only once. In comparison, *message aggregation* ensures that only one message is sent to each processor, possibly at the expense of extra buffering. After message vectorization and coalescing, the Fortran D compiler locates and aggregates all RSDs representing data being sent to the same processor that have the same communication type. During code generation, these array sections are copied to a single buffer so that they may be sent as one message. The receiving processor then copies the buffered data back to the appropriate locations. Like message coalescing, message aggregation can be applied across loop nests. The compiler steps backwards in the program comparing RSDs, stopping when the RSD is aggregated, an intervening definition is found, or no candidate RSDs remain.

Consider the kernel in Figure 5.3. The lack of true dependences for nonlocal references to ZP, ZQ, and ZM allows their communication to be vectorized and coalesced outside the l loop. The compiler discovers they are being sent one processor to the right, and aggregates them in the same message. Nonlocal references to ZR and ZZ cause true dependences carried on the l loop. Their nonlocal accesses are vectorized and communications inserted inside loop l just after the loop header, allowing values from the previous iteration to be fetched at the beginning of each new iteration. Once RSDs are placed at the header, the compiler easily recognizes that the messages may be coalesced and aggregated as one message each for the left and right processors. Finally, nonlocal references to ZA cause loop-independent dependences. Communication is vectorized and coalesced at the level of the l loop, since it is the only loop common to the endpoints of the dependence. Messages are later inserted in front of the k loop enclosing accesses to ZA.

5.2.4 Collective Communication

Message vectorization using data dependence information can determine where communication should be inserted. However, the Fortran D compiler also needs to select an efficient communication mechanism. *Communication selection* is performed by comparing the subscript expression of each distributed dimension

```
{* Compiler Output *}
REAL ZP(0:25,100), ZQ(0:25,100), ZM(0:25,100)
REAL ZR(0:26,100), ZZ(0:26,100), ZA(0:25,100)
REAL ZU(25,100), ZV(25,100), ZB(25,100)
my$p = myproc() {* 0...3 *}
if (my$p .lt. 3) send(ZP(25,2:100),ZQ(25,2:100),ZM(25,2:100),my$p+1)
if (my$p .gt. 0) recv(ZP(0,2:100),ZQ(0,2:100),ZM(0,2:100),my$p-1)
do l = 1,time
  if (my$p .gt. 0) send(ZR(1,2:99),ZZ(1,2:99),my$p-1)
  if (my$p .lt. 3) send(ZR(25,2:99),ZZ(25,2:99),my$p+1)
  if (my$p .lt. 3) recv(ZR(26,2:99),ZZ(26,2:99),my$p+1)
  if (my$p .gt. 0) recv(ZR(0,2:99),ZZ(0,2:99),my$p-1)
  do k = 2,99
    do j = 1,25
      ZA(j,k) = F1(ZP(j-1,k),ZQ(j-1,k),ZM(j-1,k),ZR(j-1,k),...)
      ZB(j,k) = F2(ZP(j-1,k),ZQ(j-1,k),...)
    enddo
  enddo
  if (my$p .lt. 3) send(ZA(25,2:99),my$p+1)
  if (my$p .gt. 0) recv(ZA(0,2:99),my$p-1)
  do k = 2,99
    do j = 1,25
      ZU(j,k) = F3(ZZ(j-1,k),ZZ(j+1,k),ZA(j-1,k),...)
      ZV(j,k) = F4(ZR(j-1,k),ZR(j+1,k),ZA(j-1,k),...)
    enddo
  enddo
  do k = 2,99
    do j = 1,25
      ZR(j,k) = F5(ZR(j,k),ZU(j,k))
      ZZ(j,k) = F6(ZZ(j,k),ZV(j,k))
    enddo
  enddo
enddo
```

Figure 5.4 Livermore 18–Compiler Output

```
    {* Fortran D Program *}
    DECOMPOSITION D(N,N)
    ALIGN A, B with D
    DISTRIBUTE D(BLOCK,BLOCK)
    do j = 2,N
      do i = 2,N
S1      A(i,j) = B(i,j-1)+B(i-1,j)
S2      A(i,j) = B(c,j)+B(j,i)
S3      A(i,j) = B(f(i),j)
      enddo
    enddo
```

Figure 5.5 Communication Selection

in the *rhs* with the aligned dimension in the *lhs* reference. This determines the communication type of the RSD representing nonlocal accesses.

Consider the example program in Figure 5.5. Message vectorization determines that communication can be extracted from both the i and j loops. The arrays A and B are aligned identically and both dimensions are distributed, so we need to compare the first dimensions with each other, then the second. In S_1, the aligned dimensions of both the *lhs* and *rhs* references contain constant offsets to the same loop index variable. For these *shifts* resulting from *stencil computations*, individual calls to *send* and *recv* primitives are very efficient. This is the case for the Jacobi, SOR, and Red-black SOR examples previously discussed.

More complicated subscript expressions indicate the need for *collective communication* [140]. For example, the loop-invariant subscript for $B(c, j)$ in S_2 can be efficiently communicated using *broadcast*. Collective communication is selected because these communication patterns are not well-described by individual messages, and can be performed significantly faster using special purpose routines. The Fortran D compiler applies techniques pioneered by Li and Chen to recognize these patterns through syntactic analysis [140].

In other words, message vectorization, coalescing, and aggregation determine the extent to which communication for nonlocal accesses may be combined into a single message. For stencil computations these are point-to-point interprocessor communication and can be performed quite efficiently by individual calls to *send* and *recv* primitives. However, when communication takes place between groups of processors in regular patterns, message overhead can be reduced by utilizing fast *collective communication* routines instead of generating individual messages [26].

Examples of collective communication routines include broadcast, all-to-all, transpose, global concatenation, and global sum. Differing interdimensional alignments between *lhs* and *rhs* references point out the need for transpose, while symbolic analysis detects reductions that can use collective communication to accumulate partial results. Collective communication routines either rely on architectural support (*e.g.,* broadcast) or special protocols (*e.g.,* pair-wise exchange with combine) to reduce the number of messages that must be sent. They can significantly reduce the number of messages without affecting parallelism. For instance, computing the global sum of an array on N processors can be reduced from O(N^2) to O(NlogN) messages.

5.2.5 Run-time Processing

Irregular computations require special treatment in the Fortran D compiler. Consider the type of communication required to communicate the values needed by by $B(f(i), j)$ in S_3 of Figure 5.5. Because f is an irregular function (*e.g.,* an index array), the Fortran D compiler cannot precisely determine at compile-time what communication is required, even though message vectorization can extract the communication out of the i loop. Message coalescing and aggregation cannot be performed at compile-time because the actual array elements accessed are unknown. However, a combination of compile-time analysis and run-time processing can be applied to optimize communication.

As previously discussed, if no loop-carried true dependences are present, *inspectors* and *executors* may be created at compile-time during code generation to combine messages at run-time [123, 155]. The inspector performs the equivalent of message coalescing and aggregation at run-time. The executor then utilizes collective communication specialized for irregular computations. Special all-to-all *gather* and *scatter* routines collect all the nonlocal data with a small number of messages.

5.2.6 Relax Owner Computes Rule

The *owner computes* rule provides the basic strategy of the Fortran D compiler. We may also relax this rule, allowing processors to compute values for data they do not own. For instance, suppose multiple *rhs* of an assignment statement are owned by a processor that is not the owner of the *lhs*. Computing the result on the processor owning the *rhs* and then sending the result to the owner of the *lhs* could reduce the amount of data communicated. Consider the following loop:

```
do i = 1, n
   A(i) = B(i) + C(i)
enddo
```

Assume that $B(i)$ and $C(i)$ are mapped together to a processor different from the owner of $A(i)$. The amount of data communicated may be reduced by half if the computation is first performed by the processor owning B and C, then sent to the processor owning A. This optimization is a simple application of the "owner stores" rule proposed by Balasundaram [15].

In particular, it may be desirable for the Fortran D compiler to partition loops amongst processors so that each loop iteration is executed on a single processor, such as in Kali and Arf [127, 209]. This technique may improve communication and provides greater control over load balance, especially for irregular computations. It also eliminates the need for individual statement masks and simplifies handling of control flow within the loop body. The Fortran D compiler will detect phases of the computation where the owner computes rule may be relaxed to improve communications or load balance.

5.2.7 Replicate Computation

The Fortran D compiler considers scalar variables to be replicated. All processors thus perform computations involving assignments to scalar variables. This causes redundant computation to be performed, but is profitable because it significantly reduces communication costs. A similar approach may be taken for computations on elements of distributed arrays. It may be more efficient to replicate computation on multiple processors, rather than incur the expense of communicating the value from the owner of that element. Consider the following loop:

```
do i = 3, n
   X(i) = f(i)
   Y(i) = (X(i-1) + X(i-2)) / 2
enddo
```

Assume that arrays X and Y are distributed arrays aligned identically onto the same decomposition, and that f is a function performing computation that does not require values from other processors. Straightforward compilation of this loop would cause messages to be generated, communicating the new values of $X(i-1)$ and $X(i-2)$ to the processor performing the assignment to $Y(i)$. However, if the Fortran D compiler replicates the computation of $X(i-1)$ and $X(i-2)$ on the receiving processor, it eliminates the need for any communications.

5.3 Hiding Communication Overhead

The previous section discussed techniques to decrease communication costs by reducing T_{start}. This section presents optimizations to hide $T_{transit}$, the message transit time, by overlapping communication with computation. The same optimizations can also hide T_{copy}, the message copy time, by using *unbuffered messages*.

5.3.1 Message Pipelining

Message pipelining inserts a *send* for each nonlocal reference as soon as it is defined [178]. The *recv* is placed immediately before the value is used. Any computation performed between the definition and use of the value can then help hide $T_{transit}$. Unfortunately, message pipelining prevents optimizations such as message vectorization, resulting in significantly greater total communication cost. It is thus generally undesirable for completely parallel programs, but may be useful for exploiting parallelism for pipelined computations, as shown in the next chapter.

5.3.2 Vector Message Pipelining

We describe a new optimization, *vector message pipelining*, that hides $T_{transit}$ without increasing total communication cost. After message vectorization, pairs of vectorized *send* and *recv* statements have been gathered either inside or outside of loop headers. Vector message pipelining uses data dependence information to move vector *send* and *recv* statements towards their definitions and uses respectively in order to hide $T_{transit}$.

Vector message pipelining may be considered to be *macro-instruction scheduling*, where macro-instructions consist of vectorized *send*, *recv* statements and entire inner loop nests. Since *send* and *recv* statements interlock, they must be scheduled apart in order to avoid idle cycles. A simple application of vector message pipelining is to invoke all *send* statements before *recv* when a number of messages are sent at the same time.

Figure 5.6, Red-Black successive over-relaxation (SOR), demonstrates a more complex case. Values of red points computed by statement S_1 are used by S_3, corresponding to a loop-independent true dependence from S_1 to S_3. Similarly, S_2 computes red points used by S_4. Message vectorization creates communication for S_3 and S_4 at the level of the l loop, since it is the deepest loop enclosing these loop-independent dependences. We label these messages as `m$1` and `m$2`. Vector message pipelining then places the vector *send* for S_3 and S_4 after the j loops enclosing S_1 and S_2, respectively, using them to hide $T_{transit}$.

```
{* Fortran D Program *}
REAL V(1000,1000)
PARAMETER (n$proc = 10)
DECOMPOSITION D(1000,1000)
ALIGN V with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  {* compute red points *}
  do j = 3,999,2
    do i = 3,999,2
S1      V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
  do j = 2,998,2
    do i = 2,998,2
S2      V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
  {* compute black points *}
  do j = 2,998,2
    do i = 3,999,2
S3      V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
  do j = 3,999,2
    do i = 2,998,2
S4      V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
enddo
```

Figure 5.6 Red-Black SOR

In addition, values of black points computed by statement S_3 are used by S_1, corresponding to a loop-carried true dependence from S_3 to S_1. Similarly, S_4 computes black points used by S_2. Message vectorization creates communication for S_1 and S_2 at the level of the l loop, since it is the loop with the deepest loop-carried dependences. We label these messages as **m$3** and **m$4**. Vector message pipelining places the vector *recv* for S_2 just before the j loop enclosing S_2, using S_1 to hide $T_{transit}$.

Hiding transit time for the values needed by S_1 is more complicated, since the communication needs to cross iterations of the l loop. Scheduling *send* and *recv* statements across iterations of the outer time-step loop is analogous to *macro-software pipelining.* Vector message pipelining places the vector *send* for S_1 after S_3, using S_4 to hide $T_{transit}$. Matching vector *send* and *recv* statements must be inserted outside the loop. The final result is shown in Figure 5.7.

Vector Message Pipelining Algorithm

Vector message pipelining is implemented as follows in the Fortran D compiler. Message vectorization, coalescing, and aggregation produce RSDs of several types at different loop levels. The compiler applies vector message pipelining in a manner similar to that for inter-loop message coalescing and aggregation.

For RSDs representing loop-independent messages at loop L, the compiler inserts the *recv* before the loop in the normal manner and tries to move the *send* back as far as possible. The compiler steps backwards through the program starting at the loop header for L. When an assignment statement is encounted, the RSD or RSDs representing the nonlocal data being communicated are compared against the RSD representing the array section defined at that loop level. If the sections intersect, the *send* is inserted after the assignment statement to ensure the appropriate new values are communicated. The compiler is guaranteed to reach an intersecting statement at some point, since the array section must have been defined before L in order to produce the loop-independent dependence.

```
{* Compiler Output *}
REAL V(1000,0:101)
my$p = myproc() {* 0...9 *}
if (my$p .lt. 9) send(m$3,V(3:999:2,100),my$p+1)
do l = 1,time
  if (my$p .gt. 0) send(m$4,V(2:998:2,1),my$p-1)
  if (my$p .gt. 0) recv(m$3,V(3:999:2,0),my$p-1)
  do j = 1,99,2
    do i = 3,999,2
S1    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
  if (my$p .gt. 0) send(m$1,V(3:999:2,1),my$p-1)
  if (my$p .lt. 9) recv(m$4,V(2:998:2,101),my$p+1)
  do j = 2,100,2
    do i = 2,998,2
S2    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
  if (my$p .lt. 9) send(m$2,V(2:998:2,100),my$p+1)
  if (my$p .lt. 9) recv(m$1,V(3:999:2,101),my$p+1)
  do j = 2,100,2
    do i = 3,999,2
S3    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
  if (my$p .lt. 9) send(m$3,V(3:999:2,100),my$p+1)
  if (my$p .gt. 0) recv(m$2,V(2:998:2,0),my$p-1)
  do j = 1,99,2
    do i = 2,998,2
S4    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
    enddo
  enddo
enddo
if (my$p .gt. 0) recv(m$3,V(3:999:2,0),my$p-1)
```

Figure 5.7 Red-Black SOR—Compiler Output

Two options are possible for RSDs representing loop-carried messages. If the loop L is partitioned across processors, the *recv* and *send* serve as data synchronization and are inserted before and after L, respectively. Vector message pipelining is not applied since it interferes with the pipeline parallelism of the computation. A different class of parallelism optimizations are applied, described in detail later in Section 5.4.5.

However, if loop L is not partitioned, vector message pipelining is applicable. Normally, *send* and *recv* are both inserted at the top of the loop body to communicate data at the beginning of each iteration of L. The compiler leaves the *send* at the top of the loop body, then attempts to move the *recv* as far forward as possible within the loop body. The compiler steps forward through the body of L, comparing RSDs in the same manner as before for each assignment statement encountered. It is guaranteed to find an intersecting statement within the body of loop L, since the dependence for the message is carried by L.

5.3.3 Iteration Reordering

Red-Black SOR is a computation structured so that careful placement of vector *send* and *recv* statements using vector message pipelining can effectively hide communication costs. Where this is not the case, *iteration reordering* may be applied to change the order of program execution, subject to dependence constraints. This allows loop iterations accessing only local data to be separated and placed between *send* and *recv* statements to hide $T_{transit}$ [124].

```
{* Fortran D Program *}
REAL A(100,100), B(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  do j = 2,99
    do i = 2,99
      A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
    enddo
  enddo
  do j = 1,99
    do i = 2,99
      B(i,j) = A(i,j)
    enddo
  enddo
enddo
```

Figure 5.8 Jacobi

```
{* Compiler Output *}
REAL A(100:25), B(100,0:26)
my$p = myproc() {* 0...3 *}
do l = 1,time
  if (my$p .lt. 3) send(B(2:99,1),my$p-1)
  if (my$p .gt. 0) send(B(2:99,25),my$p+1)
  {* perform local iterations *}
  do j = 2,24
    do i = 2,99
      A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
    enddo
  enddo
  if (my$p .lt. 3) recv(B(2:99,26),my$p+1)
  if (my$p .gt. 0) recv(B(2:99,0),my$p-1)
  {* perform non-local iterations *}
  do j = 1,25,24
    do i = 2,99
      A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
    enddo
  enddo
  do j = 1,25
    do i = 2,99
      B(i,j) = A(i,j)
    enddo
  enddo
enddo
```

Figure 5.9 Jacobi—Compiler Output

We demonstrate how the Fortran D compiler finds local loop iterations for the Jacobi algorithm in Figure 5.8. First, communication analysis calculates [2:99,0] and [2:99,26] to represent nonlocal accesses to array B. These accesses are caused by the references B(i–1,j) and B(i+1,j). Applying their inverse subscript functions yields the iteration sets [1,2:99] and [25,2:99]. Subtracting these nonlocal iterations from the full iteration set yields [2:24,2:99] as the set of local loop iterations. These iterations are placed into a separate loop nest between the vector *send* and *recv* statements to hide $T_{transit}$. More aggressive iteration reordering would also extract iterations from the second j loop to be placed before the vector *recv* statement.

5.3.4 Unbuffered Messages

The Fortran D compiler generally uses *buffered messages*, such as the *csend*() and *crecv*() routines from the Intel NX/2 message-passing library on the Intel iPSC/860. Invoking a buffered *send* causes the calling process to block until the data has been copied out of the program address space into a system buffer. Upon return the process may overwrite the original data. This does not mean that the process must wait for the message to be actually received by another processor, just that the content of the message is no longer affected by the sending processor. Invoking a buffered *recv* causes the calling process to block until the data has been received and copied into the program address space. The advantage of buffered messages is that data communicated is guaranteed not to be overwritten after returning from the communication routine.

Unbuffered messages, such as the *isend*() and *irecv*() routines found in the Intel NX/2 library on the iPSC/860, permit computation and message copying to be performed in parallel on the same processor. A unbuffered *send* returns immediately, allowing computation to be performed on the sending processor concurrently with copying the data into a system buffer. If the data to be sent is contiguous, the copy may even be eliminated completely. A unbuffered *recv* posts a message destination, enabling computation to be performed on the receiving processor while the data is being received and copied. It also avoids an extra copy into a system buffer, since the message body may be placed directly at the posted address. To avoid inadvertent overwrites, an additional system call must be made for each unbuffered message to block the computation until the copy is complete.

Vector message pipelining and iteration reordering with buffered messages can only hide $T_{transit}$, since the processor must remain idle while copying the data. By using unbuffered messages, the Fortran D compiler also hides T_{copy}, the message copy time. This is important since copying is a major component of communication overhead for large messages. However, unbuffered messages should be utilized selectively. Message startup time for unbuffered messages is generally higher than for buffered messages, since the number of system calls is doubled. Unbuffered *sends* may also require multiple buffers for noncontiguous data. Note that in our model the only source of savings for a unbuffered *send* is the time to copy data to the system buffer. After the copy is performed, both buffered and unbuffered messages can overlap communication and computation.

5.4 Exploiting Parallelism

5.4.1 Partitioning Computation

Most scientific applications are completely parallel in either a *synchronous* or *loosely synchronous* manner [74]. In these computations all processors execute SPMD programs in a loose lockstep, alternating between phases of local computation and synchronous global communication. These problems achieve good load balance because all processors are utilized during the computation phase. For instance, Jacobi and Red-black SOR are loosely synchronous computations.

If a computation can be determined by the compiler to belong to this class of parallel programs, partitioning the computation using the "owner computes" rule yields a fully parallel program. To successfully exploit parallelism in these basic cases, the compiler must be able to intelligently partition the work at compile-time. The Fortran D compiler achieves this through loop bounds reduction and guard introduction.

Compile-time partitioning of parallel computations is key to any reasonable compilation system, and should not really be considered an optimization. *Cross-processor* dependences point out sequential components of the computation that cross processor boundaries. These dependences disable parallel execution by forcing processors to remain idle, waiting for their predecessors to finish computing. We show later in this section how optimizations may extract parallelism in the presence of cross-processor dependences.

5.4.2 Private Variables

Statements performing assignments to scalar and replicated array variables present a special challenge for the Fortran D compiler. Naive application of the *owner computes* rule would cause every processor to execute the assignment on all iterations. However, often the assignment can be partitioned because its value is used only in the current loop iteration. These cases are readily recognized, since the variable being assigned will have been labeled *private* during dependence analysis.

```
{* Fortran D Program *}
REAL ZA(100,100), ZB(100,100), ZR(100,100), QA
REAL ZU(100,100), ZV(100,100), ZZ(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN ZA, ZB, ZR, ZU, ZV, ZZ with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  do j = 2,99
    do k = 2,99
S1    QA = F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
S2    ZA(k,j) = F2(ZA(k,j),QA)
    enddo
  enddo
enddo
```

Figure 5.10 Livermore 23–Implicit Hydrodynamics

To partition statement S, the Fortran D compiler calculates the union of the iteration sets of all statements that use S. These statements can be identified by tracking all true dependence edges with S as its source. This union becomes the iteration set for S. The process is simplified if the iteration sets are calculated in reverse order for statements in each loop nest.

For instance, consider the loop in Figure 5.10. Because QA is a replicated scalar, the *owner computes* rule would assign all loop iterations as the iteration set for statement S_1. However, since the only use of QA occurs in the same loop iteration, it is classified as a private variable. The Fortran D compiler can thus assign S_1 the same iteration set as S_2, the only statement containing a true dependence with S_1 as its source.

5.4.3 Reductions and Scans

Some computations with cross-processor dependences may be parallelized directly. *Reductions* are associative and commutative operations that may be applied to a collection of data to return a single result. For instance, a sum reduction would compute and return the sum of all elements of an array. *Scans* are similar but perform parallel-prefix operations instead. A sum scan would return the sums of all the prefixes of an array. Scans are used to solve a number of computations in scientific codes, including linear recurrences and tridiagonal systems [51, 128].

The Fortran D compiler applies dependence analysis to recognize reductions and scans. If the reduction or scan accesses data in a manner that sequentializes computation across processors, the Fortran D compiler may parallelize it by relaxing the "owner computes" rule and providing methods to combine partial results. This requires changing the order in which computations are performed, which is why the operations must be both associative and commutative.

Reductions are parallelized by allowing each processor to compute in parallel, later accumulating the partial results. Communication using individual *send/recv* calls can be used to calculate the global result. Broadcast may be used in place of *send* for efficiency, and specialized collective communication routines such as *global-sum*() can reduce communication overhead even further for common reductions. Figure 5.11 shows how a sum reduction may be parallelized using a *global-sum* collective communication routine to combine the partial sums.

Scans may also be parallelized by reordering operations. Each processor first computes its local values in parallel, then communicates the partial results to all other processors. The global data is used to update local results. Though extra communication and computation is introduced during parallelization, the additional parallelism yields major performance improvements. Figure 5.12 demonstrates how a prefix sum may be computed, using a *global-concat* collective communication routine to collect the partial sums from each processor in S. The partial sums of all preceding processors are combined locally and used as a basis for computing local prefix sums [51].

```
{* Fortran D Program *}
REAL X(100), Z(100), Q
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Z with D
DISTRIBUTE D(BLOCK)
do l = 1,time
  Q = 0.0
  do k = 1,100
    Q = Q + Z(k)*X(k)
  enddo
enddo
```

```
{* Compiler Output *}
REAL X(25), Z(25), Q
do l = 1,time
  Q = 0.0
  do i = 1,25
    Q = Q + Z(k)*X(k)
  enddo
  Q = global-sum(Q)
enddo
```

Figure 5.11 Livermore 3–Inner Product

```
{* Fortran D Program *}
REAL X(100), Y(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Y with D
DISTRIBUTE D(BLOCK)
do l = 1,time
  X(1) = Y(1)
  do k = 2,100
    X(k) = X(k-1) + Y(k)
  enddo
enddo
```

```
{* Compiler Output *}
REAL X(25), Y(25), S(0:3)
my$p = myproc() {* 0...3 *}
do l = 1,time
  S(my$p) = 0.0
  do k = 1,25
    S(my$p) = S(my$p) + Y(k)
  enddo
  global-concat(S)
  X(1) = Y(1)
  if (my$p .ne. 0) then
    do k = 0,my$p-1
      X(1) = X(1) + S(k)
    enddo
  endif
  do k = 2,25
    X(k) = X(k-1) + Y(k)
  enddo
enddo
```

Figure 5.12 Livermore 11–First Sum

5.4.4 Dynamic Data Decomposition

Other computations contain parallelism, but are partitioned by the "owner computes" rule in a way that causes sequential execution. In these cases *dynamic data decomposition* may be used to temporarily change the ownership of data during program execution, exposing parallelism by *internalizing* cross-processor dependences [16].

For instance, consider the two substitution phases in the Alternating Direction Implicit (ADI) integration example in Figure 5.13. The computation wavefront only crosses one spatial dimension in each phase. A fixed column or row data distribution would result in one parallel and one sequential phase. By applying dynamic data decomposition using collective communication routines to change the array decomposition after each phase, the Fortran D compiler can internalize the computation wavefront in both phases, allowing processors to execute in parallel without communication [124].

However, dynamic data decomposition is only applicable when there are full dimensions of parallelism available in the computation. For instance, it cannot be used to exploit parallelism for SOR or Livermore 23 in Figure 5.10, because the computation wavefront crosses both spatial dimensions. Even when dynamic data decomposition is applicable, it may not be efficient, as shown in Section 6.3.

```
{* Fortran D Program *}
REAL A(100), B(100), X(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN X with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  {* Phase 1:  sweep along columns *}
  do j = 1, 100
    do i = 2, 100
      X(i,j) = F1(X(i,j),X(i-1,j),A(i),B(i))
    enddo
  enddo
  {* Phase 2:  sweep along rows *}
  do j = 2, 100
    do i = 1, 100
      X1(i,j) = F2(X(i,j),X(i,j-1),A(i),B(i))
    enddo
  enddo
enddo

{* Compiler Output *}
REAL A(100), B(100), X(25,100), X1(100,25)
EQUIVALENCE (X,X1)
do l = 1,time
  do j = 1, 25
    do i = 2, 100
      X(i,j) = F1(X(i,j),X(i-1,j),A(i),B(i))
    enddo
  enddo
  redistribute-row-to-col(X)
  do j = 2, 100
    do i = 1, 25
      X1(i,j) = F2(X(i,j),X(i,j-1),A(i),B(i))
    enddo
  enddo
  redistribute-col-to-row(X1)
enddo
```

Figure 5.13 ADI Integration

5.4.5 Pipelined Computations

This section describes how the Fortran D compiler exploits parallelism found in *pipelined computations*. The Fortran D compilation strategy presented so far is well-suited to compiling fully data-parallel programs, since it identifies and inserts efficient vector or collective communication at appropriate points in the program. However, a different class of computations contain loop-carried cross-processor data dependences that sequentialize computations over distributed array dimensions. Synchronization is required and processors are forced to remain idle at various points in the computation, resulting in poor load balance. We call these computations, such as SOR or ADI integration, pipelined computations.

To demonstrate the difference between parallel and pipelined computations, consider the difference in program execution illustrated in Figure 5.14. Solid lines denote computation, and dotted arrows represent communication from sender to recipient. For parallel computations, all processors can execute concurrently, communicating data when necessary. In pipelined computations, a processor cannot begin execution until it receives results computed by its predecessor.

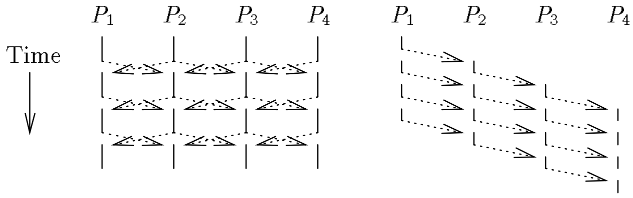


Figure 5.14 Parallel & Pipelined Computations

Data Decomposition, Cross-Processor Loops & Communication

```
ALIGN X(I,J) with A(I,J)
DISTRIBUTE A(BLOCK,:)
```

Original Order

```
{recv row from P_left}
do* i = 2,N/P
  do j = 1,N
    X(i,j) = X(i-1,j)
  enddo
enddo
{send row to P_right}
```

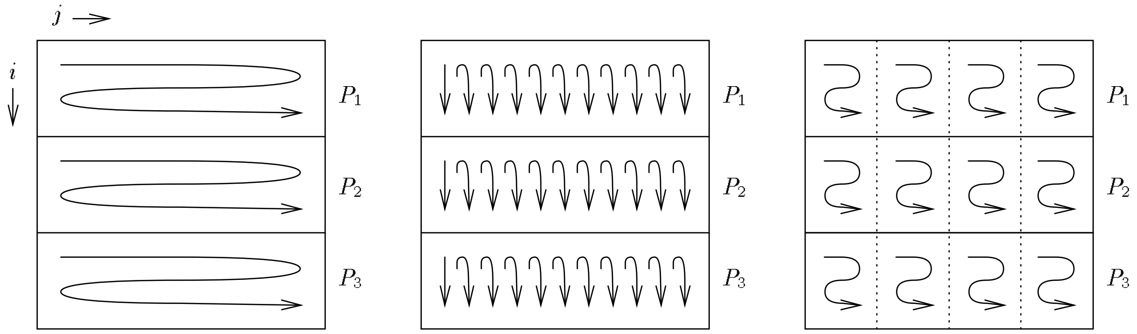
Fine-grain Pipelining

```
do j = 1,N
  {recv element from P_left}
  do* i = 2,N/P
    X(i,j) = X(i-1,j)
  enddo
  {send element to P_right}
enddo
```

Coarse-grain Pipelining

```
do jj = 1,N,Bk
  {recv block from P_left}
  do* i = 2,N/P
    do j = jj,jj+Bk-1
      X(i,j) = X(i-1,j)
    enddo
  enddo
  {send block to P_right}
 enddo
```

Data Space & Traversal Order



Processor Trace

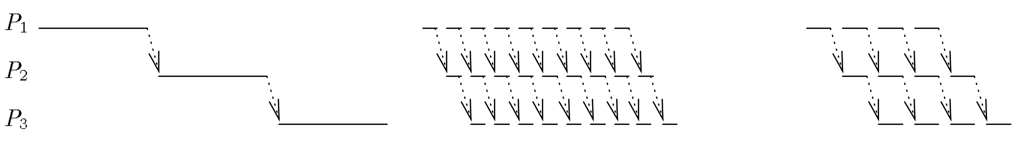


Figure 5.15 Examples of Pipelined Computations

INPUT:
Loop nest $\{L_1, \ldots, L_n\}$ with index variables $\{I_1, \ldots, I_n\}$
List of all loop-carried true dependences
Data decomposition of all distributed arrays in loop nest

OUTPUT:
$Loops$ = Set of cross-processor loops

ALGORITHM:
```
Loops ← ∅
for each loop-carried true dependence between
      references A(f_1, ..., f_m) and A(g_1, ..., g_m) do
   for each distributed dimension k of A do
      {* f_k and g_k are subscripts in dimension k *}
      if f_k ≠ g_k or f_k is not of form αI_j + β then
         for each index variable I_j in either f_k or g_k do
            if L_j encloses both references to A then
               Loops ← Loops ∪ {L_j}
            endif
         endfor
      endif
   endfor
endfor
```

Figure 5.16 Finding Cross-Processor Loops

Exploiting Pipeline Parallelism

Pipelined computations present opportunities for the compiler to exploit partial parallelism through pipelining, enabling processors to overlap computation with one another. By sending partial results to their successors earlier, processors may overlap computation with one another to achieve pipeline parallelism. When used in this fashion, messages both transmit data and serve as data synchronization. The degree of pipeline parallelism depends on how soon each processor is able to begin work after its predecessor starts.

One method of exploiting parallelism in pipelined computations through *message pipelining*—sending a message when its value is first computed, rather than waiting until its value is needed [178]. Rogers and Pingali applied this optimization to a Gauss-Seidel computation (a special case of SOR) that is distributed cyclically. However, more sophisticated approaches are usually required.

Figure 5.15 illustrates the tradeoffs between communication and parallelism that must be considered when optimizing pipelined computations. It presents the program text, data space traversal order, and a processor trace for three versions of the computation. In the processor trace, elapsed time proceeds from left to right. The computation for each processor is represented as a solid line, and messages are shown as dashed lines between processors.

In the original program execution order, message vectorization minimizes communication overhead but sequentializes the computation. Applying message pipelining alone is insufficient, since only the computation for the last row will be pipelined. The key observation is that for some pipelined computations, the program execution order must also be changed. The following sections presents compiler techniques for recognizing and optimizing such computations.

Cross-Processor Loops

The Fortran D compiler identifies pipelined computations using *cross-processor* loops. We classify loops in numeric computations as either time-bound or space-bound. *Time-bound* loops correspond to time steps in the computation, with each iteration accessing much or all of the data space. They are usually outermost loops that need to be executed sequentially. In comparison, *space-bound* loops iterate over the data

Data Decompositions, Loop Nests, & Cross-Processor Loops

Loop 1

```
DECOMPOSITION A(N,N)
REAL X(N,N)
ALIGN X(I,J) with A(I,J)
DISTRIBUTE A(BLOCK,:)

 do* i = 2,N
   do j = 1,N
    X(i,j) = X(i-1,j)
   enddo
 enddo
```

Loop 2

```
DECOMPOSITION A(N,N)
REAL X(N,N)
ALIGN X(I,J) with A(I,J)
DISTRIBUTE A(:,BLOCK)

 do i = 2,N
   do j = 1,N
     X(i,j) = X(i-1,j)
   enddo
 enddo
```

Loop 3

```
DECOMPOSITION A(N,N)
REAL X(N,N)
ALIGN X(I,J) with A(I,J)
DISTRIBUTE A(BLOCK,BLOCK)

 do* i = 2,N
  do* j = 2,N
   X(i,j) =
       X(i-1,j) + X(i,j-1)
  enddo
 enddo
```

Data Space & Cross-Processor Dependences

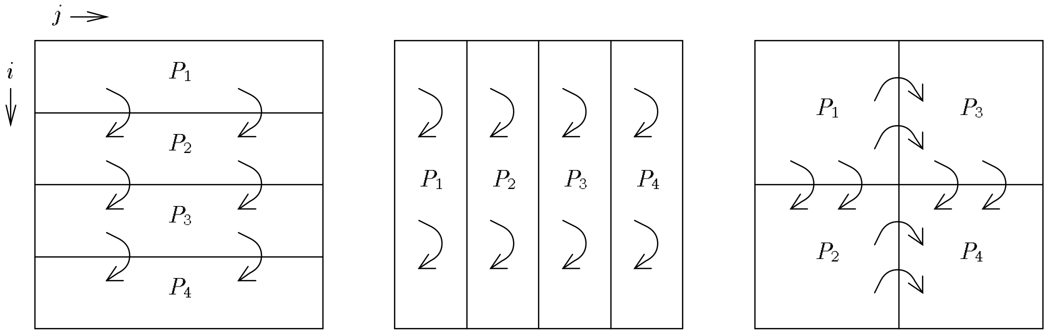


Figure 5.17 Examples of Cross-Processor Dependences and Loops

space, with each iteration accessing part of each array. These loops are usually parallel in data-parallel computations, but may be sequential if they cause a computation *wavefront* to sweep across the data space.

The Fortran D compiler labels a loop as *cross-processor* if it is a sequential space-bound loop causing computation wavefronts that cross processor boundaries (*i.e.,* sweeps over the distributed dimensions of the data space). The compiler finds cross-processor loops as follows. First, it considers all pairs of array references that cause loop-carried true dependences. If non-identical subscript expressions occur in a distributed dimension of the array, index variables appearing in the subscript expressions belong to cross-processor loops. The algorithm is shown in Figure 5.16. In most cases, cross-processor loops are loops carrying true dependences whose iterations have been partitioned across processors.

Consider the loop-carried true dependence between S_1 and S_2 caused by ZA(k,j) and ZA(k,j-1) in Figure 5.10. Since the second dimension of ZA is distributed, the compiler compares j and $j-1$, the subscripts in the second dimension. These are not identical, so the j loop is labeled as cross-processor. No other loops are cross-processor.

Figure 5.17 illustrates cross-processor dependences and loops. We denote cross-processor loops as `do*`. All loops in the example are space-bound loops that sweep the data space. In Loop 1, the i loop is cross-processor because the computation wavefront sweeps the i dimension across processors. There are no cross-processor loops in Loop 2 because the computation wavefront is internalized and does not cross processor boundaries. In Loop 3 both the i and j loops are cross-processor because the computation wavefront sweeps across processors in both dimensions.

These examples make it clear how cross-processor loops may be used to classify computations. The presence of any cross-processor loop in a loop nest indicates that it is a pipelined computation. Computations such as Loop 2 that do not possess cross-processor loops are loosely synchronous, since all processors may execute in parallel. Computations such as Loops 1 & 3 that do possess cross-processor loops are pipelined, since processors must wait in turn for computation to be completed.

5.4.6 Loop Transformations

Optimizing pipelined computations may require applying loop transformations. Here we present three transformations used to change the program execution order for pipelined computations. *Loop interchange* swaps adjacent loop headers to alter the traversal order of the iteration space. It may be applied only if the source and sink of each dependence are not reversed in the resulting program. This may be determined by examining the distance or direction vector associated with each dependence [10, 205].

Strip-mining increases the step size of an existing loop and adds an additional inner loop. The legality of applying strip-mine followed by loop interchange is determined in the same manner as for *unroll-and-jam* [117]. The Fortran D compiler may apply strip-mining in order to reduce storage requirements for computations. It may also be used with loop interchange to help exploit pipeline parallelism, as discussed in the next section.

Loop fusion combines multiple loops with identical headers into a single loop. It is legal if the direction of existing dependences are not reversed after fusion [132, 205]. Loop fusion can improve data locality, but its main use in the Fortran D compiler is to fuse imperfectly nested loops in order to enable loop interchange and strip-mine.

5.4.7 Fine-grain Pipelining

We present two optimizations to exploit pipeline parallelism. The key observation we make is that the granularity of pipeline parallelism is determined by the amount of computation enclosed by the *send* and *recv* primitives inserted around cross-processor loops. Unfortunately, this is also directly responsible for the amount of communication overhead. The Fortran D compiler thus needs to balance increases in pipeline parallelism with increases in communication overhead, choosing the degree of pipelining that results in the most efficient execution.

Fine-grain pipelining is a compiler technique where loop interchange is used to minimize the granularity of pipelining, maximizing both pipeline parallelism and communication overhead. To perform fine-grain pipelining, the Fortran D compiler interchanges all cross-processor loops as deeply as possible, so that they enclose the least amount of computation. The resulting program execution order generates values needed by other processors as quickly as possible.

Because loop-carried true dependences are carried by the cross-processor loop, the standard message vectorization algorithm will place calls to *recv* and *send* primitives before and after the cross-processor loop, respectively, during code generation. This produces the maximum parallelism with the finest granularity of pipelining. The resulting program is a major improvement over sequentialized computation, but incurs the most communication overhead since a message is sent for every iteration accessing nonlocal data.

5.4.8 Coarse-grain Pipelining

As we have shown, the granularity of pipelining is determined by the amount of computation $\mathcal{C}$ enclosed by cross-processor loops. Increasing $\mathcal{C}$ reduces communication, since all nonlocal data accessed by $\mathcal{C}$ may be communicated in a single message. However, parallelism is also reduced since processors must wait longer before beginning to compute. *Coarse-grain pipelining* attempts to balance parallelism with communication overhead, using a combination of loop interchange and strip-mining where needed to adjust the granularity of pipelining.

A simple heuristic for coarse-grain pipelining is implemented in the prototype Fortran D compiler. It first interchanges all cross-processor loops inward as deeply as possible, as in fine-grain pipelining. It then strip-mines the deepest loop enclosing the cross-processor loops. Communication is inserted outside of the newly strip-mined loop. The granularity of pipelining is determined by the strip size. Later in this chapter we discuss how to choose an efficient granularity for pipelining based on the ratio between computation and communication costs.

Figure 5.18 shows examples of both fine and coarse-grained pipelining applied to the implicit hydrodynamics kernel in Figure 5.10. Fine-grain pipelining interchanges the cross-processor loop j to the innermost position to maximize pipelining. In comparison, coarse-grain pipelining strip-mines the k loop by a factor B, then interchanges the iterator loop kk outside the j loop. This allows communication for B iterations to be vectorized at the j loop. The legality of loop interchange and strip-mine is determined exactly as

```
{* Compiler Output 1:  Fine-grain pipelining *}
REAL ZA(100,0:26), ZB(100,25), ZR(100,25), QA
REAL ZU(100,25), ZV(100,25), ZZ(100,25)
my$p = myproc() {* 0...3 *}
do l = 1,time
  if (my$p .gt. 0) send(ZA(2:99,1),my$p-1)
  if (my$p .lt. 3) recv(ZA(2:99,26),my$p+1)
  do k = 2,99
    if (my$p .gt. 0) recv(ZA(k,0),my$p-1)
    do j = 1,25
      QA = F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
      ZA(k,j) = F2(ZA(k,j),QA)
    enddo
    if (my$p .lt. 3) send(ZA(k,25),my$p+1)
  enddo
enddo
```

```
{* Compiler Output 2:  Coarse-grain pipelining *}
REAL ZA(100,0:26), ZB(100,25), ZR(100,25), QA
REAL ZU(100,25), ZV(100,25), ZZ(100,25)
my$p = myproc() {* 0...3 *}
do l = 1,time
  if (my$p .gt. 0) send(ZA(2:99,1),my$p-1)
  if (my$p .lt. 3) recv(ZA(2:99,26),my$p+1)
  do kk = 2,99,B
    if (my$p .gt. 0) recv(ZA(kk:kk+B-1,0),my$p-1)
    do j = 1,25
      do k = kk,kk+B
        QA = F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
        ZA(k,j) = F2(ZA(k,j),QA)
      enddo
    enddo
    if (my$p .lt. 3) send(ZA(kk:kk+B-1,25),my$p+1)
  enddo
enddo
```

Figure 5.18 Livermore 23–Compiler Output

for shared-memory programs [10, 117, 132]. Because pipelining disturbs the original computation order, the node compiler later permutes inner loops in *memory order*, to ensure data locality for the local computation [116].

5.5 Improve Partitioning

One of the responsibilities of the Fortran D compiler is to partition computation across processors. As we have seen, the compiler instantiates this partition by reducing loop bounds or introducing guards in the program. Guards are typically inefficient, since they must be evaluated frequently at run-time. This section describes optimizations to reduce or eliminate the need to rely on guards, reducing the cost of partitioning computation.

5.5.1 Loop Bounds Reduction

The primary and most effective means of eliminating guards is through loop bounds reduction. The method for doing so has already been described in previous chapters. More interestingly, additional program transformations may be needed to enable loop bounds reduction.

5.5.2 Loop Distribution

Loop distribution separates independent statements inside a single loop into multiple loops with identical headers. Loop distribution may be applied only if the statements are not involved in a recurrence and the direction of existing loop-carried dependences are not reversed in the resulting loops [117, 132]. It can separate statements in loop nests with different local iteration sets, avoiding the need to evaluate guards at run-time. Loop distribution may also separate the source and sink of loop-carried or loop-independent cross-processor dependences, allowing individual messages to be combined into a single vector message.

When compiling for distributed-memory machines, loop distribution can be used to simplify guard introduction and enable other transformations such as loop interchange. Recall that when all statements in a loop nest possess identical iteration sets, the Fortran D compiler can instantiate the computation partition very efficiently by reducing loop bounds. Otherwise the compiler must insert explicit guards for each group of statements possessing different local iteration sets. This is less efficient since each guard must be evaluated at run-time, once per iteration of the loop. To avoid this situation, the current Fortran D compiler applies loop distribution where safe when statements in a loop nest have differing iteration sets.

For instance, consider the simplified program fragment from an ADI integration code shown in Figure 5.19. It initializes boundary conditions for a coefficient array that is mapped as a torus. I.e., edges of the array wrap around to join at the opposite edge. Statements S_1 and S_2 initialize the first two rows of Y, while statements S_3 and S_4 initialize its last two rows. Because Y is distributed row-wise in one dimension, S_1 and S_2 are executed on all iterations of the j loop by the first processor, while S_3 and S_4 are executed on all iterations by the last processor.

Without loop distribution, the compiler is forced to insert explicit guards for S_1, S_2, S_3 and S_4, as shown in the first part of Figure 5.20. However, once data dependences and iteration sets are calculated, it is quite easy for the compiler to determine that it is both legal and desirable to distribute the i and j loops so that the two statement groups are separated. The compiler can then generate the efficient code shown in Figure 5.20. A similar process occurs for the shallow water simulation code shown in Figures 5.21 and 5.22.

Loop distribution may also improve parallelism by converting a loop-carried cross-processor dependence to a loop-independent dependence. For instance, distributing the i loop in Figure 5.23 converts a sequential loop into two parallel loops.

5.5.3 Loop Alignment

Loop alignment moves instances of a statement from one loop iteration to another. To align a statement S by a distance k, the first k iterations of the loop are *peeled* off into a separate header. The statement S is then aligned, moving the k copies of S in before the loop after the loop instead and adjusting the instance of S in the loop body. Figure 5.24 presents an example where the statement assigning to Y is aligned by one iteration of the i loop. Loop alignment is used in conjunction with replication by shared-memory parallelizing compilers to break loop-carried dependences [9, 33].

In the context of distributed-memory compilation, loop alignment may be used to change the iteration set of a given statement. This can be used to improve guard introduction by adjusting statements in a loop nest so that they possess the same iteration sets. For instance, after loop alignment both statements in Figure 5.24 have the same iteration set. The Fortran D compiler can apply loop bounds reduction to partition the computation. Guards must still be introduced for the statements extracted from the loop nest, but these guards only need to be evaluated once for the entire loop nest, rather than once per iteration.

Loop alignment is more flexible than loop distribution in improving guard introduction for certain cases, but is less effective in other cases. For instance, Figure 5.25 shows the code produced if loop alignment is used to adjust the iteration set of the statement assigning to Y. Since the loop aligned is an inner loop, imperfectly nested statements are produced whose guards must be evaluated on each outer loop iteration. Loop distribution is able to generate cleaner code. Even when alignment can be applied cleanly, the code generated is in general more complex than that produced by loop distribution. For this reason the Fortran D compiler prototype relies on loop distribution instead of loop alignment. Empirical studies are necessary to determine whether there exist cases where loop alignment is superior.

```
   {* Fortran D Program *}
   REAL X(256,256), Y(256,256)
   PARAMETER (n$proc = 4)
   DECOMPOSITION D(256,256)
   ALIGN X, Y with D
   DISTRIBUTE D(BLOCK,:)
   {* Y(i,j) = F(X(i+1,j),X(i-1,j),X(i+2,j),X(i-2,j)) *}
   do j = 1,256
S1   Y(1,j) = F(X(2,j),X(256,j),X(3,j),X(255,j))
S2   Y(2,j) = F(X(3,j),X(1,j),X(4,j),X(256,j))
S3   Y(255,j) = F(X(256,j),X(254,j),u(1,j),X(253,j))
S4   Y(256,j) = F(X(1,j),X(255,j),X(2,j),X(254,j))
   enddo
```

Figure 5.19 Circular Boundary Conditions

```
{* Compiler Output 1:  Guard Introduction *}
my$p = myproc() {* 0...3 *}
do j = 1,256
  if (my$p .eq. 0) then
    Y(1,j) = F(X(2,j),X(256,j),X(3,j),X(255,j))
    Y(2,j) = F(X(3,j),X(1,j),X(4,j),X(256,j))
  endif
  if (my$p .eq. 3) then
    Y(255,j) = F(X(256,j),X(254,j),u(1,j),X(253,j))
    Y(256,j) = F(X(1,j),X(255,j),X(2,j),X(254,j))
  endif
enddo

{* Compiler Output 2:  Loop Distribution *}
my$p = myproc() {* 0...3 *}
if (my$p .eq. 0) then
  do j = 1,256
    Y(1,j) = F(X(2,j),X(256,j),X(3,j),X(255,j))
    Y(2,j) = F(X(3,j),X(1,j),X(4,j),X(256,j))
  enddo
endif
if (my$p .eq. 3) then
  do j = 1,256
    Y(255,j) = F(X(256,j),X(254,j),u(1,j),X(253,j))
    Y(256,j) = F(X(1,j),X(255,j),X(2,j),X(254,j))
  enddo
endif
```

Figure 5.20 Circular Boundary Conditions—Compiler Output

```
{* Fortran D Program *}
REAL X(256,256), Y(256,256), Z(256,256)
PARAMETER (n$proc = 4)
DECOMPOSITION D(256,256)
ALIGN X, Y, Z with D
DISTRIBUTE D(BLOCK,:)
do j = 1,255
  do i = 1,255
    X(i+1,j) = F1(Z(i+1,j+1),Z(i+1,j))
    Y(i,j+1) = F2(Z(i+1,j+1),Z(i,j+1))
  enddo
enddo
```

Figure 5.21 Shallow—Weather Prediction

```
{* Compiler Output 1:  Guard Introduction *}
my$p = myproc() {* 0...3 *}
{* if (my$p .eq. 0) lb1 = 1 else lb1 = 0 *}
{* if (my$p .eq. 3) ub1 = 24 else ub1 = 25 *}
lb1 = max(my$p*64,1)-(my$p*64)
ub1 = min((my$p+1)*64,255)-(my$p*64)
do j = 1,255
  do i = lb1,ub1
    if (i .lt. 64) X(i+1,j) = F1(Z(i+1,j+1),Z(i+1,j))
    if (i .gt. 0) Y(i,j+1) = F2(Z(i+1,j+1),Z(i,j+1))
  enddo
enddo
{* Compiler Output 2:  Loop Distribution *}
my$p = myproc() {* 0...3 *}
lb1 = max(my$p*64,1)-(my$p*64)
ub1 = min((my$p+1)*64,255)-(my$p*64)
do j = 1,255
  do i = lb1,63
    X(i+1,j) = F1(Z(i+1,j+1),Z(i+1,j))
  enddo
enddo
do j = 1,255
  do i = 1,ub1
    Y(i,j+1) = F2(Z(i+1,j+1),Z(i,j+1))
  enddo
enddo
```

Figure 5.22 Shallow—Compiler Output

```
{* Fortran D Program *}
REAL X(100),Y(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Y with D
DISTRIBUTE D(BLOCK)
do i = 6,100
  X(i) = ...
  Y(i) = F(X(i-5))
enddo
{* Compiler Output 1:  Sequentialized *}
REAL X(-4:25),Y(25)
my$p = myproc() {* 0...3 *}
lb1 = max(my$p*25,6)-(my$p*25)
if (my$p .gt. 0) recv(X(-4:0),my$p-1)
do i = lb1,25
  X(i) = ...
  Y(i) = F(X(i-5))
enddo
if (my$p .lt. 3) send(X(21:25),my$p-1)
{* Compiler Output 2:  Parallel *}
REAL X(-4:25),Y(25)
my$p = myproc() {* 0...3 *}
lb1 = max(my$p*25,6)-(my$p*25)
do i = lb1,25
  X(i) = ...
enddo
if (my$p .lt. 3) send(X(21:25),my$p-1)
if (my$p .gt. 0) recv(X(-4:0),my$p-1)
do i = lb1,25
  Y(i) = F(X(i-5))
enddo
```

Figure 5.23 Loop Distribution—Improving Parallelism

```
{* Program *}
do i = 1,99
  X(i) = F1(i)
  Y(i+1) = F2(i)
enddo
```

```
{* Loop Alignment *}
X(1) = F1(1)
  do i = 2,99
  X(i) = F1(i)
  Y(i) = F2(i-1)
enddo
Y(100) = F1(99)
```

Figure 5.24 Loop Alignment Example

```
{* Compiler Output 3:  Loop Alignment *}
my$p = myproc() {* 0...3 *}
lb1 = max((my$p*64)+1,2)-(my$p*64)
ub1 = min((my$p+1)*64,255)-(my$p*64)
do j = 1,255
  if (my$p .eq. 0) Y(1,j+1) = F2(Z(2,j+1),Z(1,j+1))
  do i = lb1,ub1
    X(i,j) = F1(Z(i,j+1),Z(i,j))
    Y(i,j+1) = F2(Z(i+1,j+1),Z(i,j+1))
  enddo
  if (my$p .eq. 3) X(64,j) = F1(Z(64,j+1),Z(64,j))
enddo
```

Figure 5.25 Shallow—Loop Alignment

5.6 Reducing Storage

5.6.1 Partitioning Data

Most optimizations increase the amount of temporary storage required by the program. *Storage optimizations* seek to reduce storage requirements. Compile-time partitioning of the data so that each processor allocates memory only for array sections owned locally is fundamental. Otherwise the problem size is limited by the amount of data that can be placed on a single processor. We view *partitioning data* as fundamental for any reasonable compiler; like partitioning computation, it should not be merely viewed as an optimization.

5.6.2 Message Blocking

If insufficient storage is available, buffers must be used to store nonlocal data. *Message blocking* may then be applied to reduce the buffer storage needed. A loop carrying communication is strip-mined by a block factor B. Each vectorized message of size n is then divided into n/B messages of size B and sent inside the strip-mined loop. This reduces the buffer space required by a factor of n/B at the expense of additional messages.

5.7 Discussion

Techniques are presented to improve performance for both fully parallel and pipelined computations. Most messages may be eliminated by using data dependence analysis to combine messages for nonlocal accesses caused by the same variable reference. Some additional messages may be reduced by combining messages for different variable references or arrays. Message placement and type allow communication cost to be hidden by overlapping communication with computation. Parallelism optimizations recognize and exploit parallel and pipelined computations.

Chapter 6

Evaluation of Compiler Optimizations

Communication and parallelism optimizations are evaluated on the Intel iPSC/860, a MIMD distributed-memory machine. Profitability formulas are derived for each optimization. Results for stencil computations show that exploiting parallelism for pipelined computations, reductions, and scans is vital. Message vectorization, collective communication, and efficient coarse-grain pipelining also significantly affect performance. Scalability of communication and parallelism optimizations are analyzed. The effect of communication optimizations depends on the proportion of nonlocal data to local data, while parallelism optimizations depend on the total problem size and number of processors. An optimization strategy is developed based on these analyses.

6.1 Introduction

This chapter empirically evaluates the effectiveness of Fortran D compiler optimizations on a representative MIMD distributed-memory machine. Formulas are derived that allow the compiler to estimate the profitability of each optimization. The empirical and analytical results are used to evaluate the scalability of various optimizations with respect to problem and machine size. An optimization strategy is derived accordingly.

6.2 Empirical Performance Evaluation

To evaluate the usefulness of each compiler optimization, we applied them where appropriate to the Livermore and PDE kernels used as examples in the previous chapters. Table 6.1 shows the optimized versions of each program. Message vectorization, coalescing, aggregation, and fine-grain pipelining were applied by the prototype Fortran D compiler; other optimizations were performed by hand, simulating algorithms we expect to implement in the mature compiler. Nc is a parallel version of the program with all communication removed. It is meant to provide a baseline for measuring communication overhead. We also use $nc \times P$ to estimate the sequential execution time, since most problem sizes are too large for a single processor. Parallel speedup is then simply $\frac{nc \times P}{time}$.

The experiments were performed on a 32 node Intel iPSC/860 with 8 Meg of memory per node. Each program was compiled under -O4 using Release 2.0 of *if77*, the iPSC/860 compiler. All arrays are double precision and distributed block-wise in one dimension. Timings were taken using *dclock*() for one iteration of l, the time step loop. Results presented are both tabulated and plotted graphically.

In Tables 6.2, 6.3, and 6.4, results are presented in milliseconds for several machine and problem sizes. P indicates the number of processors. N describes the total problem size and its dimensionality; N/P yields the problem size on an individual processor. In addition to the timings, each table contains ratios of execution times for some selected optimizations, illustrating their relative usefulness.

Figures 6.1, 6.2, 6.3 present the same timings graphically. In each figure, program execution times in seconds are plotted logarithmically along the Y-axis. Optimizations are plotted along the X-axis, ranging from sequential execution (no optimization) to ideal parallel execution (no communication). Dotted, dashed, and solid lines represent execution times for 8, 16, and 32 processors, respectively. Lines are marked with $\circ$, $\bullet$, $\star$ and other symbols to represent the problem size.

Version	Optimizations Performed
nc	no communication
mp	message pipelining (element messages)
mv	message vectorization
mc	*mv* + message coalescing
ma	*mc* + message aggregation
vmp	*ma* + vectorized message pipelining
ir	*vmp* + iteration reordering
mc', vmp', ir'	versions w/ unbuffered messages
seq	sequential reduction/scan
sr	accumulate using send/receive
br	accumulate using broadcast/receive
cc	accumulate using collective communication
dyn	dynamic data decomposition
fgp	fine-grain pipelining
cgp	coarse-grain pipelining w/ block size B

Table 6.1 Optimized Versions of Test Kernels

6.2.1 Optimizations for Communication Overhead

We begin by measuring the effect of optimizations to reduce and hide communication overhead. We found that the nature of the computation and data partition significantly affects the utility of each optimization. For instance, we omitted execution times for Livermore 7, a 1D stencil computation, since data movement is limited and optimizations have little effect for reasonable problem sizes. The three kernels for which we present results are 2D stencil computations with 1D data distributions. Enough communication is required to make optimizations significant.

Table 6.2 presents the performance of communication optimizations for parallel stencil kernels. The timings are also depicted in Figure 6.1. For parallel computations, message vectorization is clearly the most important optimization. The numbers computed for $\frac{mp}{mv}$ (2.1–8.9) demonstrate that message vectorization significantly improves execution compared to sending element messages. Message aggregation provides a small fixed gain. Vector message pipelining and iteration reordering help, but are most effective when used in tandem with unbuffered messages. Unbuffered messages alone are insufficient, since the original program may not provide enough computation to hide all copying costs. Optimizations to hide communication lose effectiveness for small problem sizes, since insufficient computation exists to hide all message copy and transit overhead. Optimizations should not be applied in all cases. For instance, iteration reordering actually degraded performance for Livermore 18.

To evaluate the profitability of optimizations beyond message vectorization, we compute $\frac{mv}{best}$, where *best* is defined as the best time among all optimizations. The results show that other optimizations can improve somewhat on message vectorization (1.1–2.6), but the differences are less dramatic and drop quickly with increasing problem size. From $\frac{best}{nc}$ we see that optimizations can reduce communication overhead to a small percentage of total computation cost as problem size increases (5.3 to 1.01). This translates into close to linear speedup for larger problem sizes, as shown by the speedup values calculated for $\frac{nc \times P}{best}$.

These timings lead us to conclude that for parallel computations, communication optimizations can significantly reduce communication overhead, depending on the amount and nature of computation performed by each processor. The number of processors appears to have little effect, except indirectly by changing the amount of computation per processor. For larger problem sizes, message vectorization seems to yield most of the available improvement.

6.2.2 Optimizations for Reductions and Scans

Table 6.3 illustrates the performance of optimizations for parallelizing reductions and scans. In *seq*, the computation is sequentialized by requiring each processor to wait for the partial result from the previous

Kernel	P	N	nc	mp	mv	mc	ma	vmp	ir	mc'	vmp'	ir'	$\frac{mp}{mv}$	$\frac{mv}{best}$	$\frac{best}{nc}$	$\frac{nc \times P}{best}$
Liv 18	16	32x32	1.2	62.1	9.7	7.4	5.7	5.7	5.7	5.3	4.1	4.3	6.40	2.37	3.58	4.7
		64x64	3.9	62.7	16.2	12.7	11.4	11.0	11.6	11.1	8.7	9.8	3.87	1.86	2.23	7.2
		128x128	14.4	137	35.5	29.7	27.5	25.6	28.0	26.5	22.5	24.9	3.86	1.58	1.57	10.2
		256x256	54.3	317	90.2	80.9	77.6	77.8	79.3	76.6	69.7	73.6	3.51	1.29	1.30	12.5
		512x512	211	732	277	260	257	255	262	253	239	248	2.64	1.15	1.13	14.1
	32	32x32	0.7	62.5	9.5	7.1	5.5	5.4	5.5	5.0	3.7	4.3	6.58	2.57	5.29	6.1
		64x64	2.5	60.5	14.7	11.1	9.9	9.9	9.9	9.9	7.3	7.5	4.12	2.01	2.92	11.0
		128x128	8.0	128	29.3	23.6	21.3	20.7	21.7	20.6	16.5	18.3	4.37	1.76	1.50	15.5
		256x256	28.8	287	64.8	55.4	53.8	51.6	53.9	52.8	43.7	48.1	4.43	1.48	1.52	21.1
		512x512	109	630	177	159	156	155	160	152	137	146	3.56	1.29	1.25	25.5
Jacobi	16	128x128	1.3	29.3	4.1	-	-	3.9	3.7	3.8	3.6	2.9	7.15	1.41	2.23	7.2
		256x256	5.3	64.0	9.8	-	-	8.9	9.2	9.0	8.7	6.2	6.53	1.58	1.16	13.7
		512x512	20.7	145	28.3	-	-	29.0	27.6	27.4	27.3	23.2	5.18	1.22	1.10	14.3
		1Kx1K	96.9	349	110	-	-	107	112	109	109	99.1	3.17	1.11	1.02	15.6
		2Kx2K	385	889	412	-	-	417	411	410	410	391	2.16	1.05	1.02	15.8
	32	128x128	0.7	29.9	3.5	-	-	3.1	3.1	3.4	3.2	3.0	8.54	1.17	4.29	7.5
		256x256	2.7	64.2	7.2	-	-	7.2	6.6	7.2	7.1	4.9	8.92	1.47	1.81	17.6
		512x512	10.4	136	18.2	-	-	18.7	18.2	18.7	18.5	12.3	7.56	1.48	1.20	27.1
		1Kx1K	48.5	296	64.1	-	-	64.2	64.0	61.0	60.9	51.8	4.63	1.24	1.08	30.0
		2Kx2K	193	693	220	-	-	224	224	217	217	198	3.15	1.11	1.03	31.2
Red-Black SOR	16	128x128	1.7	29.8	4.8	-	-	4.9	5.1	4.8	3.9	3.6	6.21	1.33	2.12	7.6
		256x256	6.7	66.7	11.8	-	-	10.5	10.6	11.4	9.7	9.3	5.65	1.27	1.39	11.5
		512x512	26.3	158	33.9	-	-	31.5	31.8	33.9	30.8	29.7	4.65	1.14	1.15	14.2
		1Kx1K	109	397	122	-	-	118	118	122	116	114	3.25	1.07	1.05	15.3
		2Kx2K	437	971	462	-	-	453	454	457	448	442	2.10	1.04	1.01	15.8
	32	128x128	0.8	30.5	4.1	-	-	4.0	4.2	4.1	3.3	3.1	7.44	1.32	3.88	8.3
		256x256	3.3	63.6	8.6	-	-	7.3	7.4	8.0	6.3	5.8	7.40	1.48	1.76	18.2
		512x512	13.2	148	21.0	-	-	18.5	18.9	20.8	17.7	16.6	7.05	1.26	1.30	25.4
		1Kx1K	54.3	342	68.3	-	-	63.9	64.7	67.3	62.2	59.5	5.03	1.15	1.11	29.2
		2Kx2K	217	766	245	-	-	236	238	241	231	226	3.13	1.09	1.04	30.7

Table 6.2 Performance of Optimizations to Reduce and Hide Communication Overhead (in milliseconds)

processor before performing the local computation. In *sr*, *br*, and *cc* the partial results are computed in parallel by each processor, then accumulated using individual send/receives, broadcast/receives, or collective communication, respectively. These timings are also plotted in Figure 6.2.

The largest improvements (4–22) were measured for $\frac{seq}{sr}$, making discovering and extracting parallelism the most important optimization for reduction and scan operations. As expected, the benefit of exploiting parallelism increases with both the problem size and number of processors. Timings show that broadcasts can accumulate partial results quicker than sending individual messages, and specialized collective communication is even more efficient.

The values for $\frac{sr}{cc}$ (1–3.3) show that collective communication can provide large improvements over simple messages. Unlike other communication overhead optimizations, the impact of collective communication increases with the number of processors, even when the amount of computation per processor remains constant. From the values for $\frac{cc}{nc}$ (1–3.7) we conclude that communication overhead can become a major component of execution time for reductions and scans, especially when employing large numbers of processors. The values calculated for $\frac{nc \times P}{cc}$ show that close to linear speedups were measured for reductions. Scans achieved only about half of linear speedup, because computation is doubled in the parallel scan.

6.2.3 Optimizations for Pipelined Computations

Timings for pipelined computations are tabulated in Table 6.4 and graphically displayed in Figure 6.3. In all three kernels, the original loop structure and data distribution is such that message pipelining (*mp*) yields parallelism only for the last outer loop iteration, and message vectorization (*mv*) sequentializes the computation. Loop interchange is needed in order to enable fine-grain pipelining (*fgp*) in these kernels. We present measurements for these worst-case examples of *mv* and *mp* to illustrate potential pitfalls if the compiler cannot reorder computation through loop interchange. Results for unbuffered messages are not

Kernel	*P*	*N*	*nc*	*seq*	*sr*	*br*	*cc*	$\frac{seq}{sr}$	$\frac{sr}{cc}$	$\frac{cc}{nc}$	$\frac{nc \times P}{cc}$
Liv 3 *Inner Product*	8	64K	2.6	22	3.5	3.3	3.3	6.29	1.08	1.27	6.3
		256K	10.0	86	11.0	11.0	11.0	7.82	1.03	1.10	7.3
		1024K	43.5	348	44.5	44.1	44.2	7.82	1.01	1.02	7.9
	16	64K	1.3	23	3.4	2.5	2.1	6.76	1.59	1.62	9.9
		256K	5.3	86	7.4	6.6	6.1	11.6	1.21	1.15	13.9
		1024K	21.7	347	23.8	22.9	22.6	14.6	1.05	1.04	15.3
	32	64K	0.7	24	5.5	3.1	1.6	4.36	3.33	2.29	14.0
		256K	2.6	86	7.5	5.0	3.6	11.5	2.07	1.38	23.1
		1024K	10.7	345	15.6	13.1	11.7	22.1	1.33	1.09	29.3
Liv 11 *First Sum*	8	64K	2.3	19	4.6	4.3	4.2	4.13	1.10	1.82	4.4
		256K	8.9	73	15.5	15.4	15.1	4.71	1.03	1.70	4.7
		1024K	35.6	286	59.0	58.9	58.8	4.85	1.00	1.65	4.8
	16	64K	1.2	20	4.2	3.1	2.7	4.76	1.54	2.25	7.1
		256K	4.5	74	9.5	8.6	8.1	7.79	1.31	1.80	8.9
		1024K	17.9	287	31.2	30.5	30.0	9.20	1.08	1.68	9.5
	32	64K	0.6	21	5.8	3.4	2.2	3.62	2.66	3.67	8.7
		256K	2.3	75	8.6	6.1	4.8	8.72	1.77	2.09	15.3
		1024K	8.9	290	19.3	17.2	15.9	15.0	1.22	1.79	19.0

Table 6.3 Performance of Optimizations to Parallelize Reductions and Scans (in milliseconds)

Kernel	*P*	*N*	*nc*	*mp*	*mv*	*dyn*	*fgp*	*cgp* *B=4*	*cgp* *B=8*	*cgp* *B=12*	*cgp* *B=16*	$\frac{mv}{fgp}$	$\frac{fgp}{best}$	$\frac{best}{nc}$	$\frac{nc \times P}{best}$
Liv 23	16	256x256	24	613	395	-	86	51	49	52	63	4.59	1.76	2.04	7.8
		512x512	96	1990	1550	-	222	156	148	152	171	6.98	1.50	1.54	10.4
		1Kx1K	383	7190	6160	-	677	537	507	508	539	9.10	1.34	1.32	12.1
	32	256x256	12	847	412	-	78	40	39	43	55	5.28	2.00	3.25	9.8
		512x512	48	2480	1580	-	171	102	99	105	129	9.24	1.72	2.06	15.5
		1Kx1K	191	8280	6210	-	441	311	298	308	348	14.1	1.48	1.56	20.5
SOR	16	512x512	23	834	400	-	146	60	49	48	59	2.74	3.04	2.09	7.7
		1Kx1K	107	2730	1750	-	330	177	150	143	167	5.30	2.31	1.34	12.0
		2Kx2K	429	9280	6950	-	947	598	519	493	534	7.34	1.92	1.14	13.9
	32	512x512	12	431	135	-	145	50	40	39	52	2.97	3.72	3.25	9.8
		1Kx1K	53	1800	382	-	321	121	99	96	123	5.61	3.34	1.81	17.7
		2Kx2K	213	7010	1180	-	806	360	308	288	340	8.70	2.80	1.35	23.7
ADI	16	512x512	51	933	496	288	175	95	79	77	90	2.83	2.27	1.51	10.6
		1Kx1K	204	2920	1960	1110	475	315	270	261	285	4.13	1.82	1.28	12.5
		2Kx2K	817	10100	7710	4820	1520	1140	991	959	1000	5.07	1.59	1.17	13.6
	32	512x512	26	1440	515	166	162	70	56	55	69	3.18	2.95	2.12	15.1
		1Kx1K	102	4020	1970	614	383	196	163	158	183	5.14	2.42	1.55	20.7
		2Kx2K	408	12500	7630	2530	1100	644	547	531	573	6.94	2.07	1.30	24.6

Table 6.4 Performance of Optimizations to Exploit Pipeline Parallelism (in milliseconds)

displayed. They degraded the performance of both fine and coarse-grain pipelining since message sizes are too small to compensate for increased startup costs.

The values for $\frac{mv}{fgp}$ (2.7–14) show that it is essential to exploit parallelism for pipelined computations, particularly as the number of processors increases. The best overall timings, *best*, were achieved using coarse-grain pipelining. A block size of eight resulted in the best times for Livermore 23; a block size of twelve proved best for SOR and ADI integration. Results for $\frac{fgp}{best}$ (1.3–3.7) show that coarse-grain pipelining can significantly improve performance when compared to fine-grain pipelining. Values for $\frac{nc \times P}{best}$ indicate coarse-grain pipelining can achieve respectable speedup for pipelined computations.

The performance of dynamic data decomposition displays almost linear speedup with respect to the number of processors (speedup of 1.7–1.9 going from 16 to 32 processors), at least for the problem sizes tested. However, applying dynamic data decomposition to redistribute arrays in ADI proved to be undesirable and required significantly more time than pipelining, especially as problem size increases. Dynamic data decomposition should prove to be a profitable optimization on machines employing larger numbers of processors, but more experimentation is needed.

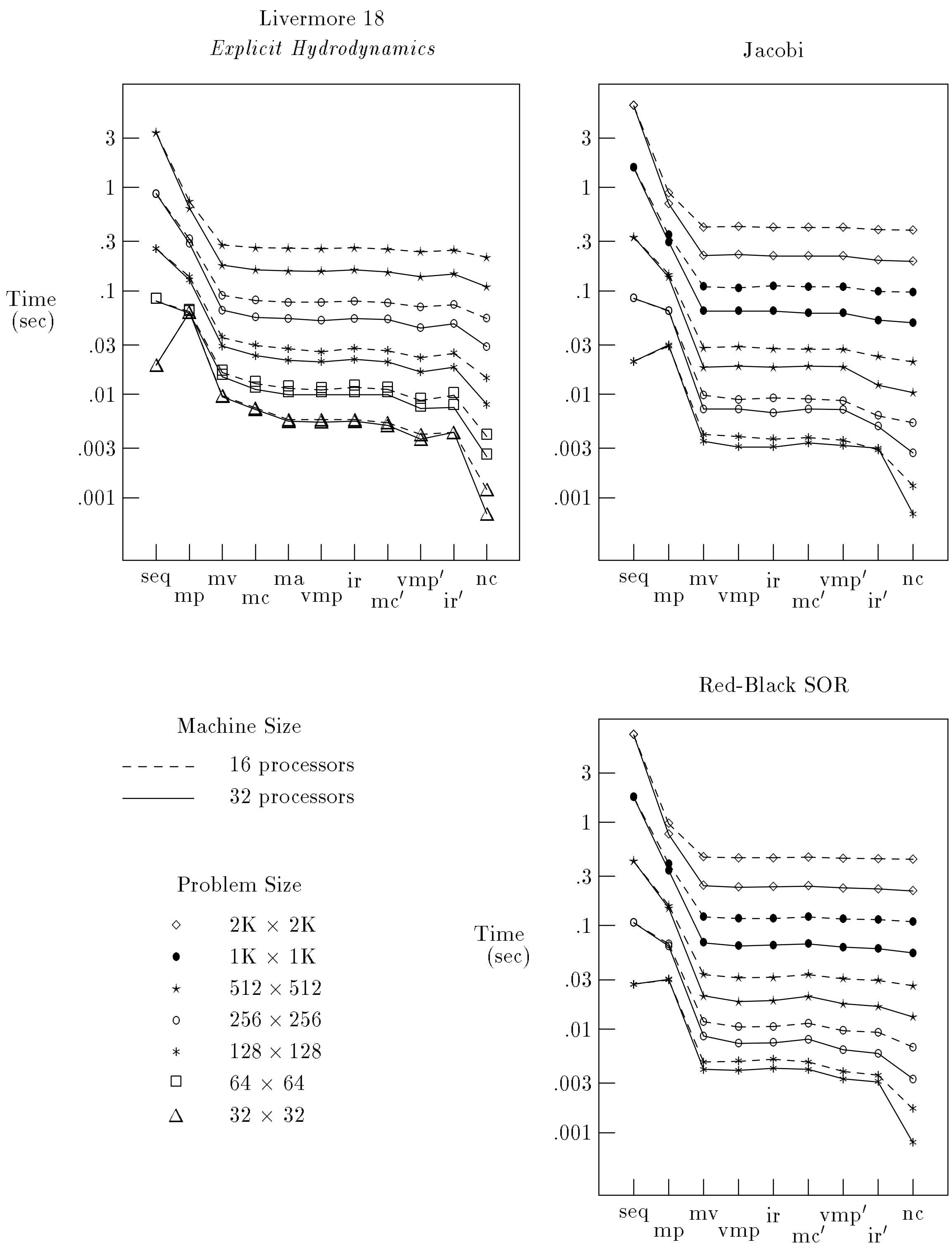


Figure 6.1 Results for Communication Optimizations

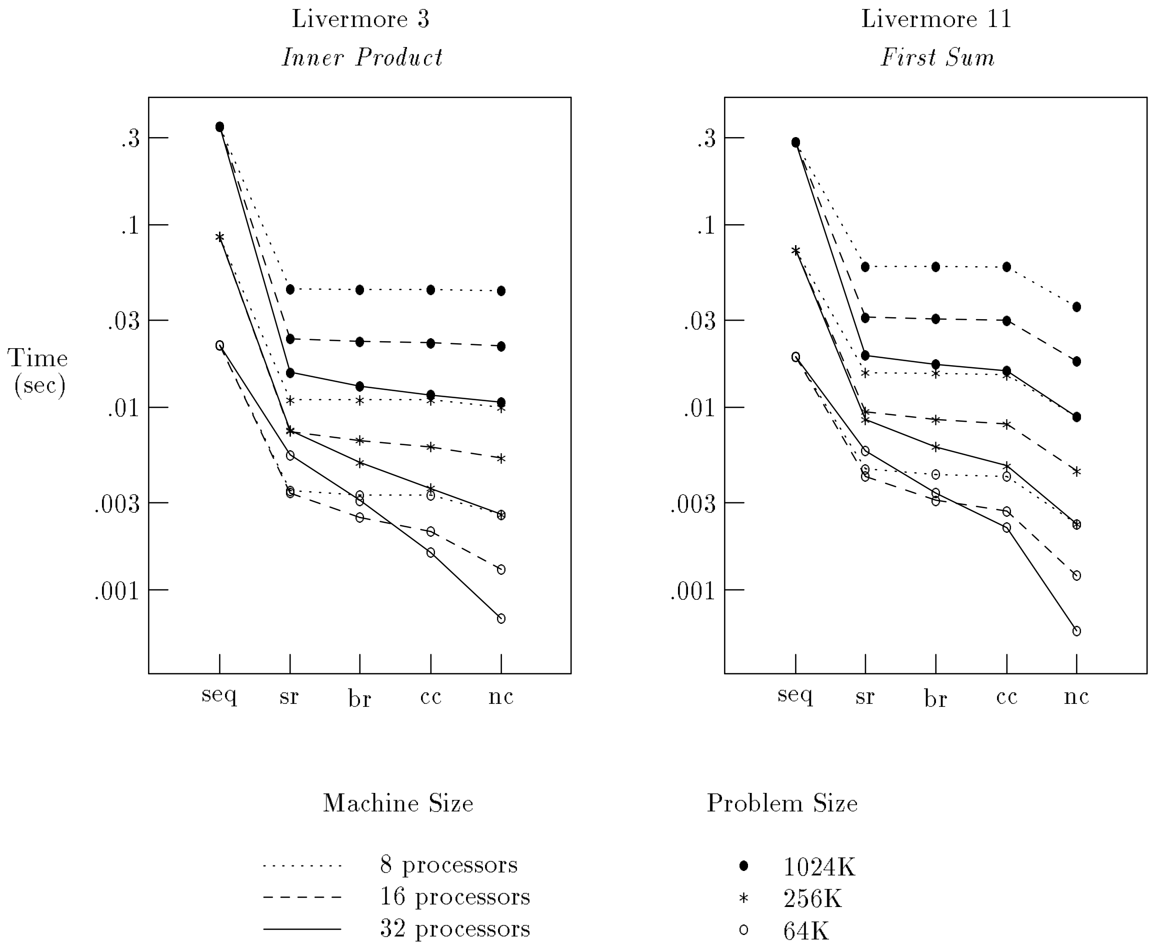


Figure 6.2 Results for Reductions and Scans

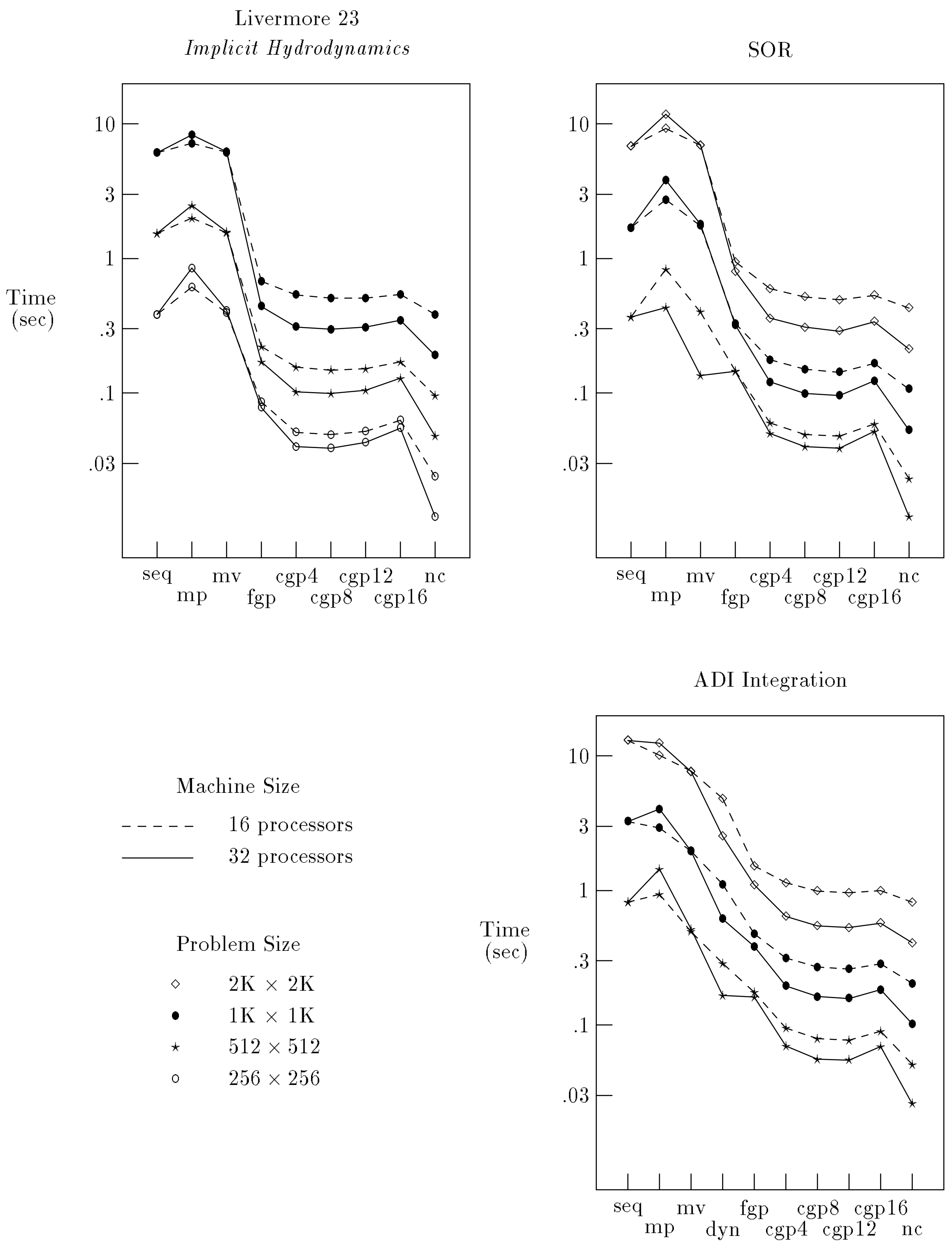


Figure 6.3 Results for Parallelism Optimizations

Opt	Resulting Communication Overhead (For n Elements)
none	$n(T_{start} + T_{copy}(1) + T_{transit}(1))$
mp	$n(T_{start} + T_{copy}(1) + \mathrm{pos}(T_{transit}(1) - T_{comp}))$
mv	$T_{start} + T_{copy}(n) + T_{transit}(n)\ [+T_{buf}(n)]$
ma	$T_{start} + T_{copy}(mn) + T_{transit}(mn) + mT_{buf}(n)$
vmp	$T_{start} + T_{copy}(n) + \mathrm{pos}(T_{transit}(n) - T_{comp})$
ir	$T_{start} + T_{copy}(n) + \mathrm{pos}(T_{transit}(n) - T'_{comp}) + \Delta T_{comp}$
mv'	$T'_{start} + T_{transit}(n)$
vmp'	$T'_{start} + \mathrm{pos}(T_{transit}(n) - T_{comp})$
ir'	$T'_{start} + \mathrm{pos}(T_{transit}(n) - T'_{comp}) + \Delta T_{comp}$

Table 6.5 Effect of Compiler Optimizations

6.3 Analysis of Optimizations

This section presents analysis and decision algorithms to evaluate the cost and effectiveness of the communication and parallelism optimizations presented in the previous chapter. They may be used by the Fortran D to determine when an optimization may be profitably applied. The success of these decision algorithms depends on how accurately we can estimate the cost of different computation and communication operations for the underlying machine.

6.3.1 Communication Optimizations

We begin by analyzing the effects of communication optimizations on communication overhead. Table 6.5 provides the cost of sending one message with n elements for each optimization (the cost for message aggregation (ma) represents m messages). These formulas for communication overhead are presented using T_{start}, T_{copy}, $T_{transit}$ and some new terms. $T_{buf}(n)$ describes the cost of buffering n noncontiguous data elements for message vectorization (mv). It is placed in square brackets [ ] because it is only incurred if data is noncontiguous. T_{buf} may also be be ignored if the underlying architecture can efficiently communicate noncontiguous data. We assume it is not needed for optimizations that include message vectorization.

$Pos()$ is a function that returns the value of its argument if it is positive, zero otherwise. T_{comp} represents the amount of computation between a pair of calls to *send* and *recv* that may be used to hide communication cost. T'_{comp} includes the computation available after applying iteration reordering. ΔT_{comp} describes the increase in computation time caused by iteration reordering. T'_{start} is the startup cost of using unbuffered messages.

The Fortran D compiler always applies message coalescing, vector message pipelining, and collective communication where applicable, since these optimizations improve performance in all cases. In the following sections, we describe profitability criteria for other communication optimizations. These criteria are derived directly from Table 6.5, but are simplified where possible. These formulas can also be used to calculate the expected savings of each optimization. For simplicity we regard copy time as linear, treating $T_{copy}(n)$ and $nT_{copy}(1)$ as equal quantities.

Message vectorization

To send n elements, message vectorization is profitable over message pipelining (assuming mp can hide $T_{transit}$) when:

$$mp > mv$$
$$\Downarrow$$
$$(n-1)T_{start} > T_{transit}(n)\ [+T_{buf}(n)]$$

The compiler thus needs to compare the reduction in startup time against the transit time and cost of buffering noncontiguous data. When startup costs are high, as on the iPSC/860, message vectorization will significantly outperform message pipelining for large values of n.

Message aggregation

To send m messages of size n, message aggregation is profitable over message vectorization when:

$$mv > ma$$
$$\Downarrow$$
$$(m-1)T_{start} + mT_{transit}(n) > T_{transit}(mn)\ \ [+ mT_{buf}(n)\,]$$

If the transit time for m messages of size n is similar to that for one message of size mn, the primary overhead of message aggregation is the cost of copying all messages to a single buffer. If the individual messages are not contiguous, then message aggregation is always profitable since message vectorization performs buffering in any case. Otherwise it is profitable only if the reduction in startup time is greater than the extra buffering cost.

Unbuffered messages

Using unbuffered messages eliminates T_{copy} by eliminating copying on the sending and receiving processors. However, the resulting program incurs a higher startup cost T'_{start}. It is profitable to use unbuffered messages with vector message pipelining when:

$$vmp > vmp'$$
$$\Downarrow$$
$$T_{comp} \geq T_{copy}(n) > (T'_{start} - T_{start})$$

The compiler will use unbuffered messages if sufficient local computation exists to hide copy cost, and the copy cost is greater than the increased startup cost. Since the savings in copy time increases with n, unbuffered messages become more useful as message size increases.

Iteration reordering

Iteration reordering makes additional local computation available, but may also affect code size, data reuse, and conventional scalar optimizations, increasing the total computation time. For instance, empirical results show that iteration reordering does not affect computation costs for Jacobi and Red-Black SOR, but slightly degrades performance for Livermore 18, a kernel that contains significant amounts of computation and data reuse. With buffered messages, iteration reordering can profitably enhance vector message pipelining when the following conditions hold:

$$vmp > ir$$
$$\Downarrow$$
$$T'_{comp} \geq T_{transit}(n) > T_{comp}$$
$$T_{transit}(n) - T_{comp} > \Delta T_{comp}$$

Iteration reordering should thus be applied if the message transit time is not completely hidden by vector message pipelining, and iteration reordering can extract sufficient local computation to hide the remaining transit time. In addition, the savings in transit time must be greater than the increased computation time. Iteration reordering using unbuffered messages is profitable when:

$$vmp' > ir'$$
$$\Downarrow$$
$$T'_{comp} \geq T_{copy}(n) + T_{transit}(n) > T_{comp}$$
$$T_{copy}(n) + T_{transit}(n) - T_{comp} > \Delta T_{comp} + T'_{start} - T_{start}$$

The criteria are similar to that of buffered messages, except that both copy and transit times are considered.

The usefulness of iteration reordering hinges on the value of ΔT_{comp}, which is quite difficult to predict. Our strategy is to simply estimate ΔT_{comp} as some small fixed percentage of the total computation time. It can then be compared against the message copy and transit times to determine whether iteration reordering is worthwhile.

6.3.2 Parallelism Optimizations

In this section we analyze optimizations to exploit parallelism. It is essential that computation be partitioned, even for private variables. Reductions and scans should always be identified and parallelized, using collective communication to accumulate results. Dynamic data decomposition may extract parallelism, but usually with high cost. We show analytically that exploiting pipeline parallelism through either fine-grain or coarse-grain pipelining is both effective and scalable.

Dynamic Data Decomposition

The previous sections show how parallel computation time can be estimated for pipelined computations. The compiler needs to compare it with the estimated cost for dynamic data decomposition (based on training sets) to determine whether it is more profitable than applying pipelining. Dynamic data decomposition is likely to be profitable only for small problems, because communication to redistribute data becomes less efficient as problem size increases. In comparison, the efficiency of pipelining improves with larger problem sizes.

Fine-grain Pipelining

Consider the simple examples presented in Figure 6.4. We define n as the number of elements along one dimension, p as the number of processors, and C as the communication overhead for each message. We normalize all costs by the cost required to compute one element, so the sequential computation time is equal to the number of data elements.

For simplicity we restrict our analysis to cases where we can interchange cross-processor loops to the *innermost* position, allowing program execution to first proceed along the distributed dimensions. This enables both fine-grain and coarse-grain pipelining. Fortunately, most if not all pipelined computations meet this requirement. For instance, loop interchange of cross-processor loops is legal for both SOR and ADI integration.

Using these assumptions, we can now calculate the time required to compute an $n{\times}n$ data array distributed block-wise in one dimension. Each processor begins execution exactly $\frac{n}{p}+C$ units later than its predecessor, where $\frac{n}{p}$ is the time for its predecessor to compute one column and C is the communication overhead. The time it takes each processor to finish its computation is $\frac{n^2}{p}$, the total computation time, plus nC, the time spent to send and receive n messages. The total parallel execution time is $(p-1)(\frac{n}{p}+C)$, the delay before the last processor begins, plus $\frac{n^2}{p}+nC$, the time required by the processor to finish computing.

Similar calculations for the $n\times 2n$, $2n\times n$, and $n\times n\times n$ example arrays result in the formulas shown in Figure 6.4. Examining the expressions, we see that the dominating term in the parallel execution time is simply (*sequential time*)$/p$. Pipeline parallelism under these conditions thus approaches perfect speedup for large problem sizes.

Coarse-grain Pipelining

The same model may also be used to calculate an efficient buffered factor for coarse-grain pipelining. Assume we strip-mine and interchange the outer loop in the pipelined computation of an $n\times n$ array by a constant block factor B, as in Figure 5.10. The delay between processors increases to $\frac{nB}{p}+C$, since B columns each costing $\frac{n}{p}$ are computed before sending a message. However, the total communication overhead for one processor drops from nC to $\frac{nC}{B}$. The total parallel execution time is thus $\frac{n^2}{p}+\frac{nC}{B}+(p-1)(\frac{nB}{p}+C)$. The times for other examples are shown in Figure 6.4.

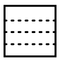
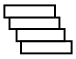
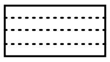
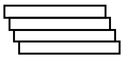
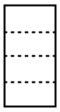
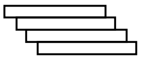
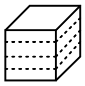
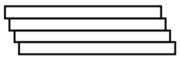
Data Arrays & Data Partition	Computation & Elapsed Time	Sequential Time	Parallel Time (fgp) & Parallel Time Blocked (cgp)	Block Size Preferred
		n^2	$fgp = \frac{n^2}{p} + nC + (p-1)(\frac{n}{p} + C)$ $cgp = \frac{n^2}{p} + \frac{nC}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{C}$
		$2n^2$	$fgp = \frac{2n^2}{p} + 2nC + (p-1)(\frac{n}{p} + C)$ $cgp = \frac{2n^2}{p} + \frac{2nC}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{2C}$
		$2n^2$	$fgp = \frac{2n^2}{p} + nC + (p-1)(\frac{2n}{p} + C)$ $cgp = \frac{2n^2}{p} + \frac{nC}{B} + (p-1)(\frac{2nB}{p} + C)$	$\sqrt{\frac{C}{2}}$
		n^3	$fgp = \frac{n^3}{p} + n^2C + (p-1)(\frac{n}{p} + C)$ $cgp = \frac{n^3}{p} + \frac{n^2C}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{nC}$

Figure 6.4 Effectiveness of Pipelining

As we can see, the asymptotic speedup is unchanged by B, but the total communication overhead can be significantly decreased at the expense of some parallelism. To determine the minimal cost while holding n and p constant, we differentiate the expression for parallel execution time with respect to B and set the result to zero. This yields the following equation and solution for B:

$$-\frac{nC}{B^2} + \frac{n(p-1)}{p} = 0$$

$$B = \sqrt{\frac{pC}{p-1}} \approx \sqrt{C} = \sqrt{\frac{\text{block communication cost}}{\text{element computation cost}}}$$

Since C has been normalized by the computation required to calculate one array element, it is actually the ratio of communication to computation cost. As expected, the results show that larger block sizes are preferred when the ratio of communication to computation cost is high; smaller blocks are desirable when communication cost is relatively low. More importantly, these formulas allow the compiler to calculate efficient block sizes and estimated execution times for pipelined computations.

Our analysis for pipelined computations is somewhat imprecise since it assumes that communication cost is fixed as the message size increases. Fortunately this is relatively true for the small block sizes that are selected. More accurate analytical models can be developed, but may be hindered by unpredictable system discontinuities. For instance, communication cost increases abruptly past 100 bytes on the iPSC/860 [26]. The Fortran D compiler will employ a flexible and precise approach using *training sets* to estimate communication and computation costs [17, 103, 113, 114]. Accurate static estimates of communication and computation are also needed by the compiler to calculate block sizes for coarse-grain pipelining.

6.4 Scalability

The *scalability* of an optimization describes whether its effectiveness increases, decreases, or remains constant in proportion to some characteristic. In this section we use scalability to summarize our insights concerning the usefulness of communication and parallelism optimizations. Our conclusions are derived from the

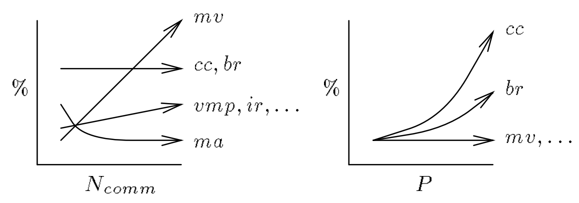


Figure 6.5 Effect on Communication Overhead

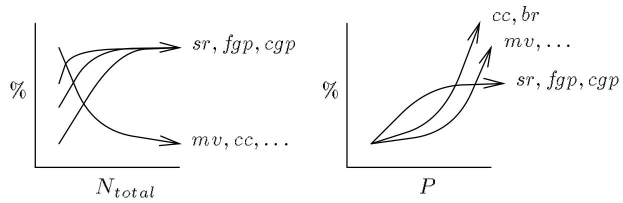


Figure 6.6 Effect on Program Execution Time

empirical and analytical results presented in the previous sections. In the following discussion we define N_{comm} to be the number of elements communicated by each processor and N_{total} to be the total number of elements. For convenience, we also use N_{local} to describe the number of elements on each processor. It is simply N_{total}/P, where P is the number of processors.

6.4.1 Communication overhead

Figure 6.5 shows the scalability of optimizations in eliminating communication overhead. The effectiveness of message vectorization (*mv*) is displayed as improvement over message pipelining. Collective communication (*cc*) and broadcast/receive (*br*) are shown as gains over send/receive (*sr*). The effectiveness of other optimizations are displayed as improvement compared with message vectorization. All improvements are shown as percentages.

We first consider how communication optimizations scale with respect to N_{comm}, the amount of data communicated by each processor. When we increase N_{comm} for a fixed number of processors, message vectorization (*mv*) improves most rapidly because it eliminates entire messages. Other optimizations (*vmp*, *ir*, ...) improve less quickly since they only affect message transit and copy times. The effectiveness of collective communication (*cc*) and broadcast/receive (*br*) remains unchanged at a level determined by the number of processors. The percentage improvement for message aggregation (*ma*) decreases because its usefulness is set by the number of arrays communicated to the same processor.

In comparison, when N_{comm} is fixed, most communication optimizations (*mv*, *vmp*, ...) are not enhanced by increasing the number of processors. Only collective communication (*cc*) and broadcast (*br*) improve in their ability to eliminate communication cost as P grows.

6.4.2 Program execution

Figure 6.6 displays the scalability of optimizations in reducing total execution time. We assume that computation cost is proportional to N_{total}. Optimizations to exploit parallelism (*sr*, *fgp*, *cgp*) are expressed as improvements relative to the sequential execution time. For a fixed number of processors P, they increase in effectiveness as N_{total} grows, reaching a plateau at the number of processors. In comparison, communication optimizations (*mv*, *cc*, ...) shrink in relative usefulness because N_{comm} grows slowly compared to N_{total} for stencil computations.

The situation is more complex when a problem with fixed size is parallelized using an increasing number of processors. Initially the amount of communication is small relative to the local problem size ($N_{comm} \ll$

N_{local}) so parallelism optimizations achieve excellent speedup, increasing linearly with P. At this stage communication optimizations only attain modest improvements, though collective communication and broadcast/receive improve more quickly.

As we show in the next section, eventually the problem is divided among enough processors that N_{comm} becomes a large percentage of N_{local}. When this point is reached, communication overhead begins to have a significant impact on execution time. Growth in the effectiveness of parallelism optimizations slows because of communication costs, while communication optimizations quickly increase in importance. How soon this point is reached depends on the communication overhead relative to computation costs.

6.4.3 Communication vs. computation

We have seen that parallelism optimizations are critical for improving overall program execution time, regardless of the problem or machine size. In comparison, the effectiveness of communication optimizations is dependent on N_{comm}, the amount of data that must be communicated. Understanding the relationship between N_{comm}, N_{total}, and N_{local} is thus crucial to determining the impact of communication optimizations.

Simple geometric analysis shows that the growth of N_{comm} relative to N_{total} varies for different data distributions. For instance, when a 2D array with n elements is distributed 1D block-wise across p processors, each processor owns a $\sqrt{n} \times \frac{\sqrt{n}}{p}$ section of the array. Assuming a stencil computation that only accesses boundary elements, a processor needs to send $\sqrt{n}$ array elements to each neighboring processor, communicating $2\sqrt{n}$ elements. Similar analyses for other examples result in the formulas for calculating N_{comm} displayed in Table 6.6.

Table 6.6 also presents relative values of N_{comm} for three different problem sizes on a machine with eight processors. Though it varies depending on the problem and machine dimensionality, N_{comm} always grows less rapidly than N_{local}. This implies that communication optimizations become less important as problem size grows. For large problems, message vectorization and collective communication are likely to yield most of the available benefits.

On the other hand, consider the situation when we attempt to speed up a problem of size N_{total} by increasing the number of processors. Similar analysis makes it clear that N_{comm} becomes an increasingly large percentage of N_{local}. Eventually communication overhead becomes the limiting factor, and all of the communication optimizations discussed become important for achieving good speedup.

6.5 Optimization Algorithm

The overall Fortran D compiler optimization algorithm is shown in Figure 6.7. It is intended only to provide a rough outline of how optimizations are organized. The compiler will decide at each point which optimizations are actually worth performing.

Problem Dimension	Dimensions Distributed	N_{comm}	N_{comm}/N_{local} for $p = 8$ & $n = 10^3$	$n = 10^4$	$n = 10^5$
3D	1D	$2\sqrt[3]{n^2}$	1.0	.74	.35
3D	2D	$4\sqrt[3]{n^2}/\sqrt{p}$	1.0	.52	.24
2D	1D	$2\sqrt{n}$	.50	.16	.05
2D	2D	$4\sqrt{\frac{n}{p}}$	.36	.11	.04
3D	3D	$6\sqrt[3]{\frac{n}{p}}$	.24	.05	.01
1D	1D	2	.016	.0016	.00016

Table 6.6 Data Communication Requirements

```
partition data across processors
partition computation using "owner computes" rule
detect and parallelize reductions & scans
compute cross-processor loops
for each loop nest L do
   if L is fully parallel (i.e., no cross-processor loops) then
      vectorize, coalesce, and aggregate messages
      select and insert collective communications
      if sufficient T_comp exists to hide T_copy, T_transit then
         apply vector message pipelining
         insert unbuffered messages
      else if T'_comp can be profitably created & used then
         reorder iterations
         apply vector message pipelining
         insert unbuffered messages
      else
         insert buffered messages
      endif
   else      {* must be pipelined computation *}
      select efficient granularity for pipelining
      apply strip-mining & loop iterchange
      vectorize, coalesce, and aggregate messages
      insert buffered messages
   endif
   if insufficient storage is available then
      apply storage optimizations
   endif
enddo
```

Figure 6.7 Fortran D Optimization Algorithm

6.6 Discussion

Empirically measured results for stencil computations show that exploiting parallelism for pipelined computations, reductions, and scans is vital. Message vectorization, coarse-grain pipelining, and collective communication also significantly affect performance by eliminating large numbers of messages. The remaining optimizations yield less dramatic results, but are still important when the proportion of nonlocal to local data is high. This is the case when attempting to speed up a problem with fixed size. Profitability formulas enable the Fortran D compiler to intelligently choose between optimization options, but rely on accurate measurements of machine parameters through the use of training sets.

Chapter 7

Interprocedural Compilation

Algorithms exist for compiling Fortran D for MIMD distributed-memory machines, but are significantly restricted in the presence of procedure calls. This chapter presents interprocedural analysis, optimization, and code generation algorithms for Fortran D that limit compilation to only one pass over each procedure. This is accomplished by collecting summary information after edits, then compiling procedures in reverse topological order to propagate necessary information. Delaying instantiation of the computation partition, communication, and dynamic data decomposition is key to enabling interprocedural optimization. Recompilation analysis preserves the benefits of separate compilation. Empirical results show that interprocedural optimization is crucial in achieving acceptable performance for a common application.

7.1 Introduction

As we have seen the Fortran D compiler requires deep analysis because it must know both *when* a computation may be performed and *where* the data and computation is located. The compiler is thus severely restricted by the limited program context available at procedures. This limitation is unfortunate since procedures are desirable for programming style, modularity, readability, code reuse, and maintainability.

Interprocedural analysis and optimization algorithms have been developed for scalar and parallelizing compilers, but are seldom implemented. We show that interprocedural analysis and optimization can no longer be considered a luxury, since the cost of making conservative assumptions at procedure boundaries is unacceptably high when compiling data-placement languages such as Fortran D. The major contribution of this chapter is to demonstrate efficient interprocedural Fortran D compilation techniques. We have begun implementing these techniques in the current compiler prototype.

In the remainder of this chapter, we illustrate the need for interprocedural compilation and show how the Fortran D compiler is integrated into the ParaScope interprocedural framework. We present analysis, optimization, and code generation algorithms in detail for a number of interprocedural problems, then provide the overall interprocedural compilation algorithm. Recompilation tests are described that preserve the benefits of separate compilation. A case study of DGEFA is used to demonstrate the effectiveness of interprocedural analysis and optimization.

7.2 Interprocedural Support in ParaScope

ParaScope is a programming environment for scientific Fortran programmers. It has fostered research on aggressive optimization of scientific codes for both scalar and shared-memory machines [35]. Its pioneering work on incorporating interprocedural optimization in an efficient compilation system has also contributed the development of the Convex Applications compiler [146]. Through careful design, the compilation process in ParaScope preserves separate compilation of procedures to a large extent. Tools in the environment cooperate so that a procedure only needs to be examined once during compilation. Additional passes over the code can be added if necessary, but should be avoided since experience has shown that examination of source code dominates analysis time. The existing compilation system uses the following 3-phase approach [35, 62, 92]:

1. **Local Analysis.** At the end of an editing session, ParaScope calculates and stores summary information concerning all local interprocedural effects for each procedure. This information includes details on call sites, formal parameters, scalar and array section uses and definitions, local constants, symbolics, loops and index variables. Since the initial summary information for each procedure does not depend on interprocedural effects, it only needs to be collected after an editing session, even if the program is compiled multiple times or if the procedure is part of several programs.

2. **Interprocedural Propagation.** The compiler collects local summary information from each procedure in the program to build an *augmented call graph* containing loop information [94]. It then propagates the initial information on the call graph to compute interprocedural solutions.

3. **Interprocedural Code Generation.** The compiler directs compilation of all procedures in the program based on the results of interprocedural analysis.

Another important aspect of the compilation system is what happens on subsequent compilations. In an interprocedural system, a module that has not been edited since the last compile may require recompilation if it has been indirectly affected by changes to some other module. Rather than recompiling the entire program after each change, ParaScope performs *recompilation analysis* to pinpoint modules that may have been affected by program changes, thus reducing recompilation costs [32, 63]. This process is described in greater detail in Section 7.6.

ParaScope computes interprocedural REF, MOD, ALIAS and CONSTANTS. Implementations are underway to solve a number of other important interprocedural problems, including interprocedural symbolic and RSD analysis. ParaScope also contains support for inlining and cloning, two interprocedural transformations that increase the context available for optimization. *Inlining* merges the body of the called procedure into the caller. *Cloning* creates a new version of a procedure for specific interprocedural information [60, 62].

Existing interprocedural analysis in ParaScope is useful for the Fortran D compiler, but it is not sufficient. The compiler must also incorporate analysis to understand the partitioning of data & computation, and to apply communication optimizations. In order to use the above 3-phase approach, additional interprocedural information is collected during code generation and propagated to other procedures in the program. These extensions are described in the rest of the chapter.

7.3 Interprocedural Compilation

As we have seen, interprocedural compilation of Fortran D is needed to generate efficient code in the presence of procedure calls. The Fortran D compilation process is complex. The list of interprocedural data-flow problems that must be solved by the Fortran D compiler is shown in Table 7.1. Each problem is labeled $\downarrow$, $\uparrow$, or $\updownarrow$ depending on whether it is computed top-down, bottom-up, or bidirectional, respectively. We have carefully structured the Fortran D compiler to perform compilation in a single pass over each procedure for programs without recursion. It has three key points. The first two support compilation in a single pass, the third improves the effectiveness of interprocedural optimization:

- Certain interprocedural data-flow problems are computed first because their solutions are needed to enable code generation. In particular, reaching decompositions information is needed to determine the data partition, the initial step in compiling Fortran D. These problems are solved by gathering local information during editing and computing solutions during interprocedural propagation.

- Other interprocedural data-flow problems depend on data produced only during code generation. For instance, local iteration and nonlocal index sets required for optimizations are calculated as part of local Fortran D compilation. While we could introduce additional local analysis and interprocedural propagation phases to solve these problems, it is much more efficient to combine their calculation with code generation. This approach is possible because the set of problems we want to compute during interprocedural code generation are all bottom-up. By visiting procedures in reverse topological order, the results of analysis for each procedure are available when compiling its callers. Only overlaps need to be handled separately.

Interprocedural Propagation	Code Generation
Call graph ↓	Local iteration sets ↑
Loop structure ↓	Nonlocal index sets ↑
Array aliasing & reshaping ↓	Overlaps ↕
Scalar & array side effects ↑	Buffers ↑
Symbolics & constants ↕	Live decompositions ↑
Reaching decompositions ↓	Loop-invariant decomps ↑

Table 7.1 Interprocedural Fortran D Data-flow Problems

- *Delayed instantiation* of the computation partition, communication, and dynamic data decomposition enables optimization across procedure boundaries. In other words, guards, messages, and calls to data remapping routines are not inserted immediately when compiling a procedure. Instead, where legal they are stored and passed to the procedure's callers, delaying their insertion. This technique provides the flexibility needed to perform interprocedural optimization.

The remainder of this section presents interprocedural solutions required by the Fortran D compiler and shows how interprocedural information is used during code generation. Section 7.4 describes additional interprocedural analysis and optimization for efficiently supporting dynamic data decomposition. The overall algorithm is then presented. For clarity, each problem and solution is described separately, even though the compilation process uses the 3-phase ParaScope approach described in the previous section.

7.3.1 Augmented Call Graph

Most interprocedural problems are solved on the call graph, where nodes represent procedures and edges represent call sites. Since the Fortran D compiler also requires information about interprocedural loop nesting, it uses the *augmented call graph* (ACG) [94]. Conceptually, the ACG is simply a call graph plus *loop nodes* that contain the bounds, step, and index variable for each loop, plus *nesting edges* that indicate which nodes directly encompass other nodes.

For instance, the Fortran D program in Figure 7.1 produces the ACG shown in Figure 7.2. The ACG shows that program P1 has two loops, i and j, both of which contain calls to F1. F1 calls F2, which in turn contains loop k. Annotations stored in the ACG show that the formal parameter i in F1 and F2 is actually the index variable for a loop in P1 that iterates from 1 to 100 with a step of 1.

The ACG also contains representations of the formal and actual parameters and their dimensions associated with each procedure and call site. This information is used by interprocedural analysis to translate data-flow sets across calls, mapping formals to actuals and vice versa. An example of this translation is the *Translate* function in Figure 7.3. Translation must also deal with *array reshaping* across procedure boundaries. Interprocedural symbolic analysis used in conjunction with *linearization* and *delinearization* of array references can discover standard reference patterns that may be compiled efficiently [30, 92, 98].

The augmented call graph construction algorithm has a local and interprocedural phase. During local analysis, a node is created for each procedure and augmented with loop information. Loops nodes are created for each loop. For each procedure or loop node X, nesting edges are added to loops directly contained in X. Call sites contained directly in X are also recorded. Loop bounds and step are stored as constants or *jump functions*, functions that describe the value of a variable as a function of input variables to the procedure (*i.e.,* formal parameters and global variables that may have constant values) [36].

During interprocedural propagation call edges are added to the call graph for each call site. Straightforward examination of recorded information is usually sufficient. If a procedure-valued formal parameter is invoked, further analysis is required to determine all procedure names that could be bound to it [92]. For greater precision, jump functions for loop information are evaluated using results from interprocedural constant and symbolic analysis.

```
 PROGRAM P1                          SUBROUTINE F1(Z,i)
   REAL X(100,100),Y(100,100)          REAL Z(100,100)
   PARAMETER (n$proc = 4)        S3    call F2(Z,i)
   ALIGN Y(i,j) with X(j,i)          end
   DISTRIBUTE X(BLOCK,:)
   do i = 1,100                      SUBROUTINE F2(Z,i)
S1   call F1(X,i)                      REAL Z(100,100)
   enddo                               do k = 1,100
   do j = 1,100                          Z(k,i) = F(Z(k+5,i))
S2   call F1(Y,j)                      enddo
   enddo                             end
 end
```

Figure 7.1 Example Fortran D Program

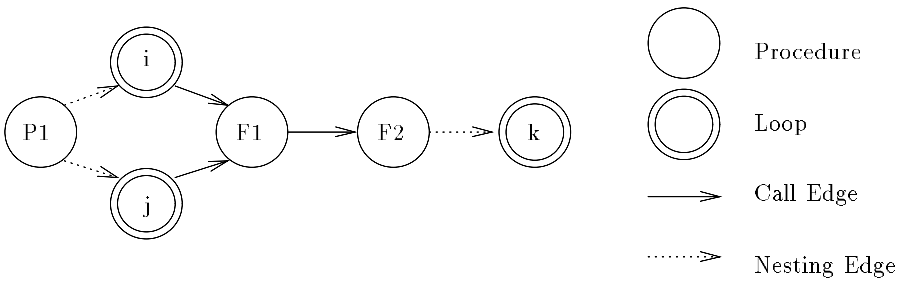


Figure 7.2 Augmented Call Graph

{∗ *Local analysis phase* ∗}
for each procedure P **do**
  initialize decomposition of all variables to $\top$
  for each call site C in P **do**
    calculate LocalReaching(C)
  endfor
endfor
{∗ *Interprocedural propagation phase* ∗}
for each procedure P **do** (in topological order)
  calculate Reaching(P) =
    $\bigcup_{P \text{ invoked at } C}$ *Translate*(LocalReaching(C))
  clone P if multiple decompositions found
  for each call site C in P **do**
    for each element $\langle \top, X \rangle \in$ LocalReaching(C) **do**
      replace with $\langle D, X \rangle \in$ Reaching(P)
    endfor
  endfor
endfor
{∗ *Interprocedural code generation phase* ∗}
for each procedure P **do** (in reverse topological order)
  calculate LocalReaching for all variables in P
endfor

Figure 7.3 Reaching Decompositions Algorithm

7.3.2 Reaching Decompositions

To effectively compile Fortran D programs, it is vital to know the data decomposition of a variable at every point it is referenced in the program. In Fortran D, procedures inherit the data decompositions of their callers. For each call to a procedure, formal parameters inherit the decompositions of the corresponding actual parameters passed at the call, and global variables retain their decomposition from the caller. A variable's decomposition may also be changed at any point in the program, but the effects of decomposition specifications are limited to the scope of the current procedure and its descendants in the call graph.

Reaching Decompositions Calculation. To determine the decomposition of distributed arrays at each point in the program, the compiler calculates *reaching decompositions*. Locally, it is computed in the same manner as *reaching definitions*, with each decomposition treated as a "definition" [4]. Interprocedural reaching decompositions is a *flow-sensitive* data-flow problem [20, 61] since dynamic data decomposition is affected by control flow. However, the restriction on the scope of dynamic data decomposition in Fortran D means that reaching decompositions for a procedure is only dependent on control flow in its callers, not its callees. The effect of data decomposition changes in a procedure can be ignored by its callers, since it is "undone" upon procedure return.

By taking advantage of this restriction, interprocedural reaching decompositions may be solved in one top-down pass over the call graph using the algorithm in Figure 7.3. During local analysis, we calculate the decompositions that reach each call site C. Formally,

> $\textsc{LocalReaching}(X) = \{ \langle D, V \rangle \mid D$ is the set of decomposition specifications reaching actual parameter or global variable V at point X $\}$.

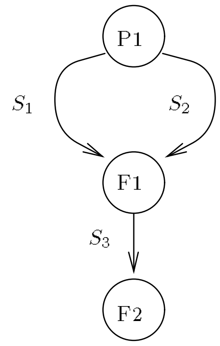
LocalReaching may include elements of the form $\langle \top, V \rangle$ if V may be reached by a decomposition inherited from a caller. $\top$ serves as a placeholder. During interprocedural propagation, we use the call graph and LocalReaching to calculate $\textsc{Reaching}(P)$, the set of decompositions reaching a procedure P from its callers. Formally,

> $\textsc{Reaching}(P) = \{ \langle D, V \rangle \mid D$ is the set of decomposition specifications reaching formal parameter or global variable V at procedure P $\}$.

The function *Translate* maps actual parameters in the LocalReaching set of a call to formal parameters in the called procedure. Global variables are simply copied, and actual parameters are replaced by the corresponding formal parameters. $\textsc{Reaching}(P)$ is computed as the union of the translated LocalReaching sets for all calls to P. We then update all LocalReaching sets in P that contain $\top$. Each element $\langle \top, V \rangle$ is expanded to $\langle D, V \rangle$, where D is the set of decompositions for variable V in $\textsc{Reaching}(P)$. This step propagates decompositions along paths in the call graph. During code generation the compiler needs to determine which decomposition reaches each variable reference. It repeats the calculation of LocalReaching for each procedure, taking Reaching into account.

Reaching Decompositions Example. Figure 7.4 illustrates the reaching decomposition calculation for the program in Figure 7.1. During the local analysis phase, LocalReaching sets are computed for the call sites S_1, S_2 and S_3. The results for S_1 and S_2 contain the decompositions that reach the actual parameter at the call site. At the first call site S_1, the actual parameter X is distributed row-wise. At the second call site S_2, Y is distributed column-wise. $\textsc{LocalReaching}(S_3)$ is set to the element $\langle \top, Z \rangle$ since the decomposition inherited by procedure F1 reaches Z.

During the interprocedural propagation phase, the call graph is constructed and Reaching sets are computed top-down for program P1 and procedures F1 and F2. Reaching(P1) is the empty set, since P1 has no callers. Reaching(F1) is calculated as the union of LocalReaching for the call sites S_1 and S_2. The *Translate* function maps the decomposition of the actual parameters X and Y at the call sites to the formal Z, resulting in $\langle \{(:, \mathit{block}), (\mathit{block}, :)\}, Z \rangle$. $\top$ for Z in $\textsc{LocalReaching}(S_3)$ is replaced with these column and row distributions from Reaching(F1). Since S_3 is the only call site invoking F2, the resulting data decompositions are also assigned to Reaching(F2). Finally, during local code generation the interprocedural reaching decompositions in Reaching are used to calculate the decomposition for each local variable.



$\textsc{LocalReaching}(S_1) = \{\, \langle\, (block, :), X \,\rangle \,\}$

$\textsc{LocalReaching}(S_2) = \{\, \langle\, (:, block), Y \,\rangle \,\}$

$\textsc{LocalReaching}(S_3) = \{\, \langle\, \top, Z \,\rangle \,\}$

$\textsc{Reaching}(\mathrm{P1}) = \emptyset$

$\textsc{Reaching}(\mathrm{F1}) = \textsc{LocalReaching}(S_1) \cup \textsc{LocalReaching}(S_2)$

$\textsc{Reaching}(\mathrm{F2}) = \textsc{LocalReaching}(S_3)$

Figure 7.4 Reaching Decompositions

partition calls C invoking P into $\{\pi_1 \ldots \pi_n\}$ such that
  Filter(*Translate*($\textsc{LocalReaching}(C)$),$\textsc{Appear}(P)$)
  is equal FORALL calls C in each partition π_i
if $n > 1$ **then** {∗ *multiple partitions created* ∗}
  for each $\pi_i \in \{\pi_1 \ldots \pi_n\}$ **do**
    create clone P_i of P
    calculate $\textsc{Reaching}(P_i) =$
      $\bigcup_{C \ in \ \pi_i}$ *Translate*($\textsc{LocalReaching}(C)$)
    for each call C in π_i **do**
      replace P with P_i as endpoint of edge
        representing C in call graph
    endfor
  endfor
endif

Figure 7.5 Procedure Cloning Algorithm

7.3.3 Procedure Cloning

The Fortran D compiler can generate much more efficient code if there is only a single decomposition reaching an array. We assume that cloning or run-time techniques will be applied locally to ensure that each array has a unique decomposition within each procedure. Procedure cloning may still be necessary if calls to procedure P provide different decompositions for variables that appear in P or its descendants. The procedure cloning algorithm is presented in Figure 7.5. We define $\textsc{Appear}(P)$ to be the set of formal parameters and global variables appearing in procedure P or its descendants. Formally,

$$\textsc{Appear}(P) = \textsc{Gmod}(P) \cup \textsc{Gref}(P).$$

Gmod and Gref represent the variables modified or referenced by a procedure or its descendants [61]. The value of Appear is readily available from interprocedural scalar side-effect analysis [20, 62]. We also define a function *Filter*(R, V) that removes from R all decompositions elements $\langle D, X \rangle$ where $X \notin V$, returning the remaining decomposition elements.

In the algorithm we partition the calls to P so that calls providing the same decompositions can share the same clone. We use *Filter* to remove reaching decompositions that are not in Appear. This step avoids unnecessary cloning that would expose decompositions for unreferenced variables. A clone of P is produced for each partition, resulting in a unique decomposition for each variable accessed. For instance, the compiler creates two copies of procedure F1 and F2 because they possess two different reaching decompositions for Z. Edges in the call graph are updated appropriately for the clone. In pathological cases, cloning can result in an exponential growth in program size [60]. Under these circumstances, cloning may be disabled when a threshold program growth has been exceeded, forcing run-time resolution instead.

```
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   for each variable V in P do
      calculate local index set for V
   endfor
   for each assignment statement S in P do
      construct iteration set for S
   endfor
   for each call to Q at site C in P do
      assign iteration set of Q to C
   endfor
   instantiate local data and computation partitions
   collect union of all iteration sets in P for callers
endfor
```

Figure 7.6 Data and Computation Partitioning Algorithm

7.3.4 Partitioning Data and Computation

Recall that a major responsibility of the Fortran D compiler is to partition the data and computation across processors. Reaching decompositions calculated in LocalReaching are translated into *distribution functions* that compute the data partition for each variable. Once the data partition is calculated, it is used with loop information in the ACG to derive the computation partition via the owner computes rule. In the Fortran D compiler, the data and computation partition are represented by local index and iteration sets, respectively. The computation partition is instantiated by modifying the program text to reduce loop bounds and/or introduce explicit guards.

When compiling a procedure, the Fortran D compiler delays local instantiation of the computation partition as much as possible. It first forms the union of all iteration sets for statements in the procedure. Bounds are reduced for loops local to the procedure. Guards are introduced for loops outside the procedure only if local statements have different iteration sets for those loops. Otherwise the compiler simply saves the unioned iteration set, using it to instantiate the computation partition later when compiling the callers. Delayed instantiation enables the compiler to reduce computation partitioning costs by using loop bounds reduction or by merging guards across procedure boundaries. The partitioning algorithm is shown in Figure 7.6.

Computation Partitioning Example. We illustrate the partitioning process for the code in Figure 7.1. For simplicity, we assume that procedure F1 contains the k loop. Cloning has already been applied to F1, producing F1$row and F1$col as shown in Figure 7.7. The compiler computes the local index set for Z to be [1:25,1:100] in F1$row and [1:100,1:25] in F1$col. Disregarding boundary conditions, applying the owner computes rule results in the local iteration sets [1:25,1:100] and [1:95,1:25] for the assignments to $Z(k,i)$ at S_3 and S_4, respectively. Since these are the only computation statements in F1$row and F1$col, they become the iteration sets for the entire procedures as well.

During code generation, the bounds of local loop k are reduced in F1$row, but not for F1$col. The iteration sets for F1$row and F1$col are stored and assigned to the call sites at S_1 and S_2 when compiling P1. This causes the bounds of the j loop enclosing S_2 to be reduced from [1:100] to [1:25], based on the iteration set calculated for F1$col. The result is shown in Figure 7.7.

7.3.5 Communication Analysis and Optimization

Once they are calculated, local iteration sets (representing the computation partition) may be used to compute nonlocal accesses. Communication is generated only for nonlocal references in procedure P that cause true dependences carried by loops within P. This may be determined from RSDs and local code. Messages for other nonlocal references will be added when P's callers are later compiled. Communication is

```
      PROGRAM P1
        REAL X(30,100), Y(100,25)
        my$p = myproc() {* 0...3 *}
        if (my$p .GT. 0) send X(1:5,1:100) to my$p-1
        if (my$p .LT. 3) recv X(26:30,1:100) from my$p+1
        do i = 1,100
S1        call F1$row(X,i)
        enddo
        do j = 1,25
S2        call F1$col(Y,j)
        enddo
      end
      SUBROUTINE F1$row(Z,i)
        REAL Z(30,100)
        ub$1 = min((my$p+1)*25,99)-(my$p*25)
        do k = 1,ub$1
S3        Z(k,i) = F(Z(k+5,i))
        enddo
      end
      SUBROUTINE F1$col(Z,i)
        REAL Z(100,25)
        do k = 1,95
S4        Z(k,i) = F(Z(k+5,i))
        enddo
      end
```

Figure 7.7 Interprocedural Fortran D Compiler Output

instantiated by modifying the text of the program to insert *send* and *recv* routines or collective communication primitives.

To see how this strategy works, first recall from Chapter 4 that message vectorization uses the level of the *deepest* loop-carried true dependence to combine messages at outer loop levels [16, 212]. Communication for loop-carried dependences is inserted at the beginning of the loop that carries the dependence. Communication for loop-independent dependences is inserted in the body of the loop enclosing both the source and sink of the dependence. If both loop-carried and loop-independent dependences exist at the same level, the loop-independent dependence takes priority.

Because the program is compiled in reverse topological order, local dependence analysis augmented with interprocedural RSDs representing array uses and definitions can precisely detect all loop-independent dependences and dependences carried by loops *within* the procedure, but not all dependences carried on loops *outside* the procedure. This imprecision is not a problem since the Fortran D compiler delays instantiation of communication for nonlocal references in any case to take advantage of additional opportunities to apply message vectorization, coalescing, aggregation, and other communication optimizations.

For interprocedural compilation, the Fortran D compiler first performs interprocedural dependence analysis. References within a procedure are put into RSD form, but merged only if no loss of precision will result. The resulting RSDs may be propagated to calling procedures and translated as definitions or uses to actual parameters and global variables [98]. During code generation, the Fortran D compiler uses intraprocedural algorithms to calculate nonlocal index sets, using the deepest true dependence to determine the loop level for vectorizing communication. If a nonlocal reference is the sink of a true dependence carried by a loop in the current procedure, communication must be generated within the procedure. Otherwise the nonlocal index set is marked and passed to the calling procedure, where its level and location may be determined more accurately and optimizations applied. The algorithm for optimizing communication is shown in Figure 7.8.

Communication Optimization Example. We illustrate the analysis and optimization techniques used to generate communication for Figure 7.7. First, the Fortran D compiler uses the local iteration sets calculated for statements S_3 and S_4 to determine the nonlocal index sets for the *rhs* $Z(k+5,i)$. In procedure F1$col, the local iteration set [1:95,1:25] yields the accesses [6:100,1:25]. Since the local index set for Z is [1:100,1:25], all accesses are local and no communication is required.

```
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   for each rhs reference V in P do
      compare with lhs to determine type of communication
      if communication is needed then
         use dependence information to calculate commlevel
         build RSD representing data to be communicated
         insert RSD at loop at commlevel if local
      endif
   endfor
   for each call site in P do
      insert RSDs from call at commlevel if local to P
   endfor
   for each loop in P do
      merge RSDs at loop if no precision is lost
      aggregate RSDs for messages to the same processor
   endfor
   instantiate communication for RSDs at local loops
   collect remaining RSDs for callers
endfor
```

Figure 7.8 Communication Analysis and Optimization

In procedure F1$row, the local iteration set [1:25,1:100] yields the accesses [6:30,1:100]. Subtracting the local index set produces the nonlocal index set [26:30,1:100]. The compiler determines that communication does not need to generated locally because $Z(k+5,i)$ has no true dependences carried by the local k loop. Instead, it computes the nonlocal index set [26:30,i] for Z and saves it for use when compiling the caller.

When compiling P1, the Fortran D compiler translates the nonlocal index set for Z into a reference to X, the actual parameter for the call to procedure F1$row at S_1. Interprocedural dependence analysis based on RSDs shows that it has no true dependence carried on the i loop either. The compiler thus vectorizes the message outside the i loop, resulting in the nonlocal index set [26:30,1:100]. Guarded messages are generated to communicate this data between processors.

7.3.6 Optimization vs. Language Extensions

An important point demonstrated in the previous sections is how delayed instantiation of the computation partition and communication is key to interprocedural optimization. For instance, consider the code generated for Figure 7.1 if the compiler cannot delay instantiation across procedure boundaries, but must immediately instantiate both the computation and communication partition. For simplicity, again assume that procedure F1 contains the k loop. When compiling F1$row, the Fortran D compiler would need to insert messages inside the procedure to communicate nonlocal data accessed. This code would result in a hundred messages for X[26:30,i], one for each invocation of F1$row, rather than a single message for X[26:30,1:100] in P1. In addition, the compiler would need to introduce explicit guards in F1$col to partition the computation, rather than simply reducing the bounds of the j loop in P1. The resulting program, shown in Figure 7.9, is much less efficient than the code in Figure 7.7.

This example also points out limitations for language extensions designed to avoid interprocedural analysis. Language features such as *interface blocks* [196] require the user to specify information at procedure boundaries. These features impose additional burdens on the programmer, but can reduce or eliminate the need for interprocedural analysis. However, current language extensions are insufficient for interprocedural optimizations. This may significantly impact performance for certain computations, as we show in Section 7.7.

```
    PROGRAM P1
      REAL X(30,100), Y(100,25)
      do i = 1,100
S1      call F1$row(X,i)
      enddo
      do j = 1,100
S2      call F1$col(Y,j)
      enddo
    end
    SUBROUTINE F1$row(Z,i)
      REAL Z(30,100)
      my$p = myproc() {* 0...3 *}
      if (my$p .GT. 0) send X(1:5,i) to my$p-1
      if (my$p .LT. 3) recv X(26:30,i) from my$p+1
      ub$1 = min((my$p+1)*25,99)-(my$p*25)
      do k = 1,ub$1
S3      Z(k,i) = F(Z(k+5,i))
      enddo
    end
    SUBROUTINE F1$col(Z,i)
      REAL Z(100,25)
      if ((i .GT. 0) .AND. (i .LT. 25)) then
        do k = 1,95
S4        Z(k,i) = F(Z(k+5,i))
        enddo
      endif
    end
```

Figure 7.9 Program with Immediate Instantiation

7.3.7 Overlap Calculation

The Fortran D compiler uses overlaps and buffers to store nonlocal data fetched from other processors. The number and sizes of temporary buffers required may be propagated up the call graph during code generation as each procedure is compiled. At the top level, the total number and size of buffers is known and can be allocated. Calculating the overlap regions needed for each array is more difficult. The problem is that multidimensional arrays must be declared to have consistent sizes in all but the last dimension, or else inadvertent array reshaping will result. Since using overlaps changes the size of array dimensions, the size of an overlap region must be the same across all procedures. This restriction prevents the use of any single-pass algorithms.

A simple algorithm can compile all procedures and record overlaps used, then perform a second pass over procedures in order to make overlap declarations uniform. To eliminate a second pass over the program, the Fortran D compiler tries to estimate the number and sizes of overlaps by storing constant offsets that appear in array variable subscripts during local analysis. These offsets are propagated in the interprocedural analysis phase to estimate the maximal overlaps needed for each array. Code generation then determines what overlaps are actually needed. The estimate may be updated incrementally if it has not been used in previously compiled procedures. Otherwise the compiler may choose to either utilize buffers or go back and modify array declarations in those procedures. The algorithm for calculating overlaps is described in Figure 7.10.

Overlap Example. For instance, the overlaps required for X and Y in Figure 7.7 are calculated as follows. In the local analysis phase, the reference $Z(k+5,i)$ results in the overlap offset Z($\{+5\}$,0). Interprocedural propagation of overlap offsets translates these offsets for the formal parameter Z to the actual parameters X and Y, discovering that this is the maximum offset for both arrays. Using the results of reaching decomposition analysis, the compiler determines that the first dimension of X and the second dimension of Y are distributed. The overlap offset ($\{+5\}$,0) yields for X the estimated overlap region [26:30,100]. No overlap is needed for Y since the offset in the distributed dimension is zero. During code generation these overlaps are discovered to be both necessary and sufficient.

```
{* Local analysis phase *}
for each procedure P do
  for each array reference R to variable V do
    mark overlap offset in each dimension
  endfor
endfor
{* Interprocedural propagation phase *}
calculate reaching decompositions
for each procedure P do (in reverse topological order)
  for each array variable V in P do
    merge local overlap offsets and those from calls
    propagate overlap offset to callers
  endfor
endfor
propagate resulting overlap offset estimates down ACG
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
  for each array variable V in P do
    determine actual overlap needed for V
    if actual overlap is greater than estimated then
      use buffer instead, or modify previous procedures
    endif
    mark overlap estimate as used by P
  endfor
  instantiate local overlaps
  collect actual overlap offsets for callers
endfor
```

Figure 7.10 Overlap Calculation Algorithm

```
PROGRAM P1              SUBROUTINE F1(X,Xlo,Xhi)
  REAL X(30)              REAL X(Xlo:Xhi)
  call F1(X,1,30)         do i = 1,25
end                         X(i) = F(X(i+5))
                          enddo
                        end
```

Figure 7.11 Parameterized Overlaps

Overlap Alternatives. The overlap estimation algorithm is not very precise, but unfortunately is hard to improve without significantly more effort during local analysis. Empirical results will be needed to establish its accuracy in practice. The difficulty posed by overlaps may motivate other storage methods altogether. When analysis is known to be imprecise, the Fortran D compiler may choose to store nonlocal data in buffers instead of overlaps. Using buffers requires additional work by the compiler to separate loop iterations accessing nonlocal data, but this is necessary in any case to perform *iteration reordering*, a communication optimization designed to overlap communication with computation. If the overlap region is noncontiguous, using buffers also has the advantage of eliminating the need to unpack nonlocal data.

Alternatively, the Fortran D compiler can rely on Fortran's ability to specify array dimensions at run time. By adding additional arguments to a procedure, the compiler can produce *parameterized overlaps* for array parameters. Since the extent of all overlaps are known after compiling the main program, they may simply be specified as compile-time constants and passed as arguments to procedures. For instance, Figure 7.11 shows how parameterized overlaps may be generated for the program in Figure 7.1. Unfortunately only overlaps for array formal parameters may be parameterized. Overlaps for global arrays found in Fortran common blocks must be determined statically at compile time using the algorithm previously described.

7.4 Optimizing Dynamic Data Decomposition

As stated previously, users can dynamically change data decompositions in Fortran D. This feature is desirable because phases of a computation may require different data decompositions to reduce data movement or load imbalance. Fortran D assumes the existence of a collection of library routines that can be invoked to remap arrays for different data decompositions. It is the task of the compiler to determine where calls to these mapping routines must be inserted to map affected arrays when executable ALIGN and DISTRIBUTE statements are encountered.

We show that straightforward placement of mapping routines may produce highly inefficient code. In comparison, an interprocedural approach can yield significant improvements. Additional language support is insufficient, because optimization must be performed across procedure boundaries. As with communication and partitioning optimizations, the key to enabling interprocedural optimization is delayed instantiation of dynamic data decomposition. In other words, the Fortran D compiler waits to insert data mapping routines in the callers rather than in the callee.

7.4.1 Live Decompositions

Because the cost of remapping data can be very high, we would like to recognize and eliminate unnecessary remapping where possible. For instance, consider the calls to procedure F1 at S_1 and S_2 in Figure 7.12. Array X is originally distributed block-wise, but is redistributed cyclically in F1. If no optimizations are performed, the compiler inserts mapping routines before each call to F1, as displayed in Figure 7.13a. This code causes array X to be mapped four times for each iteration of loop k. The same problems result if delayed instantiation is not used, because calls to mapping routines are inserted in F1 instead of P1. Analysis can show that the mapping routine for X at S_5 is *dead*, because X is not referenced before it is remapped at S_6. A more efficient version of the program would map array X just twice, before and after the calls to F1, as in Figure 7.13b.

We pose a new flow-sensitive data-flow problem to detect and eliminate such redundant mappings. We define *live decompositions* to be the set of data decomposition specifications that may reach some array reference aligned with the decomposition. The Fortran D compiler treats each ALIGN or DISTRIBUTE statement as a number of *definitions*, one for each array affected by the statement. A reference to one of these arrays constitutes a *use* of the definition for that array. With this model, the Fortran D compiler can calculate live decompositions in the same manner as *live variables* [4]. Array mapping calls that are not live may be eliminated.

One approach would be to calculate live decompositions during interprocedural propagation. During local analysis, we would collect summary information representing control flow and the placement of data decomposition specifications. We would then need to compute the solution on the *supergraph* formed by combining local control flow graphs with the call graph, taking care to avoid paths that do not correspond to possible execution sequences [158]. To avoid this complexity, we choose instead to compute live decompositions during code generation, when control flow information is available.

Live Decompositions Calculation. Interprocedural live variable analysis has been proven Co-NP-complete in the presence of aliasing [158]. Even without aliasing, interprocedural live variable analysis can be expensive since it requires bidirectional propagation, causing a procedure to be analyzed multiple times. We rely on two restrictions to make the live decompositions problem tractable for the Fortran D compiler. First, the scope of dynamic data decomposition is limited to the current procedure and its descendants. Second, Fortran D disallows dynamic data decomposition for aliased variables, as discussed in Section 7.4.4.

By inserting mapping routines in the callers rather than in the callee, we can solve live decompositions in one pass by compiling in reverse topological order during the interprocedural code generation phase. The key insight is that due to Fortran D scoping rules, we know all local dynamic data decompositions are dead at procedure exit. To determine whether they are live within a procedure, we only need information about the procedure's descendants. The compiler cannot determine locally whether calls to mapping routines to restore inherited data decompositions are live, but these mapping calls may be collected and passed to the callers. By delaying their instantiation, we eliminate the need for information about the procedure's callers.

```
      PROGRAM P1                SUBROUTINE F1(X)
        REAL X(100)               REAL X(100)
        DISTRIBUTE X(BLOCK)       DISTRIBUTE X(CYCLIC)
        do k = 1,T                ... = X(...)
S1        call F1(X)            end
S2        call F1(X)            SUBROUTINE F2(X)
        enddo                     REAL X(100)
        call F2(X)           S3   X(...) = ...
      end                       end
```

Figure 7.12 Dynamic Data Decomposition Example

```
      {* No Optimization *}           {* Live Decompositions *}
      do k = 1,T                      do k = 1,T
S4      map-block-to-cyclic(X)   S8     map-block-to-cyclic(X)
        call F1(X)                      call F1(X)
S5      map-cyclic-to-block(X)          call F1(X)
S6      map-block-to-cyclic(X)   S9     map-cyclic-to-block(X)
        call F1(X)                    enddo
S7      map-cyclic-to-block(X)        call F2(X)
      enddo
      call F2(X)

            (7.13a)                         (7.13b)

      {* Loop-invariant Decomps *}    {* Array Kills *}
      map-block-to-cyclic(X)          map-block-to-cyclic(X)
      do k = 1,T                      do k = 1,T
        call F1(X)                      call F1(X)
        call F1(X)                      call F1(X)
      enddo                           enddo
      map-cyclic-to-block(X)          mark-as-block(X)
      call F2(X)                      call F2(X)

            (7.13c)                         (7.13d)
```

Figure 7.13 Dynamic Data Decomposition Optimizations

The basic live decompositions algorithm works as follows. We calculate during code generation the following summary sets for each procedure:

- DECOMPUSE(P) = $\{ X \mid X \in$ APPEAR(P) and may use some decomposition reaching P $\}$
- DECOMPKILL(P) = $\{ X \mid X \in$ APPEAR(P) and must be dynamically remapped when P is invoked $\}$
- DECOMPBEFORE(P) = $\{ \langle D, X\rangle \mid X \in$ APPEAR(P) and must be mapped to decomposition D before P $\}$
- DECOMPAFTER(P) = $\{ \langle D, X\rangle \mid X \in$ APPEAR(P) and must be mapped to decomposition D after P $\}$

DECOMPUSE and DECOMPKILL are calculated through local data-flow analysis. They provide interprocedural information for computing live decompositions. DECOMPBEFORE consists of all variables X that need to be mapped before invoking P. DECOMPAFTER consists of all variables X that are mapped in P to some new decomposition, and thus must be remapped when returning from P. Together DECOMPBEFORE and DECOMPAFTER represent dynamic data decompositions from P whose instantiation have been delayed.

We calculate live decompositions by simply propagating uses backwards through the local control flow graph for each procedure [4]. A data decomposition statement is live with respect to a variable X only if there is some path between it and a reference to X that is not killed by another decomposition statement or by DECOMPKILL of an intervening call. Summary sets describe the effect of each procedure call encountered. Formal parameters of P in DECOMPUSE and DECOMPKILL are translated and treated as references to actual parameters. DECOMPBEFORE and DECOMPAFTER are translated and treated as decompositions affecting variables in P. Decompositions that are dead may be removed. In addition, we can *coalesce* live decompositions if they are identical and their live ranges overlap. All live decompositions except the first may then be eliminated. The live decomposition algorithm is presented in Figure 7.14.

Live Decompositions Example. Consider how live decompositions are calculated in Figure 7.12. The Fortran D compiler proceeds in reverse topological order, so we begin with either F1 or F2. For procedure F1, local live and reaching decomposition analysis shows that no incoming decompositions are used. The local redistribution of X to *cyclic* kills the incoming decomposition for X, and requires that X be distributed to *cyclic* before F1 and back to *block* after F1. Since there are no local data decompositions for F2, the incoming decomposition is used for the reference to X. No decompositions are killed in F2 or needed before or after F2. The resulting information is produced:

$$\begin{aligned}
\textsc{DecompUse}(\text{F1}) &= \emptyset \\
\textsc{DecompKill}(\text{F1}) &= \{\ X\ \} \\
\textsc{DecompBefore}(\text{F1}) &= \{\ \langle (cyclic), X\rangle\ \} \\
\textsc{DecompAfter}(\text{F1}) &= \{\ \langle (block), X\rangle\ \} \\
\textsc{DecompUse}(\text{F2}) &= \{\ X\ \} \\
\textsc{DecompKill}(\text{F2}) &= \emptyset \\
\textsc{DecompBefore}(\text{F2}) &= \emptyset \\
\textsc{DecompAfter}(\text{F2}) &= \emptyset
\end{aligned}$$

When we compile the main program body P1, we translate all summary sets in terms of local variables. The DecompBefore and DecompAfter sets correspond to potential calls to mapping routines, equivalent to the program shown in Figure 7.13a. Local live decomposition analysis discovers that there are no uses of the *block* decomposition for X at S_5, allowing it to be eliminated. Local reaching decomposition analysis can then determine that the *cyclic* decompositions for X at S_4 and S_6 are identical. They may then be coalesced, eliminating S_6 to achieve the program shown in Figure 7.13b.

7.4.2 Loop-invariant Decompositions

In addition to eliminating non-live decompositions and coalescing identical live decompositions, we can also hoist loop-invariant decompositions out of loops to reduce remapping. For instance, consider the mapping routines remaining in Figure 7.13b. If we can hoist the mapping routines, each remapping then occurs once rather than on each iteration of the loop. There are two situations where a decomposition that is live and loop-invariant with respect to variable X may be hoisted out of a loop. They vary slightly from the requirements for loop-invariant code motion [4]:

- If the decomposition is not used within the loop for X, it may be moved after the loop. We verify this condition by comparing LocalReaching and DecompUse for all statements in the loop.
- If the decomposition is the only one used within the loop for X, it may be moved prior to the loop. We verify this condition by checking that no other decompositions reach any occurrences of X.

In the program in Figure 7.13b, the mapping routine at S_9 is not used within the loop and can be moved after the loop. Now the mapping routine at S_8 is the only decomposition reaching all references to X in the loop, so it can be hoisted to a point preceding the loop, producing the desired program shown in Figure 7.13c.

7.4.3 Array Kills

Array kill analysis may be used to determine when the values of an array are *live*. An array whose values are not live does not need to be remapped by physically copying values between processors. Instead, it may be remapped in place by simply marking it as possessing the new decomposition. For instance, suppose that array kill analysis determines that statement S_3 in Figure 7.12 kills all values in array X. We can then eliminate the cyclic-to-block mapping routine preceding the call to F2, notifying the run-time system instead if necessary. This optimization results in the program shown in Figure 7.13d.

```
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   for each call site in P do
      Translate DecompAfter, DecompBefore,
      DecompUse, DecompKill to actual parameters
   endfor
   calculate local live decompositions
   eliminate dead decompositions
   coalesce identical decompositions
   for each variable X ∈ Appear(P) do
      if original decomposition may reach X then
         add X to DecompUse
      if X must be assigned a decomposition then
         add X to DecompKill
      if X is assigned a decomposition D before
            it uses its inherited decomposition then
         add ⟨D, X⟩ to DecompBefore
      if X is locally assigned a decomposition D that
         differs from the inherited decomposition D′ then
         add ⟨D′, X⟩ to DecompAfter
   endfor
endfor
```

Figure 7.14 Live Decompositions Algorithm

7.4.4 Aliasing

Two variables X and Y are *aliased* at some point in the program if X and Y may refer to the same memory location [20]. In Fortran 77, aliases arise through parameter passing, either between reference parameters of a procedure if the same memory location is passed to both formals, or between a global and formal to which it is passed.

Aliasing affects dynamic data decomposition because a variable may be remapped indirectly through one of its aliases. Unfortunately, precise alias analysis is computationally intractable [158]. As a result, the compiler cannot efficiently prove that a decomposition that has been applied to a variable holds for a possible alias. The compiler would have to evaluate reaching decompositions for a variable and all of its potential aliases, reverting to run-time resolution if multiple decompositions reach an access to the variable.

To eliminate the efficiency problems and avoid certain confusing program semantics associated with aliasing, Fortran D requires that a variable and its alias cannot have different reaching decompositions that are live at the same point in the program. This requirement is similar to the specification in the Fortran 77 standard that makes it illegal to write to aliased variables. As a result, the compiler can ignore aliasing when analyzing decompositions since it is illegal to construct a program where remapping a variable's alias changes the decomposition reaching an access to the variable.

Since it is possible to construct a syntactically correct but illegal program, the compiler should warn the programmer of situations where aliasing might cause undefined behavior. We can test the reaching decompositions for each possible alias of a variable at a decomposition statement, warning the programmer if the alias has a different decomposition that is live. Only a warning is produced since the imprecision of alias and live analysis may signal problems in a legal program.

7.5 Interprocedural Compilation Algorithm

The full interprocedural Fortran D compilation algorithm is shown in Figure 7.15. It integrates Fortran D compilation techniques with the interprocedural analysis and optimization framework of ParaScope.

```
{* Local analysis phase *}
for each procedure P do
   calculate information for: augmented call graph,
      scalar and array side effects, symbolics,
      reaching decompositions, overlap offsets
endfor
{* Interprocedural propagation phase *}
construct call graph, augment with loop information,
calculate aliasing, symbolics, scalar and array
   side effects, reaching decompositions, cloning,
   overlap offsets
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   translate information from call sites in P
   update local loop and subscript information
   perform scalar data-flow analysis, symbolic analysis,
         dependence testing, variable classification
   partition data and computation
   analyze and optimize communication
   calculate number, size, and type of overlaps & buffers
   calculate live, loop-invariant decompositions
   generate code, collect information for callers
endfor
```

Figure 7.15 Interprocedural Compilation of Fortran D

7.6 Recompilation Analysis

The Fortran D compiler will follow the ParaScope approach for limiting recompilation in the presence of interprocedural optimization [32, 63]. Recompilation analysis is used to limit recompilation of a program following changes, an important component to maintaining the advantages of separate compilation. Briefly stated, modules only need to be recompiled if they have been edited or if they have been optimized using interprocedural information that is no longer valid.

To determine whether recompilation is needed, the compiler records the interprocedural information used by a compilation. In subsequent compilations, it compares interprocedural information used in the previous compilation with what has been computed in the current compilation. The Fortran D compiler needs to record scalar data-flow analysis results and array side-effects, as well as reaching and live decompositions, overlap offsets, local iteration sets, and nonlocal index sets. The complete list of problems is shown in Table 7.1 in Section 7.3.

Recompilation mimics the interprocedural compilation algorithm presented in Figure 7.15. Local analysis is applied to edited procedures, then interprocedural propagation is performed. Following an initial test to discover which modules have been edited since the previous compilation, we apply *recompilation tests* to interprocedural data-flow information for each module and its call sites. The compiler must also ensure that cloning applied to expose reaching decompositions is still valid; it may decide to form more clones at this time. Alternatively, changes in interprocedural information may make some clones obsolete, causing their retraction. As soon as one recompilation test fails, the module is marked as needing recompilation.

In the bottom-up pass over the program, if the current node has not been marked for recompilation, the compiler applies recompilation tests on the iteration sets, nonlocal index sets and RSDs at each call site. Depending on the results, some procedures are marked for recompilation. If the current procedure has been marked, it is compiled in the usual manner, producing new interprocedural information to be tested.

7.6.1 Recompilation Tests

Recompilation tests ensure that interprocedural information used to compile a procedure conservatively approximates the current information. A simple test just verifies that the old information is equal to the new information. However, safe tests that generate less recompilation are possible if we consider how the information will be used. Improved recompilation tests for many scalar data-flow problems are described by Burke and Torczon [32]. To give the flavor of the recompilation tests, we describe the test for reaching decompositions. Let *oldP* be the representation of P from the previous compilation. The procedures needing recompilation are those for which the following is true:

$$\mathit{Filter}(\text{REACHING}(\mathit{oldP}), \text{APPEAR}(P)) \neq \mathit{Filter}(\text{REACHING}(P), \text{APPEAR}(P))$$

Filter and APPEAR are described in Section 7.3.2. They are used to determine whether differences in reaching decompositions actually affect optimization. The test thus marks a procedure P for recompilation only if the decomposition reaching a variable appearing in P or its descendants changes.

Recompilation tests for other Fortran D interprocedural data-flow problems are simpler. Callers must be recompiled if the local iteration or nonlocal index sets of a procedure have changed, since the callers' guards, loop bounds, or communication may be affected. Similarly, modifications to live or loop-invariant decomposition information requires recompilation of the caller. Changes in array section analysis may affect array kill information, requiring recompilation if array remapping routines were affected in the caller. If overlap offsets for a procedure change but do not exceed the original assigned overlaps, recompilation is not necessary. However, if the new overlap offset is greater than the overlap allocated during code generation, every procedure referencing the array will need to be recompiled to reflect the new overlap offset, not just the callers.

A little more work is needed to calculate the extent of recompilation in the presence of cloning based on reaching decompositions [32, 92]. The compiler maintains a mapping from procedures in the call graph to the list of compiled clones for that procedure. For a procedure that has been cloned, the recompilation test can be applied to all the clones in order to find a match for the procedure. It must also pass recompilation tests for other interprocedural problems.

7.7 Empirical Results

7.7.1 Compilation Strategies for DGEFA

This section demonstrates the effectiveness of interprocedural optimization using the routine DGEFA from Linpack, a linear algebra library [67]. DGEFA is also a major component in Linpackd, the Linpack Benchmark Program. DGEFA uses Gaussian elimination with partial pivoting to factor a double-precision floating-point array. A simplified version is shown in Figure 7.16. DGEFA relies on three other Linpack routines: IDAMAX, DSCAL, and DAXPY. Since arrays are stored in *column-major* order in Fortran, DGEFA performs operations column-wise to provide data locality.

To reduce both communication and load imbalance, we choose a column-wise cyclic distribution of array A. We focus on DAXPY because it performs the majority of the computation. Because the techniques discussed in this paper have not yet been implemented in the Fortran D compiler, we applied them by hand, generating three versions of the program. In the *run-time resolution* version shown in Figure 7.17, lack of decomposition information implies that processors must determine ownership and communication for individual array elements. In the *interprocedural analysis* program displayed in Figure 7.18, we assume that reaching decomposition information is provided for DAXPY through analysis or language extensions. This information allows us to vectorize messages inside the procedure.

Finally, in the version created by *interprocedural optimization*, interprocedural array section analysis can determine that DAXPY reads a column of A starting at $A(k+1,k)$ and defines a column of A starting at $A(k+1,j)$. Dependence analysis discovers that the two columns never intersect, since $k < j \leq n$, proving that no true dependences are carried by the j loop. Message vectorization can then insert communication outside the j loop altogether, avoiding redundant communication. In addition, we utilize *broadcast* rather than *send*, since the same column is required by all processors. The resulting program is shown in Figure 7.19.

```
{* Gaussian Elimination w/ Partial Pivoting *}
SUBROUTINE DGEFA(n,a,IPVT)
 INTEGER n,IPVT(n),j,k,l
 DOUBLE PRECISION A(n,n), t
 do k = 1, n-1
  l = IDAMAX(n-k+1,A(k,k),1) + k - 1
  IPVT(k) = l
  if (l .NE. k) then
   t = A(l,k)
   A(l,k) = A(k,k)
   A(k,k) = t
  endif
  t = -1.0d0/A(k,k)
  call DSCAL(n-k,t,A(k+1,k))
  do j = k+1, n
   t = A(l,j)
   if (l .NE. k) then
    A(l,j) = A(k,j)
    A(k,j) = t
   endif
   call DAXPY(n-k,t,A(k+1,k),A(k+1,j))
  enddo
 enddo
 IPVT(n) = n
end
```

```
{* Find Maximum Element in Vector *}
INTEGER FUNCTION IDAMAX(n,DX)
 DOUBLE PRECISION DX(n),dmax
 INTEGER i,ix,n
 dmax = DABS(DX(1))
 do i = 2,n
  if (DABS(DX(i)) .GT. dmax) then
   idamax = i
   dmax = DABS(DX(i))
  endif
 enddo
end

{* Scale a Vector by a Constant *}
SUBROUTINE DSCAL(n,da,DX)
 DOUBLE PRECISION da,DX(n)
 INTEGER i,n
 do i = 1,n
  DX(i) = da*DX(i)
 enddo
end

{* Constant times Vector plus Vector *}
SUBROUTINE DAXPY(n,da,DX,DY)
 DOUBLE PRECISION DX(n),DY(n),da
 INTEGER i,n
 do i = 1,n
  DY(i) = DY(i) + da*DX(i)
 enddo
end
```

Figure 7.16 Simplified Sequential Version of DGEFA

```
SUBROUTINE DAXPY(n,da,DX,DY)
  do i = 1,n
    if (own(DX(i)) .AND. .NOT. own(DY(i))) then
      send DX(i) to owner(DY(i)))
    endif
    if (own(DY(i)) .AND. .NOT. own(DX(i))) then
      recv DX(i) from owner(DX(i)))
    endif
    if (own(DY(i))) then
      DY(i) = DY(i) + da*DX(i)
    endif
  enddo
end
```

Figure 7.17 DGEFA: Run-time Resolution

```
SUBROUTINE DAXPY(n,da,DX,DY)
  if (own(DX(1)) .AND. .NOT. own(DY(1))) then
    send DX(1:n) to owner(DY(1)))
  endif
  if (own(DY(1)) .AND. .NOT. own(DX(1))) then
    recv DX(1:n) from owner(DX(1)))
  endif
  if (own(DY(1))) then
    do i = 1,n
      DY(i) = DY(i) + da*DX(i)
    enddo
  endif
end
```

Figure 7.18 DGEFA: Interprocedural Analysis

```
SUBROUTINE DGEFA(n,a,IPVT)
  do k = 1, n-1
    if (own(A(k+1,k))) then
      broadcast A(k+1:n,k)
    else
      recv A(k+1:n,k) from owner(A(k+1,k))
    endif
    do j = k+1, n
      call DAXPY(n-k,t,A(k+1,k),A(k+1,j))
    enddo
  enddo
end
SUBROUTINE DAXPY(n,da,DX,DY)
  do i = 1,n
    DY(i) = DY(i) + da*DX(i)
  enddo
end
```

Figure 7.19 DGEFA: Interprocedural Optimization

7.7.2 Measured Execution Times

For our measurements we used a 32 node Intel iPSC/860 with 8 Meg of memory per node. Each program was compiled under -O4 using Release 2.0 of *if77*, the iPSC/860 compiler. We timed the program for several problem sizes and numbers of processors using *dclock*().

Results are tabulated in Table 7.2. Execution time is presented in seconds. We define speedup in the table as follows, given parallel execution time T_{par} and sequential execution time T_{seq}. If $T_{par} < T_{seq}$, speedup is T_{seq}/T_{par}. Otherwise speedup is calculated as $-T_{par}/T_{seq}$. In some cases programs using run-time resolution sent more messages than could be handled by the iPSC/860, causing the program to deadlock. These programs are marked with "$*$".

These timings are also shown in Figure 7.20. Each graph represents a single problem size. Execution times are plotted logarithmically along the Y-axis in seconds. The number of processors is plotted along the X-axis. Results for perfect or ideal speedup are included for purposes of comparison.

We make several observations. First, run-time resolution produces code that is over a hundred times slower than the sequential program. Its performance is not affected by problem size, and degrades as the number of processors increases. Run-time resolution is thus too expensive to employ except for very infrequently executed sections of the program.

Surprisingly, the code produced with interprocedural analysis is five to ten times more expensive than the sequential program, worsening as the number of processors increases. Unlike run-time resolution, its performance improves for larger problem sizes. However, even for an 800×800 array, approximately the largest double-precision array possible on a single processor, the resulting code is still five times slower than the equivalent sequential program. Only interprocedural optimization produces positive speedups. After interprocedural optimization we observe a speedup of 8 on 32 processors. Further speedup is limited by the small problem sizes.

Problem Size	P	Run-time Resolution *time*	Run-time Resolution *speedup*	Interprocedural Analysis *time*	Interprocedural Analysis *speedup*	Interprocedural Optimization *time*	Interprocedural Optimization *speedup*
	1	*sequential time* = 0.84 seconds					
200	2	125	−149	8.7	−10.4	.57	1.47
×	4	104	−124	7.8	−9.3	.45	1.87
200	8	129	−154	9.0	−10.7	.43	1.95
	16	144	−171	9.6	−11.4	.46	1.83
	32	152	−181	9.4	−11.2	.52	1.62
	1	*sequential time* = 7.16 seconds					
400	2	1050	−147	53	−7.4	4.0	1.79
×	4	841	−117	54	−7.5	2.6	2.75
400	8	1047	−146	63	−8.8	2.0	3.58
	16	1177	−164	67	−9.4	1.8	3.98
	32	1256	−175	68	−9.5	1.9	3.77
	1	*sequential time* = 64.5 seconds					
800	2	8394	−130	358	−5.6	32.8	1.97
×	4	*	*	400	−6.2	18.4	3.51
800	8	*	*	467	−7.2	11.7	5.51
	16	*	*	499	−7.7	9.0	7.17
	32	*	*	511	−7.9	8.1	7.96

Table 7.2 Performance of DGEFA for Intel iPSC/860

Our empirical results show that interprocedural compilation can improve performance by several orders of magnitude for an important application. We do not expect interprocedural optimization to be required in all cases, but for many computations it can make a significant difference.

7.8 Discussion

This chapter shows that interprocedural compilation is needed to fully exploit the benefits of data-placement languages such as Fortran D. Efficient interprocedural analysis, optimization, and code generation techniques can be designed that require only one pass over the program. Delaying instantiation of the computation partition, communication, and dynamic data decomposition is key to improving interprocedural optimization. Recompilation analysis preserves the benefits of separate compilation. Experiments indicate that interprocedural communication optimization is essential for at least one important program. For DGEFA, a version of Gaussian elimination with partial pivoting from Linpack, extracting communication across procedure boundaries improves performance by orders of magnitude.

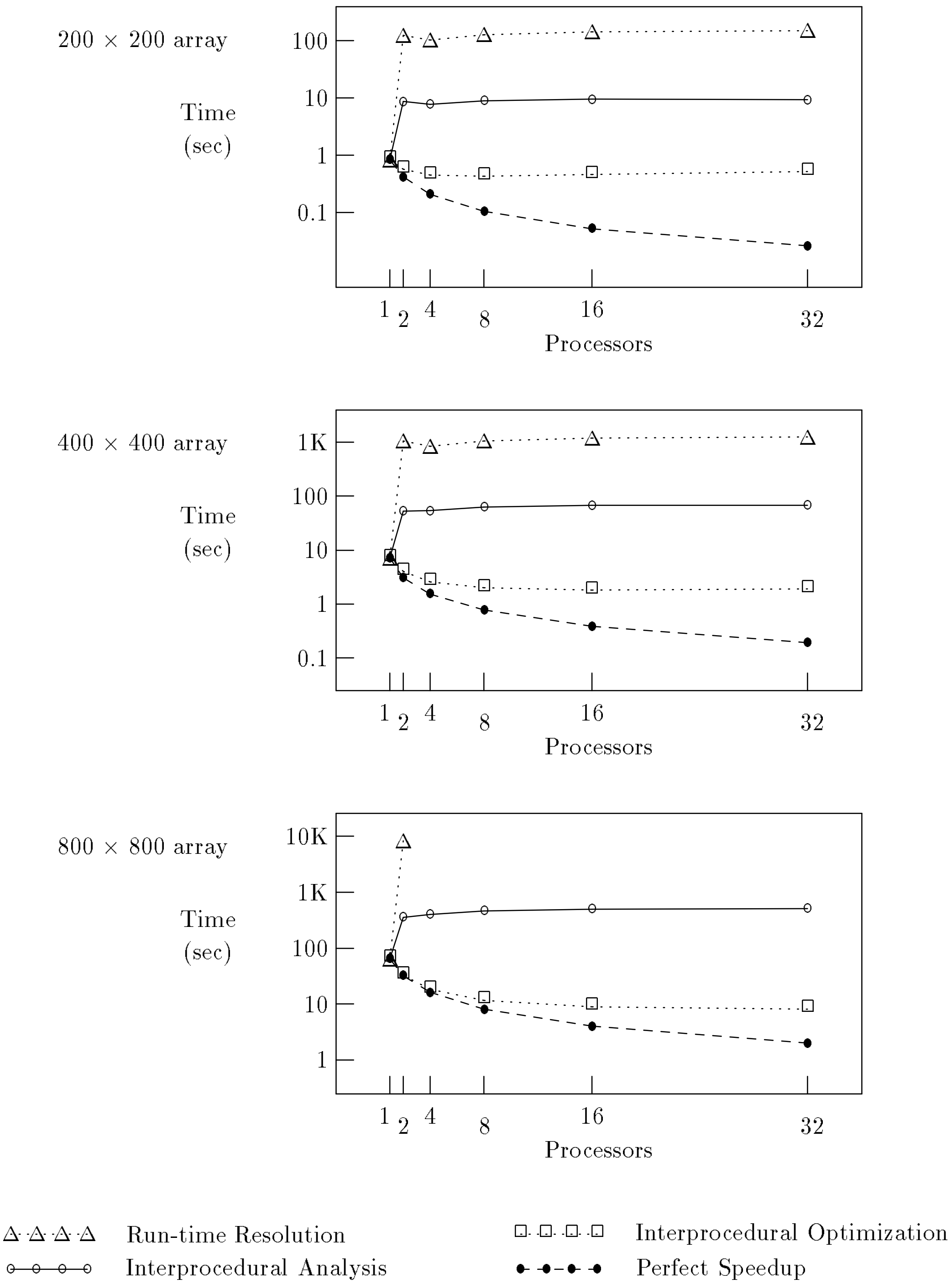


Figure 7.20 Interprocedural Optimization Results (Intel iPSC/860)

Chapter 8

Compiling Fortran 77D and 90D

We present an integrated approach to compiling Fortran 77D and Fortran 90D programs for efficient execution on MIMD distributed-memory machines. The integrated Fortran D compiler relies on two key observations. First, array constructs may be *scalarized* into FORALL loops without loss of information. Second, *loop fusion*, *partitioning*, and *sectioning* optimizations are essential for both Fortran D dialects. This chapter describes the Fortran 90D and 77D front ends and the common Fortran D back end. The design of the run-time library is discussed, and an example is used to illustrate the compilation process.

8.1 Introduction

Fortran D provides data decomposition specifications that can be applied to Fortran 77 and Fortran 90 [13] to produce Fortran 77D and Fortran 90D, respectively. In this chapter, we describe a unified strategy for compiling both Fortran 77D and Fortran 90D into efficient SPMD message-passing programs. We demonstrate how to integrate partitioning with scalarization, and show that an efficient portable run-time library can ease the task of compiling Fortran D.

8.2 Compilation Strategy

8.2.1 Overall Approach

Our approach to parallelizing Fortran D programs for distributed-memory MIMD computers is illustrated in Figure 8.1. In brief, we transform both Fortran 77D and Fortran 90D to a common intermediate form, which is then compiled to code for the individual nodes of the machine. We have several pragmatic and philosophical reasons for this strategy:

- Sharing a common back end for both the Fortran 77D and Fortran 90D avoids duplication of effort.
- Decoupling the Fortran 77D and Fortran 90D front ends allows them to become machine independent.
- Providing a common intermediate form helps us experiment with defining an efficient compiler/programmer interface for programming the nodes of a massively parallel machine.

8.2.2 Intermediate Form

To compile both dialects of Fortran D using a single back end, we must select an appropriate intermediate form. In addition to standard computation and control flow information, the intermediate form must capture three important aspects of the program:

- Data decomposition information, telling how data is aligned and distributed among processors.
- Parallelization information, telling when operations in the code are independent.
- Communication information, telling what data must be transferred between processors.

In addition, we believe that the primitive operations of the intermediate form should be relatively low-level operations that can be translated simply for single-processor execution.

We have chosen Fortran 77 with data decompositions, FORALL, and intrinsic functions to be the intermediate form for the Fortran D compiler. We show later that this form preserves all of the information available in a Fortran 90 program, but maintains the flexibility of Fortran 77. Parallelism and communication can be determined by the compiler for simple computations, and specified by the user using FORALL and intrinsic functions for complex computations.

8.2.3 Node Interface

Another topic of interest in the overall strategy is the node interface—the node program produced by the Fortran D compiler. It must be both portable and efficient. In addition, the level of the node interface should be neither so high that efficient translation to object code is impossible, nor so low that its workings are completely opaque to the user. We have selected Fortran 77 with calls to communication and run-time libraries based on Express, a collection of portable message-passing primitives [167]. Evaluating our experiences with this node interface is the first step towards defining an "optimal" level of support for programming individual nodes of a parallel machine.

8.3 Unified Compiler

The Fortran D compiler thus consists of three parts. The Fortran 90D and 77D front ends process input programs into the common intermediate form. The Fortran D back end then compiles this to the SPMD message-passing node program. The Fortran D compiler is implemented in the context of the ParaScope programming environment [35].

8.3.1 Fortran 90D Front End

The function of the Fortran 90D front end is to *scalarize* the Fortran 90D program, translating it to an equivalent Fortran 77D program. This is necessary because the underlying machine executes computations sequentially, rather than on entire arrays at once as specified in Fortran 90. For the Fortran D compiler we find it useful to view scalarization as three separate tasks:

- **Scalarizing Fortran 90 Constructs.** Many Fortran 90 features are not present in our intermediate form. They must be translated into equivalent Fortran 77D statements.
- **Fusing Loops.** Simple scalarization results in many small loop nests. Fusing these loop nests can improve the locality of data accesses, simplify partitioning, and enable other program transformations.
- **Sectioning.** Fortran 90 array operations allow the programmer to access and modify entire arrays atomically, even if the underlying machine lacks this capability. The Fortran D compiler must divide array operations into *sections* that fit the hardware of the target machine [8, 11].

We defer both loop fusion and sectioning to the Fortran D back end. Loop fusion is deferred because even hand-written Fortran 77 programs can benefit significantly [116, 150]. Sectioning is needed in the back end because FORALL loops may also be present in Fortran 77D.

We assign to the Fortran 90D front end the remaining task, scalarizing Fortran 90 constructs that have no equivalent in the Fortran 77D intermediate form. There are three principal Fortran 90 language features that must be scalarized: array constructs, WHERE statements, and intrinsic functions [13].

Array Constructs

Fortran 90 *array constructs* allow entire arrays to be manipulated atomically. Array sections may also be specified using triplet notation. This enhances the clarity and conciseness of the program, and has the advantage of making parallelism explicit. It is the responsibility of the compiler to efficiently implement array constructs for scalar machines. Previous research has shown that this is a difficult problem [8, 11].

One problem is that when Fortran 90 array constructs are used in assignment statements, the entire right-hand side (*rhs*) must be evaluated before storing the results in the left-hand side (*lhs*). If an assignment statement utilizing array constructs is translated naively without adequate analysis, *rhs* array elements would need to be stored in temporary buffers to ensure that they are not overwritten before their values are used.

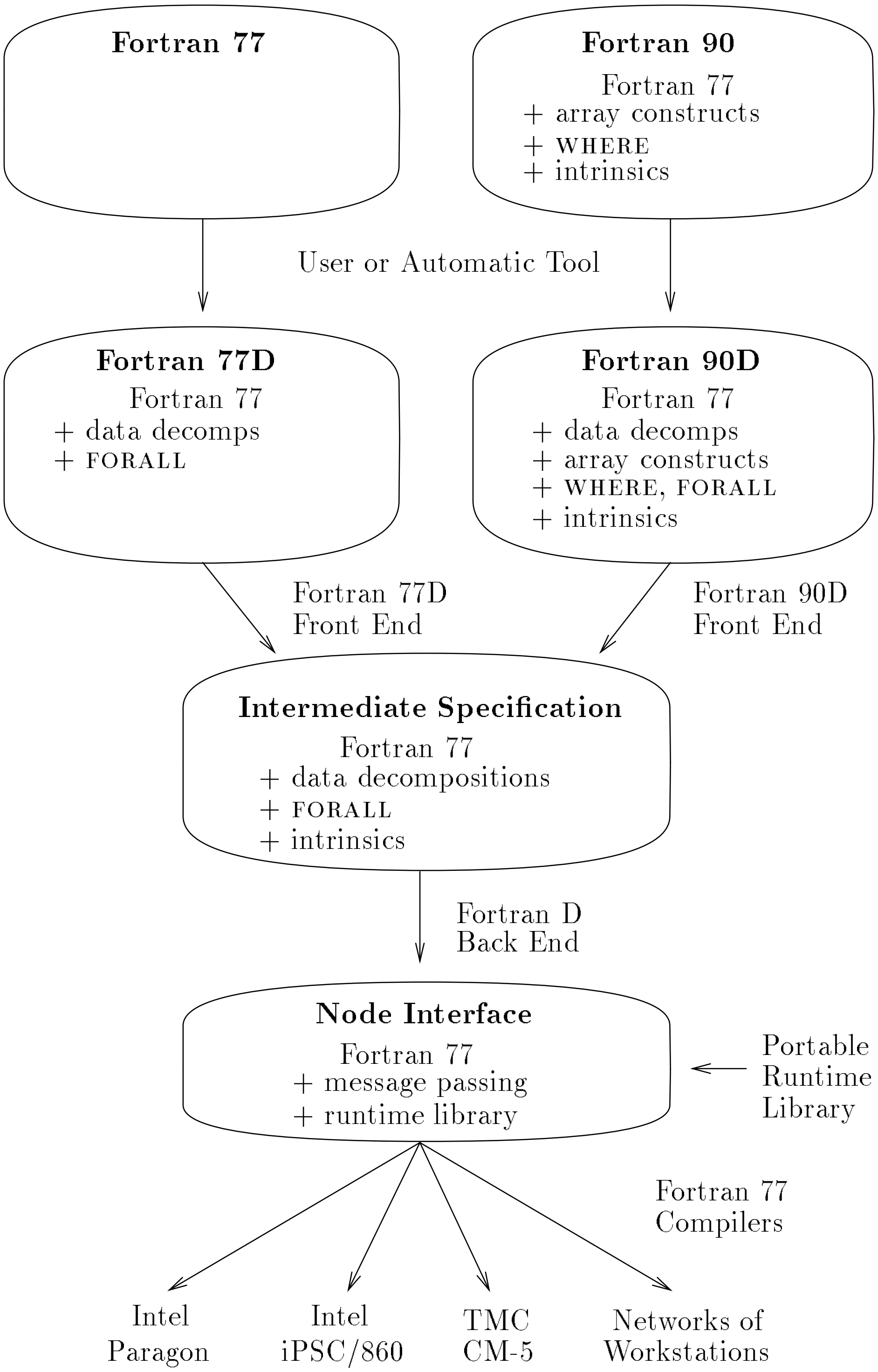


Figure 8.1 Fortran D Compilation Strategy

The Fortran 90 front end can defer this problem by relying on a key observation—the FORALL loop possesses copy-in/copy-out semantics identical to Fortran 90 assignment statements utilizing array constructs. Such statements may thus be translated into equivalent FORALL loops with no loss of information.

However, since FORALL loops specify individual element operations, indices are introduced. For simplicity, the index calculation is performed with respect to the *lhs*. In general, an array construct of the form:

$$\mathtt{A(}l_1 : u_1 : s_1\mathtt{)} = \mathtt{B(}l_2 : u_2 : s_2\mathtt{)}$$

where A and B are one dimensional arrays, is converted into:

```
forall i = l1, u1, s1
  A(i) = B(l2 + ((i - l1)/s1) * s2)
endfor
```

Of course, the expression in the *rhs* is simplified as much as possible at compile time.

WHERE Statement

Another Fortran 90 feature that has no Fortran 77 equivalent is the WHERE statement. It takes a boolean argument that is used to *mask* array operations, inhibiting assignments to array elements whose matching boolean flag has the value *false*. The boolean argument to the WHERE statement must be completely evaluated before the body of the statement may be executed.

Fortunately, the WHERE statement may be easily translated into equivalent IF and FORALL statements. Consider the following example where A is assumed to be an 1D N-element array. Because of FORALL copy-in/copy-out semantics, it is unnecessary at this point to explicitly store the value of the boolean argument to prevent it from being overwritten.

```
where (A .eq. 0)      forall i = 1,N
  A = 1.0               if (A(i) .eq. 0) then
elsewhere       ==>       A(i) = 1.0
  A = 0.0               else
endwhere                  A(i) = 0.0
                        endif
                      endfor
```

Intrinsic Functions

Intrinsic functions are fundamental to Fortran 90. They not only provide a concise means of expressing operations on arrays, but also identify parallel computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, and matrix multiplication. Additional intrinsics are described in Table 8.1. To avoid excessive complexity and machine-dependence in the Fortran D compiler, we convert most Fortran 90 intrinsics into calls to customized run-time library functions.

The strategy used by the Fortran 90D front end is thus to preserve all intrinsic functions, passing them to the Fortran D compiler back end. However, some processing is necessary. Like the WHERE statement, some intrinsic functions accept a mask expression that restricts execution of the computation. The Fortran 90D

Data Movement	Reductions	Irregular Operations	Special Routines
CSHIFT EOSHIFT SPREAD RESHAPE TRANSPOSE	DOTPRODUCT ALL, ANY, COUNT MAXVAL, MINVAL SUM, PRODUCT MAXLOC, MINLOC	PACK UNPACK	MATMUL

Table 8.1 Representative Intrinsic Functions of Fortran 90D

front end may need to evaluate the expression and store it in a temporary boolean array before performing the computation, so the mask can be passed as an argument to the run-time library.

For example, consider the following reduction operation, where X is a scalar and A, B are arrays:

```
X = MAXVAL(A, A .eq. B)
```

It should return the value of the element of A that is the maximum of all elements for which element of A is equal to the corresponding element of B. The Fortran 90D front end translates this to:

```
forall i = 1,N
  TMP(i) = A(i) .eq. B(i)
endfor
X = MAXVAL(A, TMP)
```

TMP can then be passed as an argument to the run-time routine MAXVAL. Temporary arrays may also be introduced when intrinsic functions return a value that is part of a Fortran 90 expression.

Temporary Arrays

When the Fortran 90D front end needs to create temporary arrays, it must also generate appropriate Fortran D data decomposition statements. A temporary array is usually aligned and distributed in the same manner as its master array. For example, in the previous example the temporary logical array TMP is aligned and distributed in the same manner as A and B. If A and B are distributed differently, then the temporary array is assigned the distribution of A, the first argument.

8.3.2 Fortran 77D Front End

The Fortran 77D front end does not need to perform much work since Fortran 77D is very close to the intermediate form. Its only task is to detect complex high-level parallel computations, replacing or annotating them by their equivalent Fortran 90 intrinsics. These intrinsic functions help the compiler recognize complex computations such as reductions and scans that are supported by the run-time library. With advanced program analysis, some operations such as DOTPRODUCT, SUM, TRANSPOSE, or MATMUL can be detected automatically with ease. Others computations such as COUNT or PACK may require user assistance.

8.3.3 Fortran D Back End

The Fortran D back end performs two main functions—it partitions the program onto the nodes of the parallel machine and completes the scalarization of Fortran D into Fortran 77. We find that the desired order for compilation phases is to apply loop fusion first, followed by partitioning and sectioning.

Loop fusion is performed first because it simplifies partitioning by reducing the need to consider inter-loop interactions. It also enables optimizations such as *strip-mining* and *loop interchange* [10, 205]. In addition, loop fusion does not increase the difficulty of later compiler phases. On the other hand, sectioning is performed last because it can significantly disrupt the existing program structure, increasing the difficulty of partitioning analysis and optimization.

Loop Fusion

Loop fusion is particularly important for the Fortran D back end because scalarized Fortran 90 programs present many single-statement loop nests. Fusing such loops simplifies the partitioning process and enables additional optimizations.

Data dependence is a concept developed for vectorizing and parallelizing compilers to characterize memory access patterns at compile time [10, 132, 205]. A true dependence indicates definition followed by use, while an anti-dependence shows use before definition. Data dependences may be either loop-carried or loop-independent. Loop fusion is legal if it does not reverse the direction of any data dependence between two loop nests [8, 202, 205].

The current Fortran D back end fuses all adjacent loop nests where legal, if no loop-carried true dependences are introduced. This heuristic does not adversely affect the parallelism or communication overhead of the resulting program, and should perform well for the simple cases found in practice. More sophisticated algorithms are discussed elsewhere [9, 84, 115, 150, 202].

Loop fusion also has the added advantage of being able to improve memory reuse in the resulting program. Modern high-performance processors are so fast that memory latency and bandwidth limitations become the performance bottlenecks for most scientific programs. Transformations such as loop fusion promote memory reuse and can significantly improve program efficiency for both scalar and vector machines [1, 8, 11, 40, 77, 132, 150, 189, 202]. For instance, consider the following example.

```
forall i = 1,N            forall i = 1,N
  A(i) = i                  A(i) = i
endfor             ==>      B(i) = A(i)*A(i)
forall i = 1,N            endfor
  B(i) = A(i)*A(i)
endfor
```

The occurrences of $A(i)$ in separate loops means that the memory location referenced by $A(i)$ in the first loop is likely to have been flushed from the cache by the reference in the second loop. If the two loops are fused, all accesses to $A(i)$ occur in the same loop iteration, allowing the value to be reused in a register or cache. For this example, we measured improvements of up to 30% for some problem sizes on an Intel i860, as shown in Figure 8.2. Additional transformations to enhance memory reuse and increase unit-stride memory accesses are also quite important; they are described elsewhere [116, 150].

Program Partitioning

The major step in compiling Fortran D for MIMD distributed-memory machines is to partition the data and computation across processors, introducing communication where needed. This process has been discussed in previous chapters. Two extensions are needed in the Fortran D back end to handle FORALL loops and intrinsics. During communication optimization, the Fortran D compiler treats all true dependences carried by FORALL loops as anti-dependences. This reflects the semantics of the FORALL loop and ensures that the message vectorization algorithm will place all communication outside the loop. During code generation intrinsic functions are translated into calls to the run-time library. Parameters are added where necessary to provide necessary data partitioning information.

Sectioning

The final phase of the Fortran D back end completes the scalarization process. After partitioning is performed, the compiler applies *sectioning* to convert FORALL loops into DO loops [8, 11] in the node program. The Fortran D back end detects cases where temporary storage may be needed using data dependence analysis. True dependences carried on the FORALL loop represent instances where values are defined in the loop and used on later iterations; they point out where the copy-in/copy-out semantics of the FORALL loop is being violated.

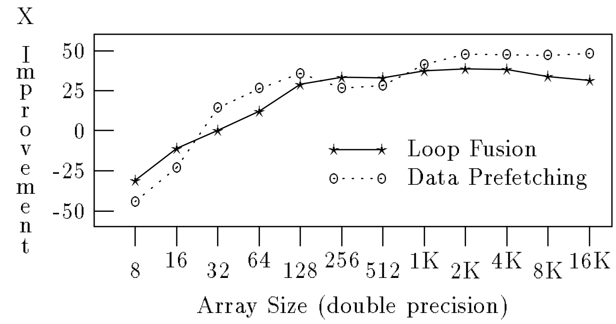


Figure 8.2 Effect of Scalarization Optimizations

During simple translation of Fortran 90 array constructs or FORALL loops, arrays involved in loop-carried true dependences must be saved in temporary buffers to preserve their old values. For instance, consider the translation of the following concise Fortran 90 formulation of the Jacobi algorithm:

```
A(2:N-1) = 0.5 * (A(1:N-2) + A(3:N))
       ⇓
forall i = 2,N-1
  A(i) = 0.5 * (A(i-1) + A(i+1))
endfor
       ⇓
do i = 1,N-2
  TMP(i) = A(i-1)
enddo
do i = 2,N-1
  A(i) = 0.5 * (TMP(i) + A(i+1))
enddo
```

A loop-carried true dependence exists between the definition to $A(i)$ and the use of $A(i-1)$. A temporary array TMP is needed so that the old values of $A(i-1)$ are not overwritten before they are used. The values of $A(i+1)$ do not need to be buffered since they are used before being redefined.

The previous example is problematic because temporary storage is required for the values of $A(i-1)$. In some cases, the Fortran D compiler can eliminate buffering through program transformations such as *loop reversal*. In other cases, the compiler can reduce the amount of temporary storage required through *data prefetching* [11]. For instance, in the Jacobi example a more efficient translation would result in:

```
X = A(1)
do i = 2,N-1
  Y = 0.5 * (X + A(i+1))
  X = A(i)
  A(i) = Y
endfor
```

This reduces the temporary memory required significantly, from an entire array to two scalars. For this version of Jacobi, we measured improvements of up to 50% for certain problem sizes on an Intel i860, as shown in Figure 8.2.

8.4 Run-time Library

Fortran 90 intrinsic functions represent computations (such as TRANSPOSE and MATMUL) that may have complex communication patterns. It is possible to support these functions at compile time, but we have chosen to implement these functions in the run-time library instead to reduce the complexity and machine-dependence of the compiler. The Fortran D compiler translates intrinsics into calls to run-time library routines using a standard interface. Additional information is passed describing bounds, overlaps, and

	Time (milliseconds)						
Proc	ALL	ANY	MAXVAL	PRODUCT	TRANSPOSE		
	$1K \times 1K$	$1K \times 1K$	$1K \times 1K$	$256K$	256×256	512×512	$1K \times 1K$
1	580.6	606.2	658.8	90.1	58	299	-
2	291.0	303.7	330.4	50.0	118	575	-
4	146.2	152.6	166.1	25.1	87	395	-
8	73.84	77.1	84.1	13.1	61	224	1039
16	37.9	39.4	43.4	7.2	41	140	539
32	19.9	20.7	23.2	4.2	36	85	316

Table 8.2 Performance of Some Fortran 90 Intrinsic Functions

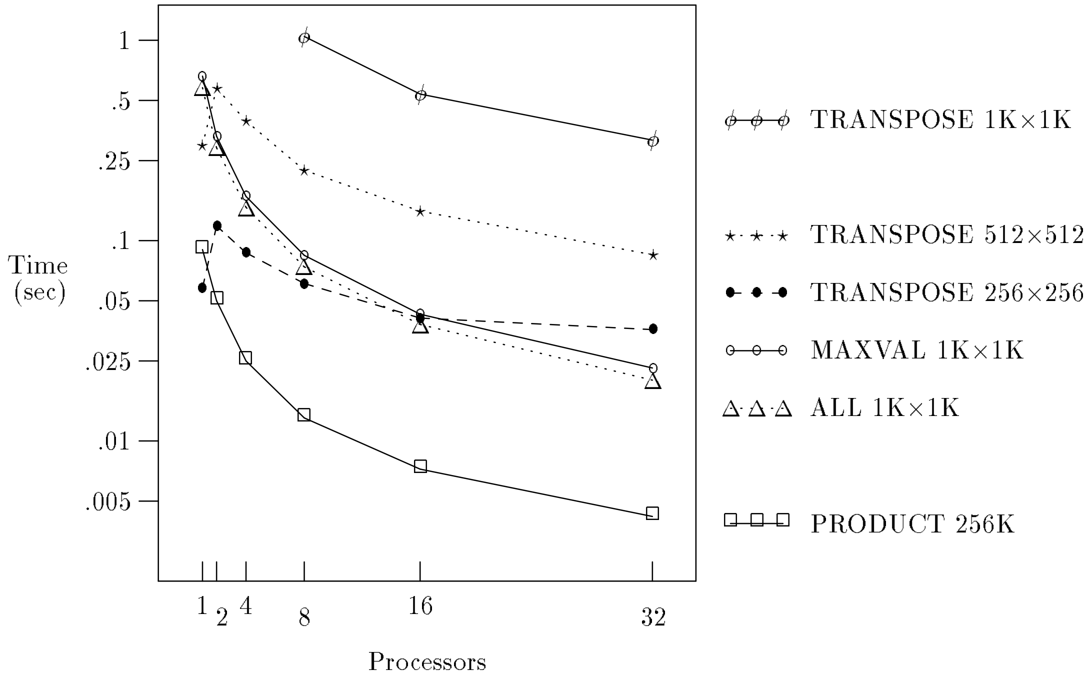


Figure 8.3 Performance of Run-time Library

partitioning for each array dimension. The run-time library is built on top of the Express communication package to ensure portability across different architectures [167].

Table 8.2 presents some sample performance numbers for a subset of the intrinsic functions on an iPSC/860, details are presented elsewhere [3]. The times in the table include both the computation and communication times for each function. These measurements are also displayed in Figure 8.3. Timings in seconds are plotted logarithmically along the Y-axis. The number of processors is plotted along the X-axis. Different lines in the graph correspond to timings of individual functions in the run-time library for different problem sizes.

We observe excellent performance for routines in the run-time library. For large problem sizes, we were able to obtain almost linear speedups for most intrinsics. An exception occurs in the case of the TRANSPOSE function, where going from one processor to two or four degrades execution time due to increased communication. However, speedup improves as the number of processors increases.

8.5 Fortran 90D Compilation Example

In this section we demonstrate how an example Fortran 90D program is compiled into message-passing Fortran 77, then measure its performance.

8.5.1 Compilation

Figure 8.4 shows a code fragment implementing one sweep of ADI integration on a 2D mesh, a typical (if short) numerical algorithm. Conceptually, the code is solving a tridiagonal system (represented by the arrays A and B) along each row of the matrix X. The tridiagonal systems are solved by a sequential method, but separate columns are independent and may be solved in parallel. The full version of ADI integration sweeps each dimension of the mesh, preventing completely parallel execution for any static data decomposition.

In the example, Fortran D data decomposition statements are used to partition the 2D array into blocks of columns. For clarity, we declare the number of processors (N$PROC) to be 32 at compile time. The Fortran 90D example is concise and convenient for the user, since it can be written for a single address

```
PARAMETER (N = 512, N$PROC = 32)
REAL X(N,N), A(N,N), B(N,N)
DECOMPOSITION D(N,N)
ALIGN X, A, B with D
DISTRIBUTE D(:,BLOCK)
do I = 2,N
  X(1:N,I) = X(1:N,I) - X(1:N,I-1)*A(1:N,I)/B(1:N,I-1)
  B(1:N,I) = B(1:N,I) - A(1:N,I)*A(1:N,I)/B(1:N,I-1)
enddo
X(1:N,N) = X(1:N,N) / B(1:N,N)
do J = N-1,1,-1
  X(1:N,J) = (X(1:N,J)-A(1:N,J+1)*X(1:N,J+1))/B(1:N,J)
enddo
```

Figure 8.4 ADI integration in Fortran 90D

```
PARAMETER (N = 512, N$PROC = 32)
REAL X(N,N), A(N,N), B(N,N)
do I = 2,N
  forall K = 1,N
    X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
  endfor
  forall K = 1,N
    B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
  endfor
enddo
forall K = 1,N
  X(K,N) = X(K,N)/B(K,N)
endfor
do J = N-1,1,-1
  forall K = 1,N
    X(K,J) = (X(K,J)-A(K,J+1)*X(K,J+1))/B(K,J)
  endfor
enddo
```

Figure 8.5 ADI in Intermediate Form

space without requiring explicit communication. However, additional compilation techniques are required to generate efficient code. First, the Fortran 90D front end translates the program into intermediate form as shown in Figure 8.5, converting all array constructs into FORALL loops. Since no true dependences are carried on the FORALL loops, they may be directly replaced with DO loops.

The compilation process for the Fortran D back end merits closer examination. First, array bounds are reduced to the local sections plus overlaps. The local processor number is determined using *myproc*(), a library function; it is used to compute expressions for reducing loop bounds. Analysis determines that both I and J are *cross-processor* loops—loops carrying true dependences that sequentialize the computation across processors. To exploit pipeline parallelism, the Fortran D compiler interchanges such loops inward. This *fine-grain pipelining* optimization is discussed in Chapter 5.

For this version of ADI integration, data dependences permit the Fortran D compiler to interchange the J loop inwards. However, if loop fusion is not performed, the imperfectly nested K loops inhibit loop interchange for loop I, forcing it to remain in place. During code generation, true dependences for nonlocal references carried on the I and J loop cause calls to *send* and *recv* to be inserted to provide communication and synchronization.

Figure 8.6 shows the resulting program. Many details in the example programs have been elided or simplified; however, they are precisely equivalent to code generated and executed on the iPSC/860. Unfortunately, the computation in the I loop has been sequentialized, since each processor has to wait for its predecessor to complete. Note that this is not due to communication placement; the values needed by the succeeding processor are simply computed last.

```
REAL X(512,0:17), A(512,17), B(512,0:16)
my$p = myproc()        {* 0...31 *}
lb1 = max((my$p*16)+1,2) - my$p*16
ub1 = min((my$p+1)*16,511) - my$p*16
if (my$p .gt. 0) recv(X(1:N,0),B(1:N,0),my$p-1)
do I = lb1, 16
  do K = 1,N
    X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
  enddo
  do K = 1,N
    B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
  enddo
enddo
if (my$p .lt. 31) send(X(1:N,16),B(1:N,16),my$p+1)
if (my$p .eq. 31) then
  do K = 1,N
    X(K,16) = X(K,16)/B(K,16)
  enddo
endif
if (my$p .gt. 0) send(A(1:N,1),my$p-1)
if (my$p .lt. 31) recv(A(1:N,17),my$p+1)
do K = 1,N
  if (my$p .lt. 31) recv(X(K,17),my$p+1)
  do J = ub1,1,-1
    X(K,J) = (X(K,J)-A(K,J+1)*X(K,J+1))/B(K,J)
  enddo
  if (my$p .gt. 0) send(X(K,1),my$p-1)
enddo
```

Figure 8.6 ADI without Loop Fusion

If loop fusion is enabled, the Fortran D back end will fuse the two inner K loops. This is legal because the dependence between the definition and use of B is carried on the I loop and is thus unaffected. Fusion is also conservative because it does not introduce any true dependences carried by the K loop. Fusing the K loops promotes reuse of A and B, but its main benefit is to enable the Fortran D back end to interchange the I and K loops, exposing pipeline parallelism. The resulting program is displayed in Figure 8.7. For simplicity, only the first loop is shown. The remaining loops are compiled in a similar manner as before.

To reduce communication overhead, we can also apply strip-mining in conjunction with loop interchange to adjust the granularity of pipelining. This technique is called *coarse-grain pipelining* in Chapter 5. In the ADI example, we strip-mine the K loop by four (an empirically derived value), then interchange the resulting loop outside the I loop. Messages inserted outside the K loop allow each processor to reduce communication costs at the expense of some parallelism, resulting in Figure 8.8. All these versions of ADI integration were generated automatically by the Fortran D compiler.

8.5.2 Performance Results

To validate these methods, we executed these codes for double precision arrays on an Intel iPSC/860. The programs were compiled under -O4 using Release 3.0 of *if77*, the iPSC/860 compiler. Timings were taken using *dclock*() on a 32 node Intel iPSC/860 with 8 Meg of memory per node. Results for three problem sizes are tabulated in Table 8.3. Timings are not provided where problem size exceeds available memory.

We also graphically display the timings in Figure 8.9. Execution times in seconds are plotted logarithmically along the Y-axis. The number of processors used is plotted logarithmically along the X-axis. Numbers for perfect or ideal speedup (assuming no communication cost) are provided for comparison.

The original version of ADI (Figure 8.6) exploits pipeline parallelism in the J loop, but shows limited speedup, since the I loop is sequentialized. Fusing the K loops to improve memory reuse provides very little improvement in this case, yielding nearly identical results. Applying loop interchange after fusion to enable

```
REAL X(512,0:17), A(512,17), B(512,0:16)
my$p = myproc()        {* 0...31 *}
lb1 = max((my$p*16)+1,2) - my$p*16
do K = 1,N
  if (my$p .gt. 0) recv(X(K,0),B(K,0),my$p-1)
  do I = lb1,16
    X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
    B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
  enddo
  if (my$p .lt. 31) send(X(K,16),B(K,16),my$p+1)
enddo
```

Figure 8.7 ADI with Fine-grain Pipelining

```
REAL X(512,0:17), A(512,17), B(512,0:16)
my$p = myproc()        {* 0...31 *}
lb1 = max((my$p*16)+1,2) - my$p*16
do KK = 1,N,4
  if (my$p .gt. 0) recv(X(KK:KK+3,0),B(KK:KK+3,0),my$p-1)
  do I = lb1,16
    do K = KK,KK+3
      X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
      B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
    enddo
  enddo
  if (my$p .lt. 31) send(X(KK:KK+3,16),B(KK:KK+3,16),my$p+1)
enddo
```

Figure 8.8 ADI with Coarse-grain Pipelining

Problem Size	P	ADI w/o Loop Fusion	Fine-grain Pipelining	Coarse-grain Pipelining	Data Redistribution	Perfect Speedup
	1	1.22	1.45	1.32	-	1.22
256	2	1.13	0.78	0.69	1.32	0.61
×	4	1.02	0.45	0.38	0.83	0.30
256	8	0.96	0.28	0.21	0.52	0.15
	16	0.93	0.20	0.12	0.32	0.08
	32	0.96	0.17	0.08	0.25	0.04
	1	5.44	6.26	5.93	-	5.44
512	2	4.79	3.18	2.98	6.17	2.72
×	4	4.29	1.72	1.53	3.72	1.36
512	8	4.04	0.97	0.81	2.02	0.68
	16	3.94	0.59	0.44	1.18	0.34
	32	3.94	0.41	0.26	0.68	0.17
	1	21.74	-	-	-	21.74
1K	4	17.13	6.44	5.98	-	5.44
×	8	16.09	3.42	3.07	8.95	2.72
1K	16	15.61	1.91	1.62	4.59	1.36
	32	15.47	1.17	0.89	2.58	0.68

Table 8.3 Performance of ADI Integration (in seconds)

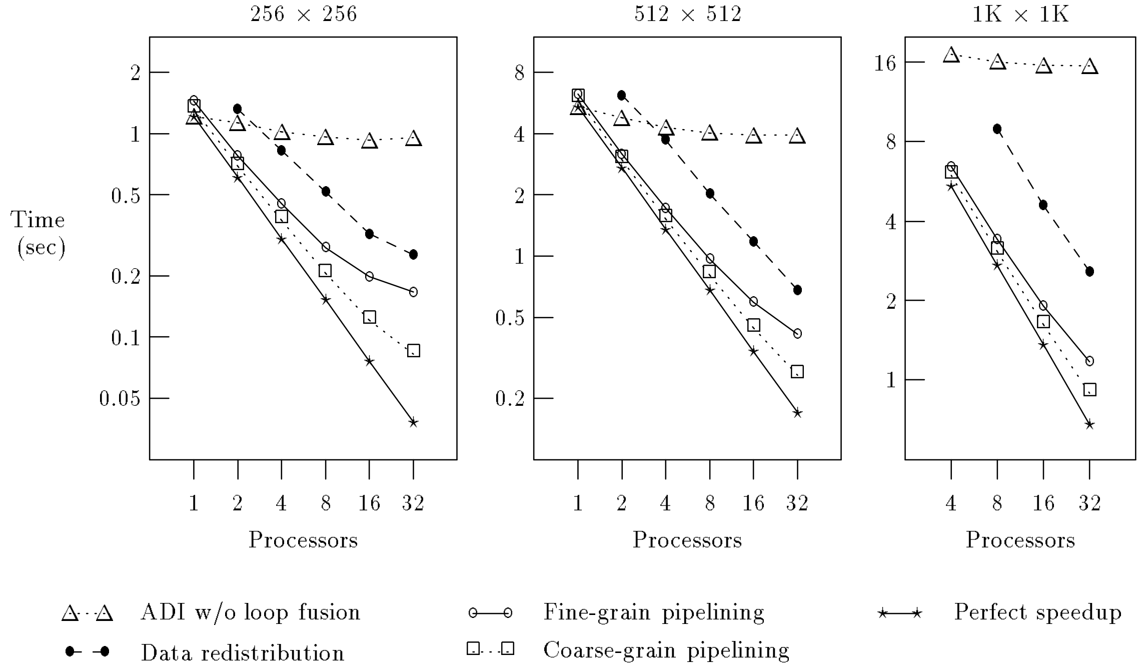


Figure 8.9 Execution Times for ADI Integration (double precision)

fine-grain pipelining (Figure 8.7) parallelizes the I loop as well, yielding significant speedup. Strip-mining to apply coarse-grain pipelining can improve efficiency an additional 10–50% (Figure 8.8). Pipelining comes closest to perfect speedup for large problems on a few processors.

We also compared the efficiency of pipelining versus dynamic data decomposition. By changing the distribution of data at run-time from columns to rows, all dependences in each sweep of ADI may be internalized, enabling completely parallel execution. Data must be redistributed twice, once to achieve the desired distribution, then a second time to return it to its original configuration. The cost of redistributing is approximated by the performance of the TRANSPOSE routine shown in Table 8.2.

Our results show that on the iPSC/860, dynamic data decomposition for this formulation of ADI integration achieves speedup. However, the resulting program is significantly slower than pipelining, even for small problems distributed over large numbers of processors, the expected best case for dynamic data decomposition. Our experiences show that some common algorithms, such as ADI integration, require significant amounts of optimization to compete with hand-crafted code.

8.6 Discussion

This chapter presents an integrated approach to compiling both Fortran 77D and 90D based on a few key observations. First, using FORALL preserves information in Fortran 90 array constructs. Dividing the scalarization process into translation, loop fusion, and sectioning allows it to be easily integrated with the partitioning performed by the Fortran D compiler. A portable run-time library can also reduce the complexity and machine-dependence of the compiler. All optimizations except data prefetching have been implemented in the current Fortran D compiler prototype.

Chapter 9

Preliminary Experiences

We use case studies to illustrate strengths and weaknesses of the prototype Fortran D compiler when compiling linear algebra codes, large subroutines, and whole programs. We compare the performance of compiler-generated code against hand-coded kernels and programs on the Intel iPSC/860. Acceptable performance is obtained for parallel stencil computations, but improvements are needed for linear algebra and pipelined computations. The Fortran D compiler significantly outperforms the CM Fortran compiler on the Thinking Machines CM-5. These experiences show that the Fortran D compiler is successful for parallel stencil computations. However, it requires symbolic analysis, greater flexibility, and improved optimization of communication-intensive codes such as linear algebra & pipelined computations.

9.1 Introduction

In this chapter, we present some preliminary experiences with the prototype Fortran D compiler. We begin by describing its implementation status, then point out its strengths and weaknesses using case studies of four example programs and subroutines:

- DGEFA — Gaussian elimination subroutine from LINPACK
- SHALLOW — weather prediction benchmark using finite-difference
- DISPER — subroutine from UTCOMP, an oil reservoir simulator
- ERLEBACHER — tridiagonal solver benchmark using ADI integration

To evaluate the efficiency of the Fortran D compiler, we compare the performance of its output for these programs and some kernels against hand-coded versions on the Intel iPSC/860. We also compare the Fortran D and CM Fortran compilers on the Thinking Machines CM-5 by translating representative kernels into CM Fortran. Results are discussed and used to point out directions for future research.

9.2 Fortran D Compiler Prototype

The prototype Fortran D compiler is implemented as a source-to-source Fortran translator in the context of the ParaScope parallel programming environment [35, 59]. It utilizes existing tools for performing dependence analysis, program transformations, and interprocedural analysis [62, 83, 117]. The design of the prototype compiler has been described in the preceding chapters. The current implementation supports:

- inter-dimensional alignments
- 1D BLOCK and CYCLIC distributions
- loop interchange, fusion, distribution, strip-mining
- message vectorization, coalescing, aggregation
- vector message pipelining
- broadcasts, collective communications, point-to-point messages
- SUM, PRODUCT, MIN, MAX, MINLOC, MAXLOC reductions
- fine-grain and coarse-grain pipelining (preset granularity)
- relax "owner computes rule" for reductions, private variables
- nonlocal storage in overlaps, buffers
- loop bounds reduction, guard introduction
- global ↔ local index conversion

- interprocedural reaching decompositions, overlap offsets
- common blocks
- I/O (performed by processor 0)
- generation of calls to the Intel NX/2 message-passing library

For simplicity, the prototype compiler requires that all array sizes, loop bounds, and number of processors in the target machine to be compile-time constants. All subscripts must also be of the form c or $i + c$, where c is a compile-time constant and i is a loop index variable. These restrictions are not due to limitations of our compilation techniques, but reflect the immaturity of the prototype compiler.

9.3 Fortran D Compilation Case Studies

Our previous examples with the Fortran D compiler have mostly dealt with stencil computation kernels from iterative partial difference equation (PDE) solvers. Here we illustrate the compilation process for linear algebra kernels, large subroutines, and whole programs. For these more complex codes, we find that the compiler must be able to:

- Partition computation in complex non-uniform loop bodies across processors.
- Provide robust translation of loop index variables and loop bounds between global and local indices.
- Relax the owner computes rule, particularly for reductions performed by replicated variables.

Section 4.2.4 described how statement groups formed during partitioning analysis may be used to guide partitioning and loop bounds & index generation for non-uniform loops. Section 5.4.2 has discussed relaxing the owner computes rule for reductions and private variables. We show during compilation of DGEFA and ERLEBACHER how this technique may be extended for location reductions, multi-reductions, and replicated variables.

9.3.1 DGEFA

DGEFA is a key subroutine in LINPACK written by Jack Dongarra *et al.* at Argonne National Laboratory, and is also the principal computation kernel in the LINPACKD benchmark program. DGEFA performs LU decomposition through Gaussian elimination with partial pivoting. Its memory access patterns are quite different from stencil computations, and is representative of linear algebra computations. As many linear algebra algorithms involve factoring matrices, CYCLIC and BLOCK_CYCLIC data distributions are desirable for maintaining good load balance. These distributions and the prevalence of triangular loop nests pose additional challenges to the Fortran D compiler.

Figure 9.1 shows the original program as well as the output produced by the prototype Fortran D compiler. For good load balance we choose a column-cyclic distribution, scattering array columns round-robin across processors. The Fortran D compiler then uses this data decomposition to derive the computation partition. The most complex part of compilation is generating the proper loop bounds and indices, using methods previously described in Chapter 4.

MIN/MAX and MINLOC/MAX Reductions

During compilation, the Fortran D compiler begins by putting statements in the body of the k loop into statement groups. The first group is the pivot selection step, $S_1 \ldots S_4$. The compiler recognizes it as a MAX/MAXLOC reduction by detecting that the *lhs* of an assignment *al* at statement S_3 is being compared against its *rhs* in an enclosing IF statement. The level of the reduction is set to the k loop, since it is the deepest loop enclosing a use of *al*. At this loop level, the reduction only examines a single column of a. Since array a has been distributed by columns, the reduction may be computed locally by the processor owning the column. The Fortran D compiler relaxes the owner computes rule for the reduction. It inserts a guard to ensure the reduction is performed by the processor owning column k, then broadcasts the result.

```
{* Original Fortran D Program *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l
  DOUBLE PRECISION a(n,n),al,t
  DISTRIBUTE a(:,CYCLIC)
  do k = 1, n-1
    {* Find max element in a(k:n,k) *}
S1  l = k
S2  al = dabs(a(k, k))
    do i = k + 1, n
      if (dabs(a(i, k)) .GT. al) then
S3      al = dabs(a(i, k))
S4      l = i
      endif
    enddo
    ipvt(k) = l
    if (al .NE. 0) then
      if (l .NE. k) then
        t = a(l,k)
        a(l,k) = a(k,k)
        a(k,k) = t
      endif
      {* Compute multipliers in a(k+1:n,k) *}
      t = -1.0d0/a(k,k)
      do i = k+1, n
        a(i, k) = a(i, k) * t
      enddo
      {* Reduce remaining submatrix *}
      do j = k+1, n
        t = a(l,j)
        if (l .NE. k) then
          a(l,j) = a(k,j)
          a(k,j) = t
        endif
        do i = k+1, n
S5        a(i, j) = a(i, j) + t*a(i, k)
        enddo
      enddo
    endif
  enddo
  ipvt(n) = n
end
```

```
{* Compiler Output for 4 Processors *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l
  DOUBLE PRECISION a(n,n/4),al,t,dp$buf1(n)
  do k = 1, n-1
    k$ = ((k - 1) / 4) + 1
    {* Find max element in a(k:n,k$) *}
    if (my$p .EQ. MOD(k - 1, 4)) then
      l = k
      al = dabs(a(k, k$))
      do i = k + 1, n
        if (dabs(a(i, k$)) .GT. al) then
          al = dabs(a(i, k$))
          l = i
        endif
      enddo
      broadcast l, al
    else
      recv l, al
    endif
    ipvt(k) = l
    if (al .NE. 0) then
      if (my$p .EQ. MOD(k - 1, 4)) then
        if (l .NE. k$) then
          t = a(l,k$)
          a(l,k$) = a(k,k$)
          a(k,k$) = t
        endif
        {* Compute multipliers in a(k+1:n,k$) *}
        t = -1.0d0/a(k,k$)
        do i = k+1, n
          a(i, k$) = a(i, k$) * t
        enddo
      endif
      {* Reduce remaining submatrix *}
      if (my$p .EQ. MOD(k - 1, 4)) then
        buffer a(k+1:n, k$) into dp$buf1
        broadcast dp$buf1(1:n-k)
      else
        recv dp$buf1(1:n-k)
      endif
      lb$1 = (k / 4) + 1
      if (my$p .LT. MOD(k, 4)) lb$1 = lb$1+1
      do j = lb$1, n
        t = a(l,j)
        if (l .NE. k) then
          a(l,j) = a(k,j)
          a(k,j) = t
        endif
        do i = k+1, n
          a(i, j) = a(i, j) + t*dp$buf1(i-k)
        enddo
      enddo
    endif
  enddo
  ipvt(n) = n
end
```

Figure 9.1 DGEFA: Gaussian Elimination with Partial Pivoting

For MIN/MAX and MINLOC/MAXLOC reductions, the Fortran D compiler must also search for initialization statements for the *lhs* of assignment statements in the k loop, assigning them the same iteration set as the body of the reduction. In DGEFA this identifies statements S_1 and S_2 as initialization statements for the MAX/MAXLOC reduction at S_3. By putting them in the same statement group as the reduction, the Fortran D compiler avoids inserting an additional broadcast to update the value of al at S_2.

Buffer Indexing

Once the reduction is processed, the Fortran D compiler inserts a guard to ensure the column of multipliers is calculated only by the processor owning the column. Since all processors participate in the final submatrix elimination, the processor owning the multiplier column must broadcast it to the other processors. The Fortran D compiler uses temporary buffers to store nonlocal data communicated by broadcasts and point-to-point messages. Because this data is packed into 1D buffers, the compiler has the responsibility of generating the proper subscripts for indexing into the buffers. The task is made more difficult when multiple references in the original program reference the nonlocal data. The Fortran D compiler examines the subscripts of the original reference and linearizes those traversing the nonlocal portions. For instance, the reference to $a(i,k)$ in S_5 of DGEFA is broadcast and received in buffer *dp$buf*1. Later it is accessed directly out of the buffer, using the generated subscript *dp$buf*1$(i-k)$.

9.3.2 SHALLOW

SHALLOW is a 200 line benchmark weather prediction program written by Paul Swarztrauber, National Center for Atmospheric Research (NCAR). It is a stencil computation that applies finite-difference methods to solve shallow-water equations. SHALLOW is representative of a large class of existing supercomputer applications. The computation is highly data-parallel and well-suited for MIMD distributed-memory machines.

Figure 9.2 outlines the version of SHALLOW we used to test the Fortran D compiler; it was modified to eliminate I/O. Data can be partitioned quite simply by aligning all 2D arrays identically, then distributing the result column-wise. We chose to block distribute the second dimension, assigning a block of columns to each processor. The prototype Fortran D compiler was able to generate message-passing code fairly simply. The principal issues encountered during compilation were boundary conditions, loop distribution, and inter-loop communication optimizations.

Boundary Conditions

SHALLOW contains many code fragments solving boundary conditions for periodic continuations. As a result, the Fortran D compiler needed to insert explicit guards for many statement groups. These boundary conditions also required the creation of several individual point-to-point messages between boundary processors to transfer data required.

Loop Distribution

Because of the programming style used in writing SHALLOW, almost all loop nests were non-uniform, *i.e.,* contained statements with differing iteration sets. Fortunately, none of the loops carried recurrences, so the Fortran D compiler applies loop distribution to separate statements, creating uniform loop nests. Loop bounds reduction is then sufficient to partition the computation during code generation, excepting boundary conditions.

Inter-loop Message Coalescing and Aggregation

While loop distribution enables inexpensive partitioning of the program computation, it has the disadvantage of creating a large number of loop nests. In many cases these loop nests, along with loops representing boundary conditions, required communication with neighboring processors. The current Fortran D compiler prototype applies message coalescing and aggregation only within a single loop nest. Its output for SHALLOW thus missed many opportunities to coalesce or aggregate messages because the nonlocal references were located in loop nests not enclosed by a common loop. By applying message coalescing and aggregation manually across loop nests, we were able to eliminate about half of all calls to communication routines.

```
{* Original Fortran D Program *}
PROGRAM SHALLOW
  REAL u(N,N),v(N,N),p(N,N),unew(N,N),pnew(N,N),vnew(N,N),psi(N,N)
  REAL pold(N,N),uold(N,N),vold(N,N),cu(N,N),cv(N,N),z(N,N),h(N,N)
  DECOMPOSITION d(N,N)
  ALIGN u,v,p,unew,pnew,vnew,psi,pold,uold,vold,cu,cv,z,h WITH d
  DISTRIBUTE d(:,BLOCK)
  {* initial values of the stream function & velocities *}
  do j = 1,N-1
    do i = 1,N-1
      u(i+1,j) = -(psi(i+1,j+1)-psi(i+1,j))*dy
      v(i,j+1) = (psi(i+1,j+1)-psi(i,j+1))*dx
    enddo
  enddo
  do k = 1,Time
    {* periodic continuation *}
    ...
    {* compute capital u, capital v, z, and h *}
    do j = 1,N-1
      do i = 1,N-1
        cu(i+1,j) = .5*(p(i+1,j)+p(i,j))*u(i+1,j)
        cv(i,j+1) = .5*(p(i,j+1)+p(i,j))*v(i,j+1)
        z(i+1,j+1) = (fsdx*(v(i+1,j+1)-v(i,j+1))-fsdy*(u(i+1,j+1)
              -u(i+1,j))) / (p(i,j)+p(i+1,j) +p(i+1,j+1)+p(i,j+1))
        h(i,j) = p(i,j)+.25*(u(i+1,j)*u(i+1,j)+u(i,j)*u(i,j)+v(i,j+1)
              *v(i,j+1)+v(i,j)*v(i,j))
        enddo
    enddo
    {* periodic continuation *}
    ...
    {* compute new values u, v, and p *}
    do j = 1,N-1
      do i = 1,N-1
        unew(i+1,j) = uold(i+1,j)+tdts8*(z(i+1,j+1)+z(i+1,j))*(cv(i+1,j+1)
              +cv(i,j+1)+cv(i,j)+cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
        vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1) +z(i,j+1))*(cu(i+1,j+1)
              +cu(i,j+1)+cu(i,j)+cu(i+1,j))-tdtsdy*(h(i,j+1)-h(i,j))
        pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j))
              -tdtsdy*(cv(i,j+1)-cv(i,j))
      enddo
    enddo
  enddo
end
```

Figure 9.2 SHALLOW: Weather Prediction Benchmark

9.3.3 DISPER

DISPER is a 1000 line subroutine for computing dispersion terms. It is taken from UTCOMP, a 33,000 line oil reservoir simulator developed at the University of Texas at Austin. Like SHALLOW, DISPER is a stencil computation that is highly data-parallel and well-suited for the Fortran D compiler. Unfortunately, UTCOMP was originally written for a Cray vector machine. Arrays were linearized to ensure long vector lengths, then addressed through complex subscript expressions and indirection arrays. This style of programming, while efficient for vector machines, does not lend itself to massively-parallel machines.

To explore whether UTCOMP can be written in a machine-independent programming style using Fortran D or High Performance Fortran (HPF), researchers at Rice rewrote DISPER to have regular accesses and simple subscripts on multidimensional arrays. Figure 9.3 shows a fragment of the rewritten form of DISPER. Its main arrays have differing sizes and dimensionality, but have the same size in the first dimension. Arrays were aligned along the first dimension and distributed block-wise. The resulting code was was for the most part compiled successfully by the prototype Fortran D compiler.

Execution Conditions

The major difficulty encountered by the Fortran D compiler was the existence of execution conditions caused by explicit guards in the input code. There are two types of execution conditions. Data-dependent execution

```
{* Original Fortran D Program *}
SUBROUTINE DISPER
  LOGICAL lsat(256)
  DOUBLE PRECISION ddx(256,8,8), ddy(256,8,8), ddz(256,8,8)
  DOUBLE PRECISION pmfr(256,8,8,4,5), gradx(256), grady(256), gradz(256)
  DECOMPOSITION d(256)
  ALIGN ddx(i,j,k),ddy(i,j,k),ddz(i,j,k) WITH d(i)
  ALIGN lsat(i,j,k,l),pmfr(i,j,k,l,m) WITH d(i)
  ALIGN gradz,grady,gradz WITH d
  DISTRIBUTE d(BLOCK)
  {* compute dispersion terms *}
  do j = 2,4
    do i3 = 1,8
      do i2 = 1,8
        do i1 = 1,256
          ...
S1        if ((i1 .NE. 1) .AND. (i1 .NE. 256)) then
S2          if (lsat(i1-1,i2,i3,j) .AND. lsat(i1+1,i2,i3,j)) then
S3            grady(i1)=(pmfr(i1+1,i2,i3,j,k)-pmfr(i1-1,i2,i3,j,k)) /
                (.5 * (ddy(i1+1,i2,i3) + ddy(i1-1,i2,i3)) + ddy(i1,i2,i3))
              ...
            endif
          endif
          ...
        enddo
      enddo
    enddo
  enddo
end
```

Figure 9.3 DISPER: Oil Reservoir Simulation

conditions, such as the guard at S_2 in Figure 9.3, were not a problem. Message vectorization moves communication caused by such guarded statements out of the enclosing loops. Overcommunication may result if the statement is not executed, but the resulting code is still much more efficient than sending individual messages after evaluating each guard.

Execution conditions that reshape the iteration space, on the other hand, pose a significant problem. For instance, the guard at S_1 in Figure 9.3 restricts the execution of statement S_3 on the first and last iteration of loop $i1$. It has in effect changed the iteration set for the assignment S_3, causing it to be executed on a subset of the iterations. These guards are frequently used by programmers to isolate boundary conditions in a modular manner, avoiding the need to peel off loop iterations.

Unlike data-dependent execution conditions, these execution conditions always hold and can be detected at compile-time. If they are not considered, the compiler will generate communication for nonlocal accesses that never occur. Future versions of the Fortran D compiler will need to examine guard expressions. If its effects on the iteration set can be determined at compile-time, the iteration set of the guarded statements must be modified appropriately. Because this functionality is not present in the current Fortran D compiler, unnecessary guards and communication in the compiler output were corrected by hand.

9.3.4 ERLEBACHER

ERLEBACHER is a 800 line benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). It performs 3D tridiagonal solves using Alternating-Direction-Implicit (ADI) integration. Like Jacobi iteration and Successive-Over-Relaxation (SOR), ADI integration is a technique frequently used to solve PDEs. However, it performs vectorized tridiagonal solves in each dimension, resulting in computation wavefronts across all three dimensions of the data array.

Each sweep in ERLEBACHER consists of a set-up and computation phase, followed by forward and backward substitutions. Figures 9.4 and 9.5 illustrate the core computation performed by ERLEBACHER during a sweep of the Z dimension. We chose to distribute the Z dimension of all 3D arrays blockwise; all 1D and 2D arrays are replicated. Here we relate some issues that arose during compilation of Erlebacher to a machine with four processors, $P_0 \ldots P_3$.

```
{* Original Fortran D Program *}
 SUBROUTINE DZ3D6P
  REAL uud(n,n,n),uu(n,n,n)
  DECOMPOSITION dd(n,n,n)
  ALIGN uud, uu with dd
  DISTRIBUTE dd(:,:,BLOCK)
  do j = 1,n
   do i = 1,n
S1  uud(i,j,1) =
       F(uu(i,j,3),uu(i,j,n-1))
S2  uud(i,j,2) =
       F(uu(i,j,4),uu(i,j,n))
S3  uud(i,j,n-1) =
       F(uu(i,j,1),uu(i,j,n-3))
S4  uud(i,j,n) =
       F(uu(i,j,2),uu(i,j,n-2))
   enddo
  enddo
  do k = 3,n-2
   do j = 1,n
    do i = 1,n
S5   uud(i,j,k) =
        F(uu(i,j,k+2),uu(i,j,k-2))
    enddo
   enddo
  enddo
 end
```

```
   {* Compiler Output for 4 Processors *}
   SUBROUTINE DZ3D6P
   REAL uud(n,n,n),uu(n,n,-1:(n/4)+2)
    n$ = n/4
    if (my$p .EQ. 0)
C1   send uu(1:n,1:n,1:2) to P3
    if (my$p .EQ. 3)
C2   send uu(1:n,1:n,n$-1:n$) to P0
    if (my$p .LT. 3)
C3   send uu(1:n,1:n,n$-1:n$) to my$p+1
    if (my$p .GT. 0)
C4   send uu(1:n,1:n,1:2) to my$p-1
    if (my$p .EQ. 0) then
     recv uu(1:n,1:n,n$+1:n$+2) from P3
     do j = 1,n
      do i = 1,n
S1     uud(i,j,1) = F(...)
S2     uud(i,j,2) = F(...)
      enddo
     enddo
    endif
    if (my$p .EQ. 3) then
     recv uu(1:n,1:n,-1:0) from P1
     do j = 1,n
      do i = 1,n
S3     uud(i,j,n$-1) = F(...)
S4     uud(i,j,n$) = F(...)
      enddo
     enddo
    endif
    if (my$p .GT. 0)
     recv uu(1:n,1:n,n$+1:n$+2) from my$p+1
    if (my$p .LT. 3)
     recv uu(1:n,1:n,-1:0) from my$p-1
    do k = lb$,ub$
     do j = 1,n
      do i = 1,n
S5     uud(i,j,k) = F(...)
      enddo
     enddo
    enddo
   end
```

Figure 9.4 Erlebacher: Computation Phase in Z Dimension

Overlapping Communication with Computation

In Erlebacher, we discovered unexpected benefits for vector message pipelining, an optimization discussed in Chapter 5 that separates matching *send* and *recv* statements to create opportunities for overlapping communication with computation. Consider compilation of the setup phase in the Z dimension, shown in Figure 9.4. The Fortran D compiler first distributes the loops enclosing statements $S_1 \ldots S_4$ because they belong to two distinct statement groups. Message vectorization then extracts all communication outside of each loop nest. The Fortran D compiler then applies vector message pipelining.

We found vector message pipelining to be particularly effective here because it moves the *send*s C_3 and C_4 before the *recv*s in the first two loop nests. If C_3 and C_4 are left in their original positions before S_5, the computation will be idle until two message transfers complete, because the boundary processors P_0 and P_3 will need to first exchange messages before communicating to the interior processors. The prototype thus saved the cost of waiting for an entire message. More advanced analysis could determine that the statements $S_1 \ldots S_4$ are simply incarnations of statement S_5 created to handle periodic boundary conditions. We can perform the reverse of *index set splitting* and merge the loop bodies to simplify the resulting code.

```
{* Original Fortran D Program *}
SUBROUTINE TRIDVPK
 REAL a(n),b(n),c(n),d(n),e(n)
 REAL tot(n,n),f(n,n,n)
 DISTRIBUTE f(:,:,BLOCK)
 {* perform forward substitution *}
 ...
 {* perform backward substitution *}
 do k = 1,n
  do j = 1,n
   do i = 1,n
S1    tot(i,j) =
        tot(i,j)+d(k)*f(i,j,k)
   enddo
  enddo
 enddo
 do j = 1,n
  do i = 1,n
S2   f(i,j,n) =
        (f(i,j,n)-tot(i,j))*b(n)
  enddo
 enddo
 do j = 1,n
  do i = 1,n
S3   f(i,j,n-1) =
        f(i,j,n-1)-e(n-1)*f(i,j,n)
  enddo
 enddo
 do k = n-2,1,-1
  do j = 1,n
   do i = 1,n
S4    f(i,j,k) = f(i,j,k)-c(k)*
        f(i,j,k+1)-e(k)*f(i,j,n)
   enddo
  enddo
 enddo
end
```

```
{* Compiler Output for 4 Processors *}
SUBROUTINE TRIDVPK
 REAL a(n),b(n),c(n),d(n),e(n)
 REAL tot(n,n),f(n,n,0:(n/4)+1),r$buf1(n)
 {* perform forward substitution *}
 ...
 {* perform backward substitution *}
 n$ = n/4
 off$0 = my$p * n$
 do k = 1,n$
  k$ = k + off$0
  do j = 1,n
   do i = 1,n
    tot(i,j) = tot(i,j)+d(k$)*f(i,j,k)
   enddo
  enddo
 enddo
 global-sum tot(1:n,j:n)
 if (my$p .EQ. 3) then
  do j = 1,n
   do i = 1,n
    f(i,j,n$-1) = (f(i,j,n$)-tot(i,j))*b(n)
   enddo
  enddo
  do j = 1,n
   do i = 1,n
    f(i,j,n$-1) = f(i,j,n$-1)-e(n-1)*f(i,j,n$)
   enddo
  enddo
  buffer f(1:128, 1:128, n$) into rbuf$1(n*n)
  broadcast rbuf$1(1:n*n)
 else
  recv rbuf$1(1:n*n)
 endif
 do j = 1,n
  do i$ = 1,n,8
   i$up = i$+7
   if (my$p .LT. 3)
    recv f(i$:i$up, j, n$+1) from my$p+1
   do i = i$,i$+8
    do k = ub$,1,-1
     k$ = k + off$0
     f(i,j,k) = f(i,j,k)-c(k$)*f(i,j,k+1)
              - e(k$)*r$buf1(j*n+i-n)
    enddo
   enddo
   if (my$p .GT. 0)
    send f(i$:i$up, j, 1) to my$p-1
  enddo
 enddo
end
```

Figure 9.5 ERLEBACHER: Solution Phase in Z Dimension

Multi-Reductions

Another problem faced by the Fortran D compiler was handling reductions on replicated variables. A multidimensional reduction performs a reduction on multiple dimensions of an array. Finding the maximum value in a 3D array would be a 3D MAX reduction over an n^3 data set. We examine a special case of multidimensional reduction that we call a *multi-reduction*, where the program performs multiple reductions simultaneously. For instance, finding the maximum value of each column in a 3D array would be a 2D MAX multi-reduction composed of n^2 1D MAX reductions. Unlike normal multidimensional reductions, multi-reductions are directional in that they only transfer data across certain dimensions. This property allows the compiler to determine when communication is necessary. It also allows the problem to be partitioned in other dimensions so that no global reductions are required at the end.

The Fortran D compiler handles multi-reductions as follows. If the direction of the multi-reduction crosses a partitioned array dimension, then compilation proceeds as normal. The compiler produces code so

```
     {* Original Fortran D Program *}
     SUBROUTINE TRIDVPJ
      REAL a(n),b(n),c(n),d(n),e(n)
      REAL tot(n,n),f(n,n,n)
      DISTRIBUTE f(:,:,BLOCK)
      do k = 1,n
       do j = 1,n
        do i = 1,n
S1       tot(i,k) =
            tot(i,k) + d(j)*f(i,j,k)
        enddo
       enddo
      enddo
      do k = 1,n
       do i = 1,n
S2      f(i,n,k) =
            (f(i,n,k) - tot(i,k))*b(n)
       enddo
      enddo
     end
```

```
     {* Compiler Output for 4 Processors *}
     SUBROUTINE TRIDVPK
      REAL a(n),b(n),c(n),d(n),e(n)
      REAL tot(n,n),f(n,n,0:(n/4)+1)
      n$ = n/4
      off$0 = my$p * n$
      do k = 1,n$
       k$ = k + off$0
       do j = 1,n
        do i = 1,n
         tot(i,k$) = tot(i,k$) + d(j)*f(i,j,k)
        enddo
       enddo
      enddo
      global-concat tot(1:n,j:n)
      do k = 1,n$
       k$ = k + off$0
       do i = 1,n
        f(i,n,k) = (f(i,n,k) - tot(i,k$))*b(n)
       enddo
      enddo
     end
```

Figure 9.6 ERLEBACHER: Solution Phase in Y Dimension

that each processor computes part of every reduction in the multi-reduction, then inserts a global collective communication routine to accumulate the results. ERLEBACHER performs 2D SUM multi-reductions along each dimension of a 3D array for each of its three computation wavefronts. Consider statement S_1 in Figure 9.5, which performs a SUM multi-reduction in the Z dimension. Because this dimension is distributed, the compiler partitions the computation based on f, the distributed *rhs*, and inserts a call to *global-sum* to accumulate the results.

If the multi-reduction does not cross any distributed dimensions, no information is transferred across distributed dimensions. A processor can thus evaluate some of the reductions comprising the multi-reduction using local data. This case occurs in the solution step in the X and Y dimensions in ERLEBACHER. Simple loop bounds reduction is sufficient to partition the reduction; no communication is needed. If all results are needed, a global *concatenation* routine can be called to collect the results from each processor.

Array Kills

For instance, a multi-reduction is performed in the Y dimension solution step of ERLEBACHER, shown in Figure 9.6. Because the Y dimension of f is local, relaxing the owner computes rule allows each processor to compute its reductions locally. Unfortunately the multi-reduction is being computed for *tot*, a replicated array. The compiler thus inserts a global concatenation routine to collect values of *tot* from other processors. This concatenation is the only communication inserted in the computation sweeps in the X and Y dimensions, and turns out to be unnecessary. Array kill analysis would show that the values of *tot* only reach uses in the next loop nest S_2, where it is used only on iterations executed locally. Values for *tot* not computed locally are not ever used. This information can be employed to eliminate the unnecessary global concatenation.

Exploiting Pipeline Parallelism

Finally, because the computational wavefront traverses across processors in the Z dimension, the Fortran D compiler must efficiently exploit pipeline parallelism. In Figure 9.5, the compiler detects that the k loop enclosing statement S_4 is a cross-processor loop because it carries a true dependence whose endpoints are on different processors. To exploit pipeline parallelism, the compiler interchanges k innermost, then strip-mines the enclosing i loop to reduce the communication overhead. Note that the nonlocal reference to $f(i,j,n)$ has also been converted to a vectorized broadcast. The compiler replaced the reference with r\$buf1$(j*n+i-n)$ to properly access data in the buffer array.

9.4 Empirical Evaluation of the Fortran D Compiler

To evaluate the status of the current Fortran D compiler prototype, the output of the Fortran D compiler is compared with hand-optimized programs on the Intel iPSC/860 and the output of the CM Fortran compiler on the TMC CM-5. Our goal is to validate our compilation approach and identify directions for future research. In many cases, problems sizes were too large to be executed sequentially on one processor. In these cases sequential execution times are estimates, computed by projecting execution times for smaller computations to the larger problem sizes. Empirical results are presented in both tabular and graphical form.

9.4.1 Comparison with Hand-Coded Kernels

We begin by comparing the output of the Fortran D compiler against hand-coded stencil kernels on the Intel iPSC/860 hypercube. Our iPSC timings were obtained on the 32 node Intel iPSC/860 at Rice University. It has 8 Meg of memory per node and is running under Release 3.3.1 of the Intel software. Each program was compiled under -O4 using Release 3.0 of *if77*, the iPSC/860 compiler. Timings were made using *dclock*(), a microsecond timer.

Program	Problem Size	Proc	Time		Speedup		Hand-Coded / Fortran D
			FortD	Hand	FortD	Hand	
Livermore 3 *Inner Product*	$64K$	16	0.002	0.002	10.4	10.4	1.0
		32	0.002	0.002	10.4	10.4	1.0
	$256K$	16	0.006	0.006	13.3	13.3	1.0
		32	0.004	0.004	20.0	20.0	1.0
	$1024K$	16	0.023	0.023	14.9	14.9	1.0
		32	0.012	0.012	28.5	28.5	1.0
Jacobi Iteration	512×512	16	0.027	0.019	12.3	17.5	0.70
		32	0.018	0.011	18.5	30.3	0.61
	$1K \times 1K$	16	0.101	0.089	15.4	17.4	0.88
		32	0.058	0.047	26.7	33.0	0.81
	$2K \times 2K$	16	0.375	0.350	16.4	17.6	0.93
		32	0.198	0.177	31.1	34.8	0.89
Livermore 18 *Explicit Hydrodynamics*	128×128	16	0.027	0.023	8.53	10.0	0.85
		32	0.021	0.017	11.0	13.6	0.81
	256×256	16	0.081	0.070	10.7	12.4	0.86
		32	0.055	0.044	15.8	19.7	0.80
	512×512	16	0.309	0.239	10.9	14.1	0.77
		32	0.182	0.137	18.6	24.6	0.75
Successive Over Relaxation	512×512	16	0.043	0.048	8.56	7.67	1.12
		32	0.041	0.039	8.98	9.44	0.95
	$1K \times 1K$	16	0.106	0.143	16.0	11.9	1.35
		32	0.086	0.096	19.7	17.7	1.12
	$2K \times 2K$	16	0.305	0.493	22.3	13.8	1.62
		32	0.215	0.288	31.7	23.7	1.34
Livermore 23 *Implicit Hydrodynamics*	256×256	16	0.031	0.049	12.4	7.8	1.58
		32	0.030	0.039	12.8	9.8	1.30
	512×512	16	0.085	0.148	18.1	10.4	1.74
		32	0.069	0.099	22.3	15.5	1.43
	$1K \times 1K$	16	0.248	0.507	24.6	12.1	2.04
		32	0.168	0.298	36.4	20.5	1.77

Table 9.1 Intel iPSC/860 Execution Times for Stencil Kernels (in seconds)

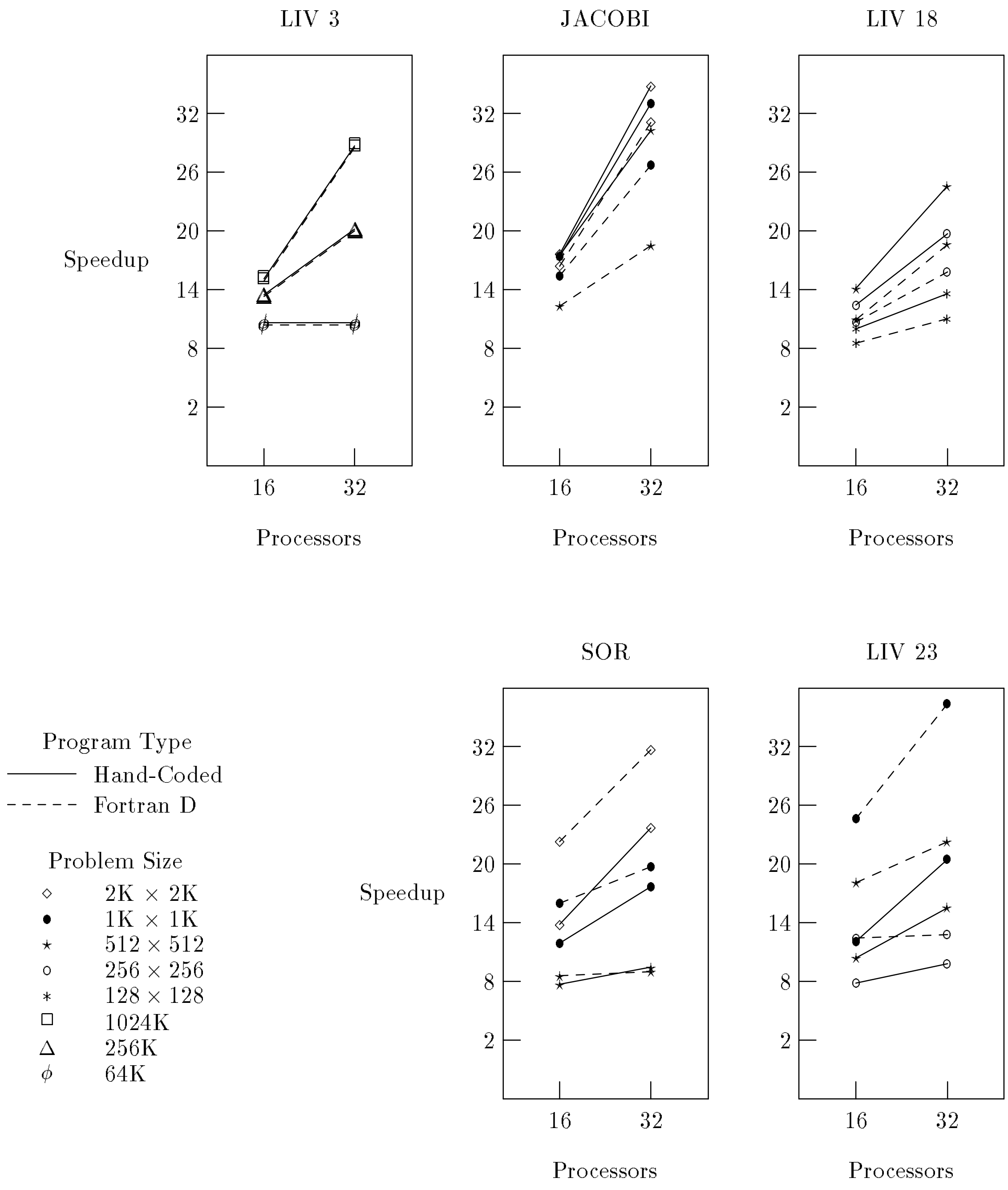


Figure 9.7 Speedups for Stencil Kernels (Intel iPSC/860)

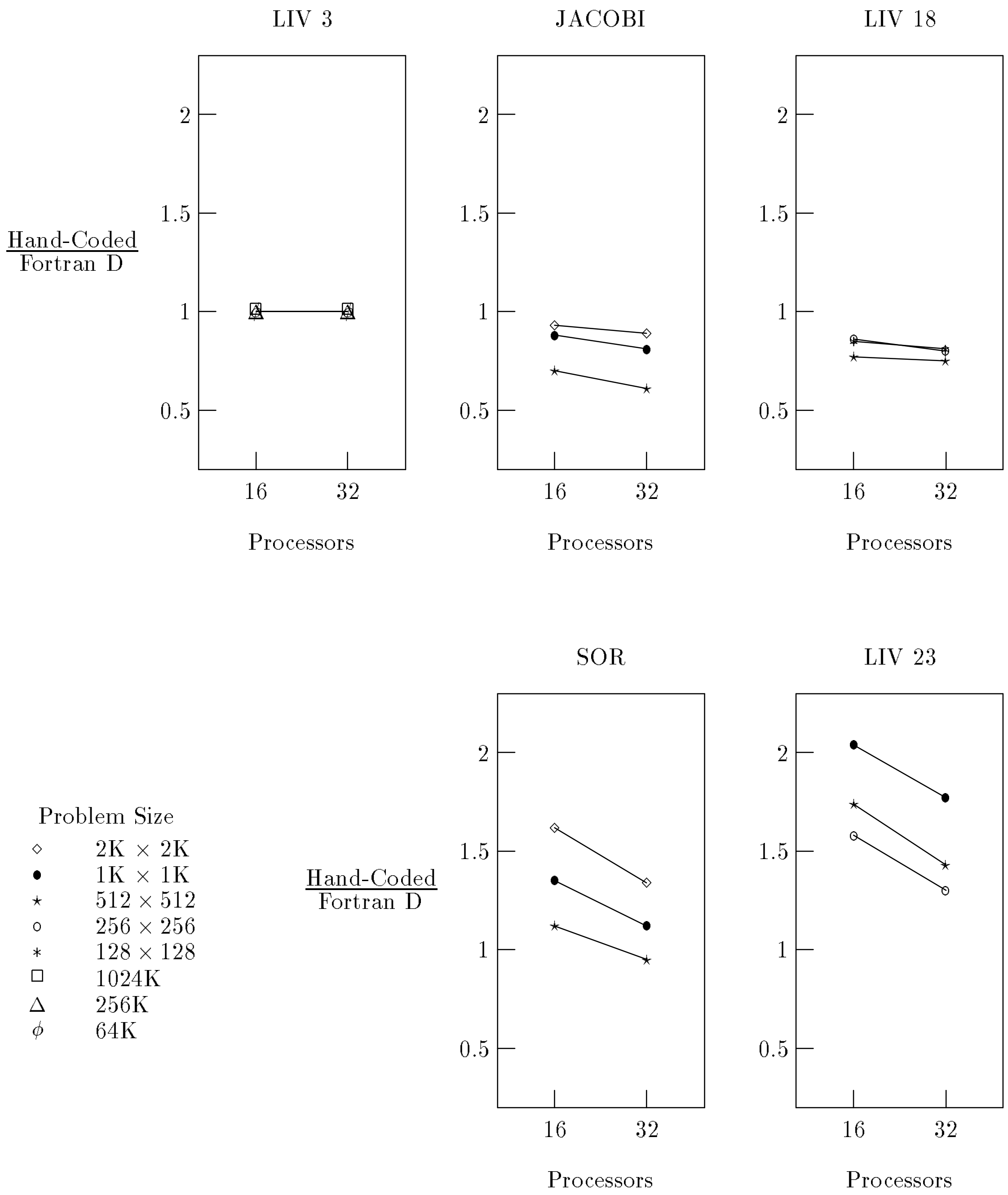


Figure 9.8 Comparisons for Stencil Kernels (Intel iPSC/860)

We first measure the output of the Fortran D compiler against the hand-optimized stencil kernels studied in Chapter 5. We selected a sum reduction (Livermore 3), two parallel kernels (Livermore 18, Jacobi), and two pipelined kernels (Livermore 23, SOR). As before, all arrays are double precision and distributed block-wise in one dimension.

Execution times and speedups for different problem and machine sizes are shown in Table 9.1. Figure 9.7 displays speedups graphically, with speedups plotted along the Y-axis and number of processors along the X-axis. Solid and dashed lines correspond to speedups for hand-coded and Fortran D programs, respectively. Each line represents the speedup for a given problem size. Figure 9.8 compares the ratio of execution times between the hand-coded and Fortran D versions of each kernel. Each line represents the ratio for a given problem size.

We found that the code generated for the inner product in Livermore 3 were identical to the hand-coded versions, since the compiler recognized the sum reduction and used the appropriate collective communication routine. For parallel kernels, the output of the Fortran D compiler was within 50% of the best hand-optimized codes. The deficit was mainly caused by the Fortran D compiler not exploiting unbuffered messages in order to eliminate buffering and overlap communication overhead with local computation. The compiler-generated code actually outperformed the hand-coded pipelined codes, probably due to complications with the scalar i860 node compiler in the parameterized hand-coded version.

9.4.2 Comparison with Hand-Coded Programs

We now turn our attention to evaluating the performance of the Fortran D compiler for large subroutines and application codes. In the following sections, we display speedups and comparisons for SHALLOW and the three other codes studied, both in tables and graphically.

Figure 9.9 displays speedups for each program graphically, with speedups plotted along the Y-axis and number of processors along the X-axis. Solid and dashed lines correspond to speedups for hand-coded and Fortran D programs, respectively. Each line represents the speedup for a given problem size. Figure 9.10 compares the ratio of execution times between the hand-coded and Fortran D versions of each program. Each line represents the ratio for a given problem size.

Results for SHALLOW

Table 9.2 contains timings for performing one time step of SHALLOW. It presents speedups, calculated as $\frac{\text{seq}}{\text{time}}$, as well as the ratio of execution times between hand-coded and Fortran D versions of the program. We found the program to be ideal for distribute-memory machines. Computation is entirely data-parallel, with nearest-neighbor communication taking place between phases of each time step. The compiler output achieved excellent speedups (21–29), even for smaller problems. To evaluate potential improvements, we performed aggressive inter-loop message coalescing and aggregation by hand, halving the total number of messages. The hand-coded versions of SHALLOW exhibited only slight improvements (1–10%) over the compiler-generated code, except when small problems were parallelized on many processors (12–26%). Communication costs apparently only contributed to a small percentage of total execution time, reducing the impact and profitability of advanced communication optimizations.

Results for DISPER

Like SHALLOW, DISPER is a completely data-parallel computation that requires only nearest-neighbor communications. Timings for DISPER in Table 9.3 show near-linear speedups for the output of the Fortran D compiler, once errors introduced by execution conditions were corrected. We also created a hand-coded version of DISPER by applying aggressive inter-loop message message aggregation. The resulting message was large enough that it became profitable to also employ unbuffered *isend* and *irecv* messages. However, since communication overhead is small, the hand-coded version only yielded minor improvements (1-3%) for the single problem size tested.

Problem Size	Proc	Fortran D time	Fortran D $\frac{\text{seq}}{\text{time}}$	Hand-Coded time	Hand-Coded $\frac{\text{seq}}{\text{time}}$	$\frac{\text{Hand-Coded}}{\text{Fortran D}}$
256×256	1	*sequential time = 0.728*				
	2	0.354	2.06	0.348	2.09	0.98
	4	0.195	3.73	0.188	3.87	0.96
	8	0.097	7.50	0.091	8.00	0.94
	16	0.056	13.0	0.049	14.86	0.88
	32	0.035	20.8	0.026	28.00	0.74
512×512	1	*estimated sequential time = 2.9*				
	2	1.529	1.90	1.521	1.91	0.99
	4	0.707	4.10	0.698	4.15	0.99
	8	0.377	7.69	0.368	7.88	0.98
	16	0.201	14.43	0.191	15.18	0.95
	32	0.107	27.10	0.095	30.53	0.89
$1K \times 1K$	1	*estimated sequential time = 11.6*				
	8	1.620	7.16	1.610	7.20	0.99
	16	0.755	15.36	0.739	15.70	0.98
	32	0.397	29.22	0.380	30.53	0.95

Table 9.2 Intel iPSC/860 Execution Times for SHALLOW (in seconds)

Problem Size	Proc	Fortran D time	Fortran D $\frac{\text{seq}}{\text{time}}$	Hand-Coded time	Hand-Coded $\frac{\text{seq}}{\text{time}}$	$\frac{\text{Hand-Coded}}{\text{Fortran D}}$
$256 \times 8 \times 8 \times 4$	1	*estimated sequential time = 39.0*				
	4	9.971	3.91	10.222	3.81	1.03
	8	5.040	7.74	4.979	7.83	0.99
	16	2.440	15.98	2.414	16.16	0.99
	32	1.284	30.37	1.240	31.45	0.97

Table 9.3 Intel iPSC/860 Execution Times for DISPER (in seconds)

Problem Size	Proc	Fortran D time	Fortran D $\frac{seq}{time}$	Hand-Coded time	Hand-Coded $\frac{seq}{time}$	$\frac{\text{Hand-Coded}}{\text{Fortran D}}$
256×256	1	*sequential time = 2.151*				
	2	1.051	2.05	1.108	1.94	1.05
	4	0.744	2.89	0.683	3.15	0.92
	8	0.670	3.21	0.551	3.90	0.82
	16	0.695	3.09	0.644	3.34	0.93
	32	0.782	2.75	0.758	2.84	0.97
512×512	1	*sequential time = 17.53*				
	2	7.988	2.19	7.879	2.22	0.99
	4	4.786	3.66	4.322	4.06	0.90
	8	3.373	5.20	2.601	6.74	0.77
	16	2.908	6.03	2.259	7.76	0.78
	32	2.916	6.01	2.619	6.69	0.90
$1K \times 1K$	1	*estimated sequential time = 140*				
	2	66.74	2.10	68.91	2.03	1.03
	4	36.29	3.86	35.61	3.93	0.98
	8	21.83	6.41	18.93	7.40	0.87
	16	15.32	9.14	10.97	12.76	0.72
	32	12.96	10.80	9.654	14.50	0.74
$2K \times 2K$	1	*estimated sequential time = 1120*				
	8	160.45	6.98	145.83	7.68	0.91
	16	97.22	11.52	76.28	14.68	0.78
	32	68.86	16.26	44.62	25.10	0.65

Table 9.4 Intel iPSC/860 Execution Times for DGEFA (in seconds)

Problem Size	Proc	Fortran D time	Fortran D $\frac{seq}{time}$	Hand-Coded Array Kill time	Array Kill $\frac{seq}{time}$	Hand-Coded Pipelining time	Pipelining $\frac{seq}{time}$	Hand-Coded Memory time	Memory $\frac{seq}{time}$	$\frac{\text{Hand-Coded}}{\text{Fortran D}}$
$64 \times 64 \times 64$	1	*sequential time = 1.577*								
	2	1.104	1.43	1.071	1.47	1.051	1.50	0.805	1.96	0.73
	4	0.765	2.06	0.726	2.17	0.630	2.50	0.586	2.69	0.77
	8	0.657	2.40	0.599	2.63	0.452	3.49	0.448	3.52	0.68
	16	0.539	2.93	0.427	3.69	0.312	5.05	0.311	5.07	0.53
	32	0.613	2.57	0.461	3.43	0.314	5.02	0.315	5.00	0.51
$96 \times 96 \times 96$	1	*estimated sequential time = 5.3*								
	4	1.677	3.17	1.590	3.33	1.311	4.06	1.151	4.60	0.70
	8	1.475	3.61	1.312	4.04	0.961	5.54	0.917	5.78	0.62
	16	1.492	3.57	1.189	4.46	0.824	6.46	0.813	6.52	0.54
	32	1.355	3.93	1.059	5.00	0.741	7.15	0.720	7.36	0.54
$128 \times 128 \times 128$	1	*estimated sequential time = 12.6*								
	8	3.341	3.77	3.101	4.06	2.508	5.03	1.905	6.61	0.59
	16	2.997	4.21	2.528	4.98	1.876	6.72	1.584	7.95	0.56
	32	2.683	4.70	2.146	5.87	1.497	8.42	1.347	9.35	0.50

Table 9.5 Intel iPSC/860 Execution Times for ERLEBACHER (in seconds)

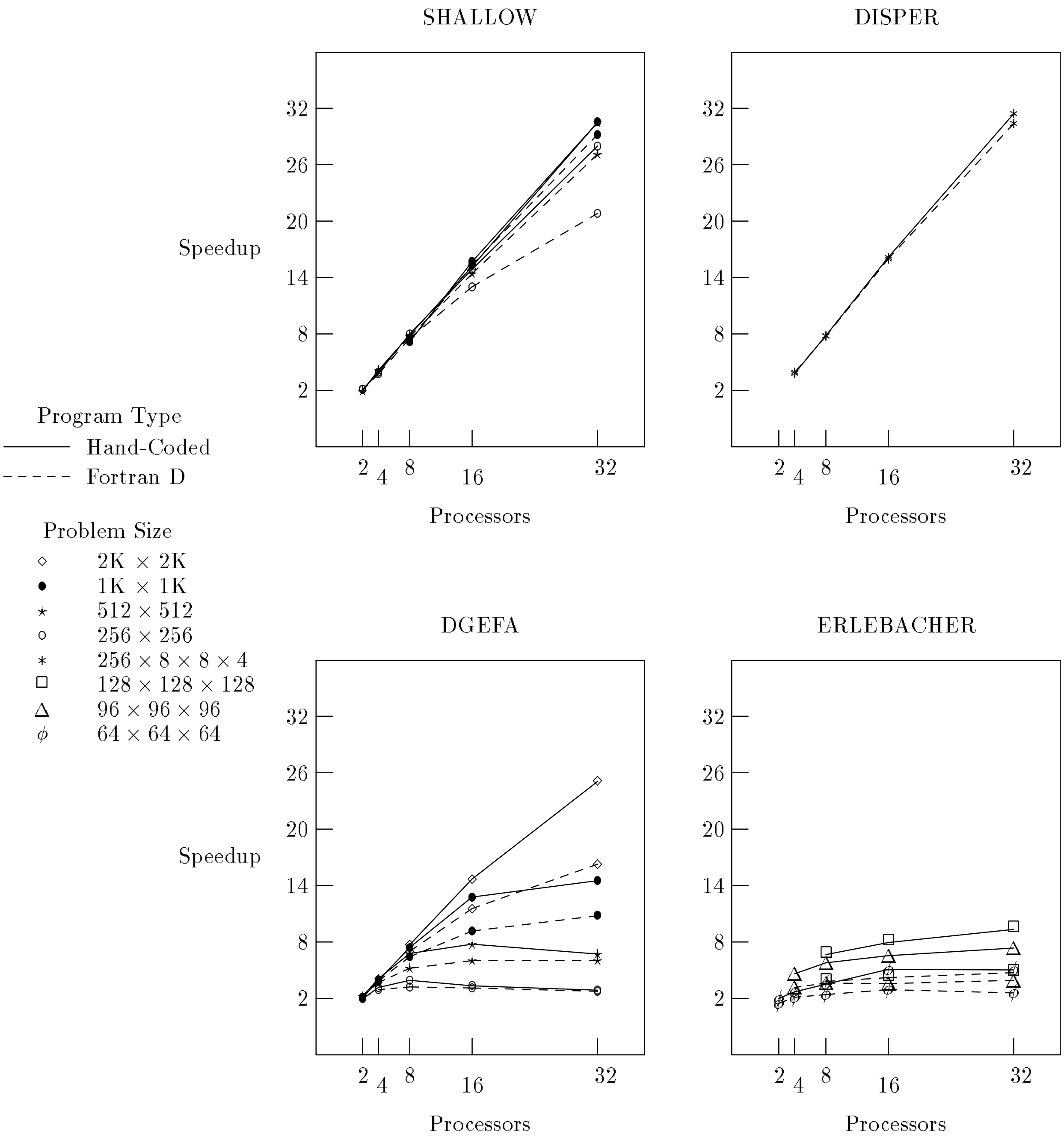


Figure 9.9 Speedups for Programs (Intel iPSC/860)

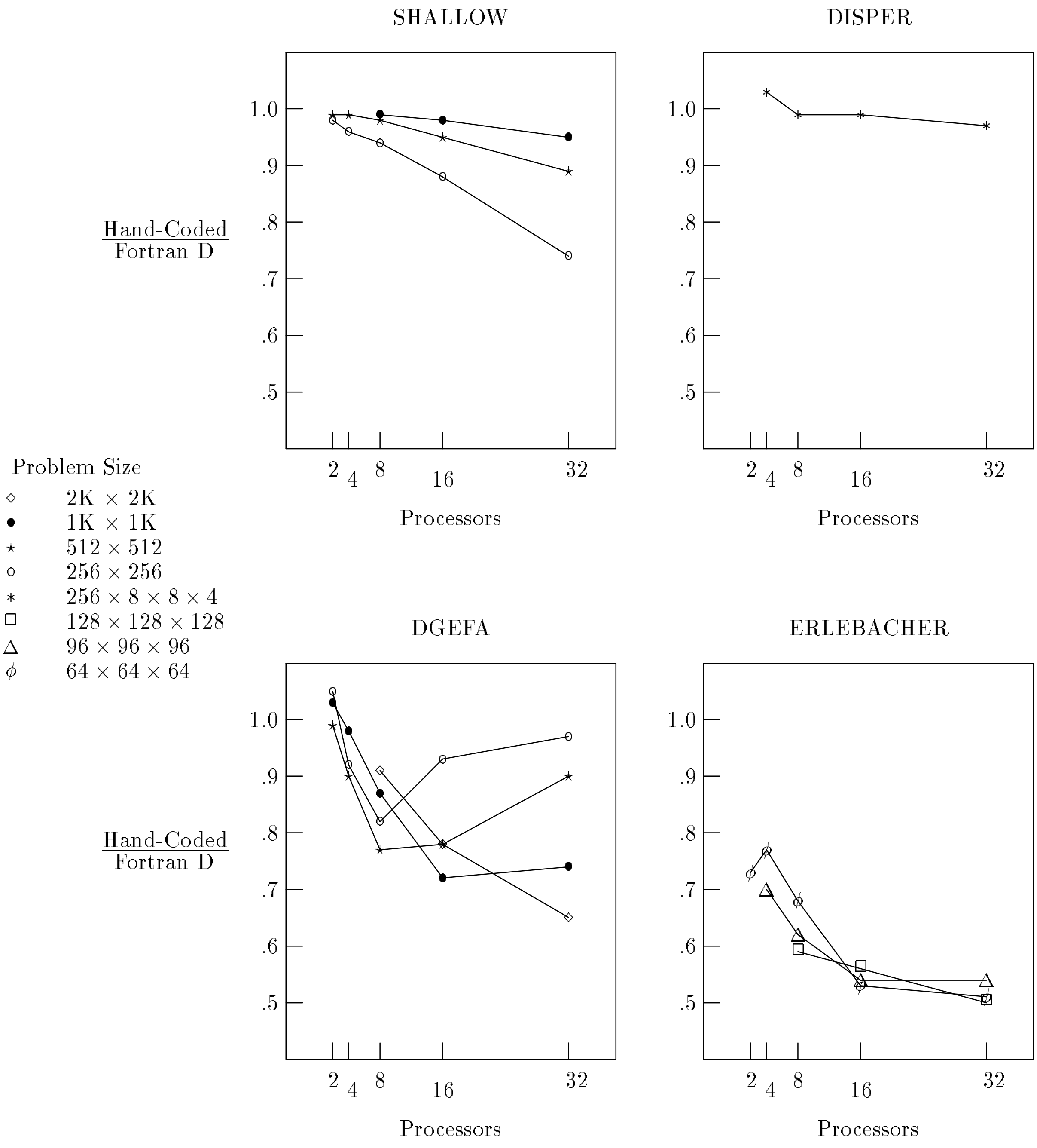


Figure 9.10 Comparisons for Programs (Intel iPSC/860)

Results for DGEFA

Table 9.4 presents execution times and speedups for DGEFA, Gaussian elimination with partial pivoting. Results indicate that the Fortran D compiler output, shown in Figure 9.1, provided limited speedups (3–6) on small problems. For larger problems moderate speedups (11–16) were achieved. Due to the large number of global broadcasts required to communicate pivot values and multipliers, performance of DGEFA actually degrades when solving small problems on many processors.

To determine whether improved performance is attainable, we created a hand-coded version of DGEFA based on optimizations described in the literature [78, 157]. First, we combined the two messages broadcast on each iteration of the outermost k loop. Instead of broadcasting the pivot value immediately, we wait until multipliers are also computed. The values can then be combined in one broadcast. Overcommunication may result when a zero pivot is found, since messages now include multipliers even if they are not used. However, combining broadcasts is still profitable as zero pivots occur rarely.

Second, we restructured the computation so that upon receiving the pivot for the current iteration, the processor P_{k+1} responsible for finding the pivot for the next iteration does so immediately. P_{k+1} performs row elimination on just the first column of the remaining subarray, scans that column to find a pivot and calculates multipliers. P_{k+1} then broadcasts the pivot and multipliers to the other processors before performing row elimination on the remaining subarray. Since row eliminations make up most of the computation in Gaussian elimination, each broadcast in effect takes place one iteration ahead of the matching receive, hiding communication costs by overlapping message latency with local computation.

Results for the hand-coded version of DGEFA are presented in Table 9.4. The new algorithm showed little or no improvement for small problems or when few processors were employed. However, it increased performance by over 30% for large problems on many processors, yielding decent speedups (14–25). The Fortran D compiler can thus benefit from more aggressive optimization of linear algebra routines. Experience also indicates that programmers can achieve higher performance for linear algebra codes with block versions of algorithms. The Fortran D compiler will need to provide BLOCK_CYCLIC data distributions to support these block algorithms.

Results for ERLEBACHER

Unlike SHALLOW and DISPER, ERLEBACHER is not fully data-parallel. It is a more complex program that requires global communication, and also contains computation wavefronts that sequentialize parts of the computation. For ERLEBACHER, the Fortran D compiler first performs interprocedural reaching decomposition and overlap analysis, then invokes local code generation for each procedure. The compiler inserts global communication for array SUM reductions, and also applies coarse-grain pipelining. Timings for ERLEBACHER in Table 9.5 show that the compiler-generated code is rather inefficient, with speedup peaking at 3–5 even for large programs.

To determine how much improvement is attainable, we applied additional optimizations to create three hand-coded versions. Optimizations are cumulative from left to right, so each hand-coded program contains optimizations applied in the previous version. In the "Array Kill" version we used interprocedural array kill analysis to eliminate global concatenation for local multi-reductions on replicated arrays in the X and Y sweeps. In the "Pipelining" version we also experimented with the granularity of coarse-grain pipelining performed during forward and backward substitution in the Z sweep. We found that a strip size around 16 yielded significantly better performance than the default strip size of 8 selected by the Fortran D compiler.

Finally, in the "Memory" version we also performed loop interchange to improve the data locality of each node program during forward and backward substitution in the Z sweep. The current algorithm for pipelining in the Fortran D compiler simply interchanges the cross-processor loop innermost, without taking data locality into account. It thus placed the k loop innermost in TRIDVPK. We interchanged the strip-mined i loop innermost by hand, improving data locality by restoring unit-stride memory accesses. The two versions are shown in Figure 9.11.

Timings show that all three optimizations contribute to improved performance. Using array kill information and adjusting pipelining granularity reduced communication costs, especially when many processors were used. Improving data locality of the node program helped most when few processors were used and large data sizes caused many cache misses. Together these optimizations yielded speedups of 5–9, improving performance by up to 50% over the Fortran D compiler-generated code.

```
{* Compiler Output for 4 Processors *}
SUBROUTINE TRIDVPK
 ...
 do j = 1,n
  do i$ = 1,n,8
   i$up = i$+7
   if (my$p .LT. 3)
    recv f(i$:i$up, j, n$+1) from my$p+1
   do i = i$,i$+8
    do k = ub$,1,-1
     k$ = k + off$0
     f(i,j,k) = f(i,j,k)-c(k$)*f(i,j,k+1)
              - e(k$)*r$buf1(j*n+i-n)
    enddo
   enddo
   if (my$p .GT. 0)
    send f(i$:i$up, j, 1) to my$p-1
  enddo
 enddo
end
```

```
{* Interchange into Memory Order *}
SUBROUTINE TRIDVPK
 ...
 do j = 1,n
  do i$ = 1,n,8
   i$up = i$+7
   if (my$p .LT. 3)
    recv f(i$:i$up, j, n$+1) from my$p+1
   do k = ub$,1,-1
    k$ = k + off$0
    do i = i$,i$+8
     f(i,j,k) = f(i,j,k)-c(k$)*f(i,j,k+1)
              - e(k$)*r$buf1(j*n+i-n)
    enddo
   enddo
   if (my$p .GT. 0)
    send f(i$:i$up, j, 1) to my$p-1
  enddo
 enddo
end
```

Figure 9.11 ERLEBACHER: Data Locality Optimization

9.4.3 Comparison with CM Fortran Compiler

We also evaluated the performance of the Fortran D compiler against a commercial compiler. We selected the CM Fortran compiler, the most mature and widely used compiler for MIMD distributed-memory machines, and compared it against the Fortran D compiler on the Thinking Machines CM-5.

Our CM-5 timings were obtained on the 32 node CM-5 at Syracuse University. It has Sun Sparc processors running SunOS 4.1.2 and vector units running CMOST 7.2 S2. CM Fortran programs were compiled using *cmf* version 2.0 beta, with the -O and -vu flags. They were timed using *CM_timer_read_elapsed*(). CM Fortran programs were compared against message-passing Fortran 77 programs using CMMD version 2.0 beta, the CM message-passing library. Fortran 77 node programs were compiled using the Sun Fortran compiler *f77*, version 1.4, with the -O flag. They were linked with *cmmd* version 2.0 beta. Fortran 77 node programs were timed using *CMMD_node_timer_elapsed*().

Results for Kernels and Programs

The output of the Fortran D compiler was easily ported to the CM-5 by replacing calls to Intel NX/2 message-passing routines with equivalent calls to TMC CMMD message-passing routines. We converted program kernels into CM Fortran by hand for the CM Fortran compiler, inserting the appropriate LAYOUT directives to achieve the same data decomposition. The inner product in Livermore 3 was replaced by DOTPRODUCT, a CM Fortran intrinsic. Jacobi and Livermore 18 can be transformed directly into CM Fortran. Loop skew and interchange must be applied to SOR and Livermore 23 to expose parallelism in the form of FORALL loops. A mask array *indx* is used to implement Gaussian elimination. The resulting CM Fortran kernels are shown in Figure 9.12.

```
{* Livermore 3 (Inner Product) *}
  s = DOTPRODUCT(a,b)

{* Jacobi *}
  forall (j=2:N-1,i=2,N-1)
    a(i,j) = 0.25*(b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
  b = a

{* Livermore 18 (Explicit Hydrodynamics) *}
  forall (k=2:N-1, j=2:N-1)
    za(j, k) = (zp(j-1, k+1) + zq(j-1, k+1) - zp(j-1, k) - zq(j-1, k))
        * (zr(j, k) + zr(j-1, k)) / (zm(j-1, k) + zm(j-1, k+1))
  forall (k=2:N-1, j=2:N-1)
    zb(j, k) = (zp(j-1, k) + zq(j-1, k) - zp(j, k) - zq(j, k)
        * (zr(j, k) + zr(j, k-1)) / zm(j, k) + zm(j-1, k))
  forall (k=2:N-1, j=2:N-1)
    zu(j, k) = zu(j, k) + s * (za(j, k) * (zz(j, k) - zz(j+1, k))
        - za(j-1, k)* (zz(j, k) - zz(j-1, k)) - zb(j, k)
        * (zz(j, k) - zz(j, k-1)) + zb(j, k+1) * (zz(j, k) - zz(j, k+1)))
  forall (k=2:N-1, j=2:N-1)
    zv(j, k) = zv(j, k) + s * (za(j, k) * (zr(j, k) - zr(j+1, k))
        - za(j-1, k) * (zr(j, k) - zr(j-1, k)) - zb(j, k+1) * (zr(j, k)
        - zr(j, k-1)) + zb(j, k+1) * (zr(j, k) - zr(j, k+1)))
  forall (k=2:N-1, j=2:N-1)
    zr(j, k) = zr(j, k) + t * zu(j, k)
  forall (k=2:N-1, j=2:N-1)
    zz(j, k) = zz(j, k) + t * zv(j, k)

{* SOR *}
  do j = 4,2*(N-1)
    forall (i=max(2,j-N+1):min(N-1,j-2))
      a(i,j-i) = 0.175*(a(i-1,j-i)+a(i+1,j-i)+a(i,j-i-1)
        + a(i, j-i+1)) + 0.3 * a(i, j-i)
  enddo

{* Livermore 23 (Implicit Hydrodynamics) *}
  do j = 4,2*(N-1)
    forall (k=max(2,j-N+1):min(N-1,j-2))
      za(k,j-k) = za(k,j-k)+.175*((za(k,j-k+1)*zr(k,j-k)
          * zb(k, j-k) + za(k+1,j-k)*zu(k,j-k)
          + za(k-1, j-k) * zv(k, j-k)) - za(k,j-k))
  enddo

{* Gassian Elimination *}
  indx = 0
  do k = 1,N
    iTmp = maxloc(abs(a(:,k)), MASK = indx .EQ. 0)
    indxRow = iTmp(1)
    maxNum = a(indxRow,k)
    indx(indxRow) = k
    fac = a(:,k) / maxNum
    row = a(indxRow,:)
    forall (i = 1:N, j = 1:N+1, (indx(i).EQ.0) .AND. (j.GE.k))
      a(i,j) = a(i,j) - fac(i) * row(j)
  enddo
```

Figure 9.12 CM Fortran Versions of Kernels

Currently, only the CM Fortran compiler generates code utilizing vector units. Since the node Fortran 77 compiler is not able to utilize vector units, Fortran D message-passing programs are forced to rely on the Sparc processor. To permit a balanced comparison, we provide timings for CM Fortran programs using either Sparc or vector units. Table 9.6 shows the elapsed times we measured on the CM-5 for CM Fortran and Fortran D programs, as well as the ratio of execution times between CM Fortran and Fortran D code.

In addition, we graphically present the execution times measured on the CM-5. Figure 9.13 displays timings for stencil kernels. Execution times in seconds are plotted logarithmically along the Y-axis. The problem size is plotted logarithmically along the X-axis. Solid, dotted, and dashed lines represent the CM Fortran using Sparc, CM Fortran using vector units, and Fortran D using Sparc, respectively. Execution times for sequential execution on a single Sparc processor are included for comparison. All parallel execution times are for 32 processors. Figure 9.14 directly displays the ratio of execution times of both versions of CM Fortran to Fortran D, plotting ratios along the Y-axis. Figure 9.15 displays similar comparisons of execution times and ratios for SHALLOW and DGEFA.

Our measurements indicate the current CM Fortran compiler produces code that is significantly slower than the corresponding message-passing programs generated by the Fortran D compiler. The difference is especially pronounced for small data sizes. Even intrinsic functions such as DOTPRODUCT yield very poor performance. The CM Fortran compiler fared best on data-parallel kernels such as Jacobi and Livermore 18. It appears to handle pipelined computations and Gaussian elimination poorly, even when expressed in a form that contains vector parallelism.

Based on examining the assembly code output of the CM Fortran compiler, we believe the poor performance is due to the fact that the current CM Fortran compiler generates code for executing virtual processes on each node. This mode of execution requires extensive run-time calculation of addresses and results in much unnecessary data movement. Improvements in an upcoming release of the CM Fortran compiler will allow more meaningful comparisons in the future.

Program	Problem Size	Sequential Execution Sparc	Fortran D + CMMD Sparc	CM Fortran Sparc	CM Fortran Vector	CM Fortran / Fortran D Sparc	CM Fortran / Fortran D Vector
Livermore 3	$64K$	0.005	0.002	0.018	0.005	9.92	3.19
Inner	$256K$	0.020	0.007	0.032	0.006	4.67	0.88
Product	$1024K$	0.079	0.027	0.098	0.007	3.63	0.27
Jacobi	512×512	0.877	0.085	0.236	0.079	2.78	0.95
Iteration	$1K \times 1K$	3.525	0.103	0.766	0.133	7.44	1.29
	$2K \times 2K$	14.14	0.409	2.834	0.288	6.93	0.70
Livermore 18	128×128	0.457	0.028	0.165	0.100	5.89	3.57
Explicit	256×256	1.861	0.096	0.332	0.148	3.46	1.54
Hydrodynamics	512×512	7.554	0.251	0.994	0.243	3.96	0.97
Successive	512×512	0.376	0.060	17.04	8.684	284	145
Over	$1K \times 1K$	1.519	0.130	116.1	45.42	893	349
Relaxation	$2K \times 2K$	6.134	0.364	209.9	210.0	577	577
Livermore 23	256×256	0.389	0.035	3.704	3.597	106	103
Implicit	512×512	1.562	0.118	19.33	11.57	164	98.1
Hydrodynamics	$1K \times 1K$	6.252	0.320	122.7	71.43	383	223
	256×256	4.791	0.577	11.81	4.006	20.5	6.94
DGEFA	512×512	40.61	2.858	112.5	66.25	39.4	23.2
	$1K \times 1K$	337.1	17.13	1071	165.2	62.5	9.64
	$2K \times 2K$	6809	110.8	8449	1365	76.3	12.3
	256×256	1.297	0.043	0.409	0.553	9.51	12.9
SHALLOW	512×512	5.210	0.153	2.696	0.696	17.6	4.55
	$1K \times 1K$	20.88	0.565	6.425	0.988	11.4	1.75

Table 9.6 TMC CM-5 Execution Times (for 32 processors, in seconds)

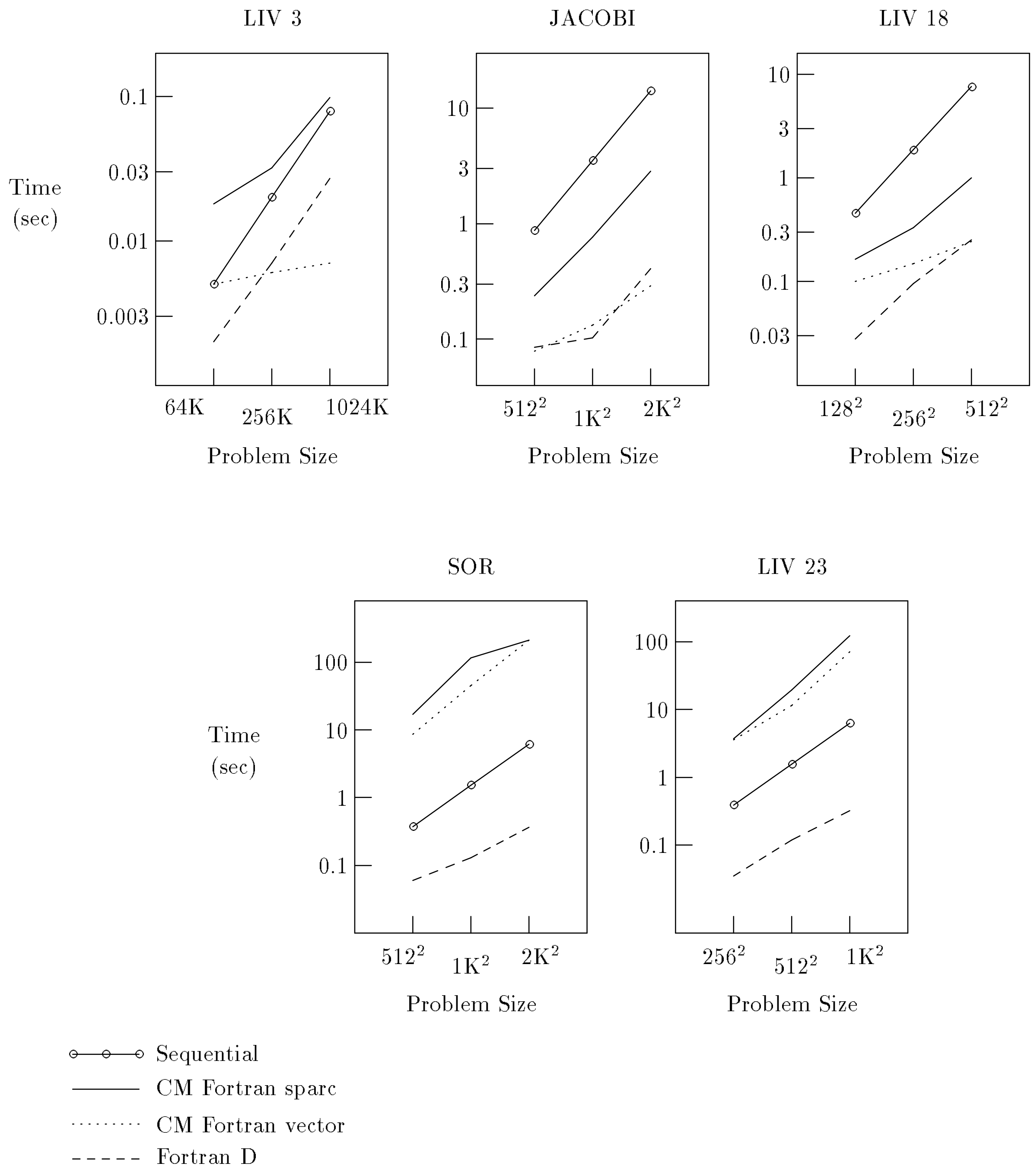


Figure 9.13 Stencil Execution Times (32 Processor Thinking Machines CM-5)

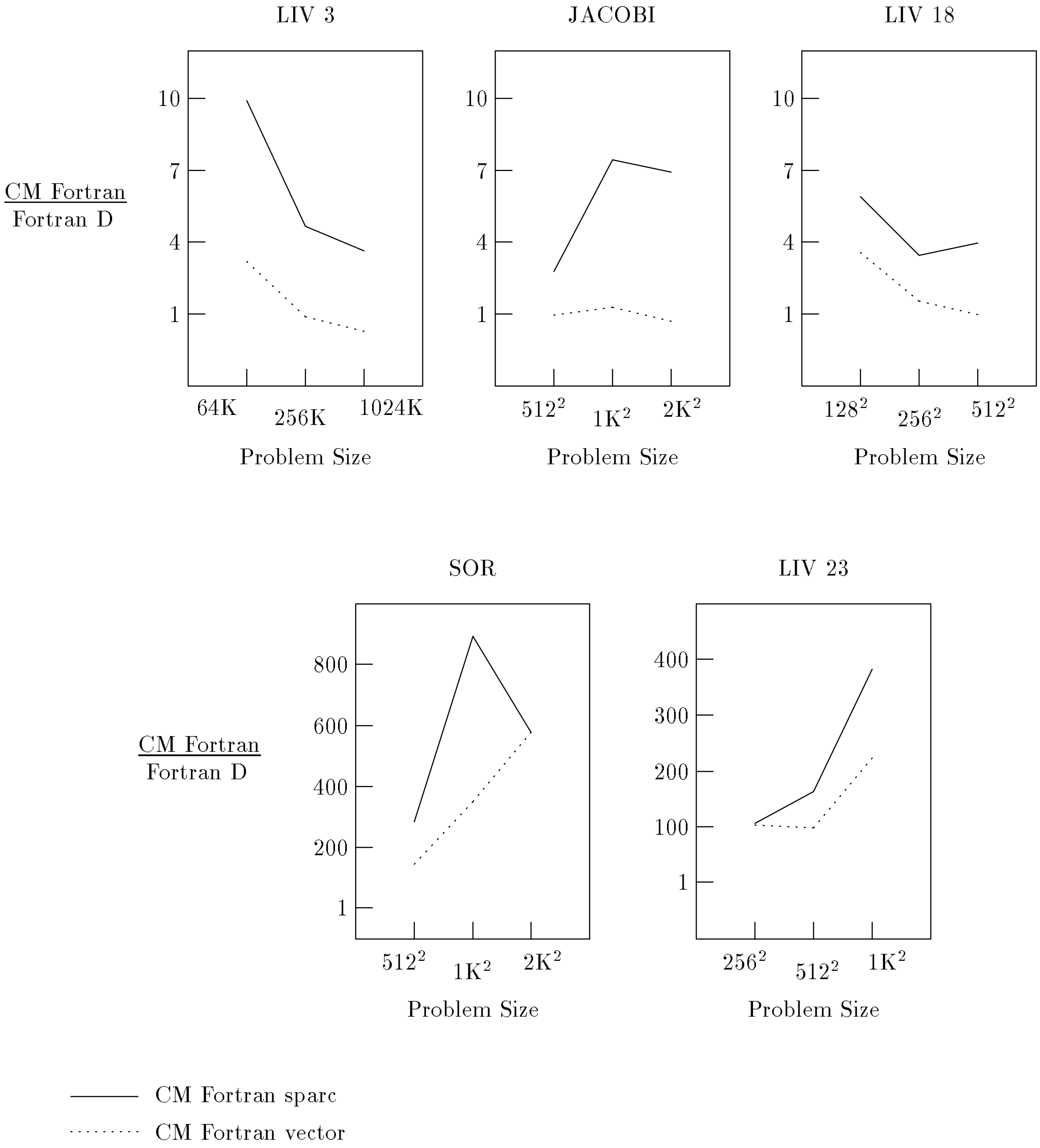


Figure 9.14 Stencil Comparisons (32 Processor Thinking Machines CM-5)

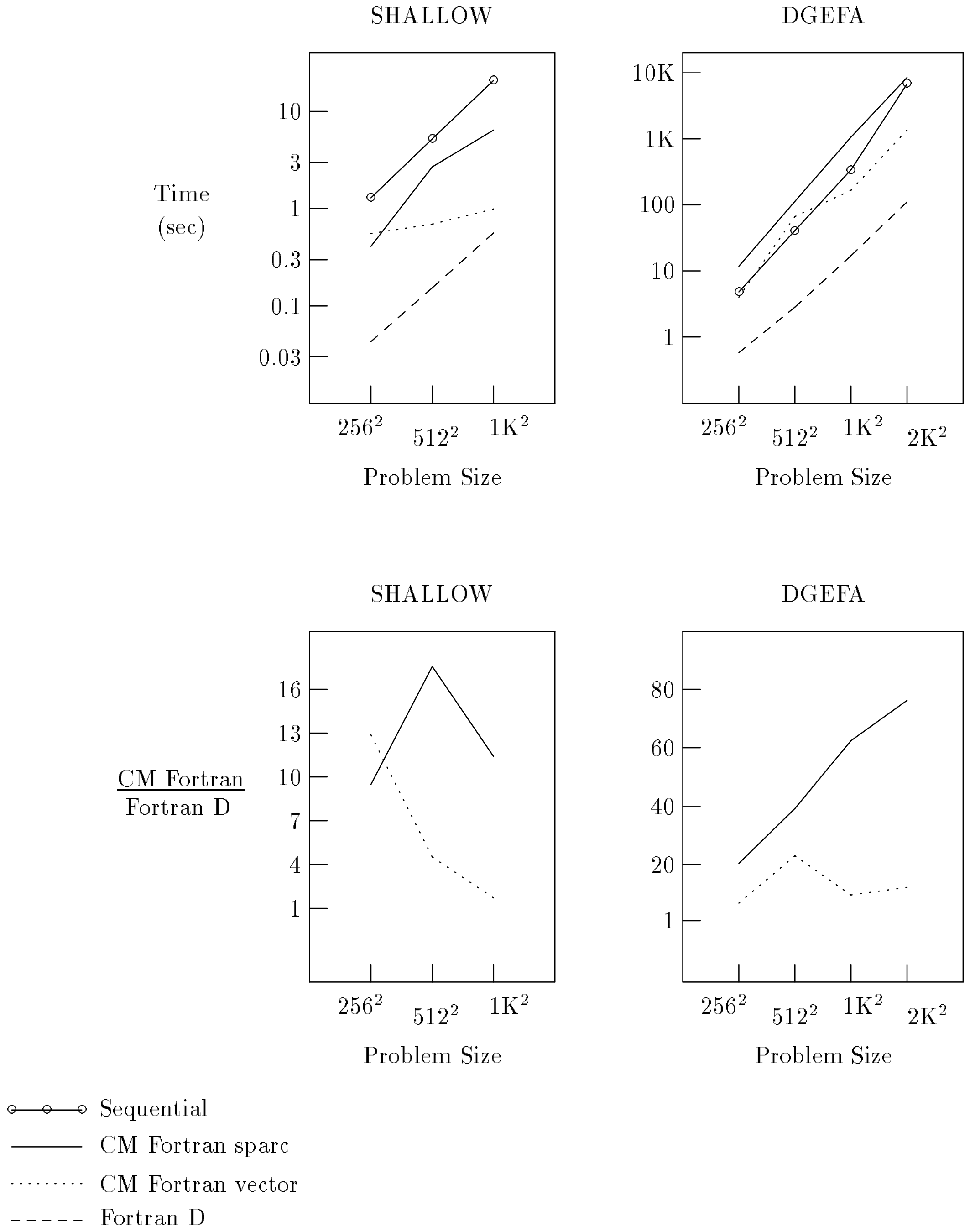


Figure 9.15 Program Execution Times & Comparisons (32 Processor CM-5)

9.5 Status of the Fortran D Compiler

Our preliminary experiences have helped us evaluate the current Fortran D compiler prototype. The initial results, though encouraging, point out a number of areas that require additional work. We summarize our appraisal of the Fortran D compiler with these observations:

- It has achieved considerable success in generating efficient code for stencil computations.
- It needs to improve its optimization of pipelined and linear algebra codes.
- It must become much more flexible before it can become a successful machine-independent programming model. Symbolic information and run-time support must be added.

In addition, our experiences confirm that the nature of the computation is the overriding factor in determining the success of the Fortran D compiler. We discuss each point in greater detail in the following sections.

9.5.1 Parallel Stencil Computations

By generating output for SHALLOW and DISPER that virtually matched optimized hand-coded versions, the Fortran D compiler has demonstrated its success for parallel stencil computations. This is despite the fact that the compiler is not producing the most efficient communication, since it does not yet support unbuffered messages. The Fortran D compiler succeeds because it does a sufficiently good job that communication costs become a minor part of the overall execution time. In particular, scalability is excellent because performance improves as the problem size increases. Implementing additional optimizations is desirable for achieving good speedup for small programs or many processors, but is not crucial. Instead, the focus should be on improving the flexibility and robustness of the Fortran D compiler, as discussed in section 9.5.3.

9.5.2 Pipelined and Linear Algebra Computations

In comparison, there is considerable room for improvement when compiling communication-intensive codes such as pipelined and linear algebra computations. Results for DGEFA and ERLEBACHER show that the current Fortran D compiler prototype only attains limited speedups. It can achieve noticeable performance gains by applying advanced communication optimizations. These optimizations are important because communication is performed much more frequently than in parallel stencil computations. Their effect on overall execution time gain in importance as the problem size and number of processors increases. In particular, the Fortran D compiler will need to use information from training sets and static performance estimation to select an efficient granularity for coarse-grain pipelining.

9.5.3 Increase Flexibility

Finally, when evaluating its overall performance, we find that the most serious problem facing the prototype Fortran D compiler is its lack of flexibility. In the course of conducting our study, we were unable to apply the Fortran D compiler to a large number of standard benchmark programs, despite the fact they contained dense-matrix computations that should have been acceptable to the compiler. Even programs that were written in a "clean" data-parallel manner required fairly extensive rewriting to eliminate programming artifacts that the prototype proved unable to compile. The causes for this inflexibility can be categorized as follows:

Immature Symbolic and Interprocedural Analysis

The lack of symbolic analysis in the current Fortran D compiler proved to be a major stumbling block. Unlike parallelizing compilers for shared-memory machines, simply providing precise dependence information was insufficient for the Fortran D compiler. The compiler performs deep analysis that requires knowledge of all subscript expressions and loop bounds in the program. For most programs, constant propagation, forward expression folding, and auxiliary induction variable substitution all need to be performed before the Fortran D compiler can proceed.

The current prototype is also inhibited by missing pieces in interprocedural analysis. It does not understand formal parameters that represent subarrays in the calling procedure, multiple entry points, or missing procedures representing calls to system library routines. Both symbolic and interprocedural analysis need to be completed and integrated with the Fortran D compiler before many existing programs can be considered.

Lack of Run-time Support

Another problem with the current compiler prototype is its reliance on compile-time analysis. The only run-time support it requires are routines for packing and unpacking non-contiguous array elements into contiguous message buffers. The compiler attempts to calculate at compile-time all information, including ownership, communication, and partitioning. While this approach is necessary for advanced optimizations and generating efficient code, it limits the Fortran D compiler to computations it can completely analyze.

It turns out real programs contain many components that cannot be easily analyzed at compile-time, such as indirect references, complex control flow, and scalar computations. These occur fairly frequently in initializations and boundary condition calculations. In many cases the Fortran D compiler was forced to abort, despite being able to compile the important kernel computations in the program.

What the Fortran D compiler must provide are methods of utilizing run-time support, trading performance for greater flexibility in non-critical regions of the program. The compiler can either apply run-time resolution or demand more support from the run-time library to calculate ownership, partitioning, and communication at run-time. Since in most cases the code affected is executed infrequently, the expense of run-time methods should not significantly impact overall execution time.

Immature Fortran D Compiler

A major part of the problem lies with the immaturity of the Fortran D compiler itself. There are a number of dense-matrix computations that it is not able to analyze and compile efficiently. For instance, the prototype compiler does not handle non-unit loop steps or subscript coefficients. It is thus unable to compile Red-Black SOR or multigrid computations, both of which possess constant step sizes greater than one. Computations such as Fast Fourier Transform (FFT), linear recurrences, particle-in-cell, finite-element, n-body problems, and banded tridiagonal solvers all possess regular but specialized data access patterns that the Fortran D compiler needs to recognize and efficiently support. In addition, run-time support for irregular and sparse computations must also be added. Only when these obstacles are overcome can the Fortran D compiler serve as a credible general-purpose programming model.

Dusty Decks

Finally, the Fortran D compiler cannot compile a number of "dusty deck" Fortran programs that were originally written for sequential or vector machines. These programs contain programming constructs that the compiler does not understand, such as linearized arrays, loops formed by backward GOTO statements, and storing and using constants in arrays. Dusty deck programs have proven to be very challenging for even shared-memory vectorizing and parallelizing compilers. Because of the deep analysis required, they are even more difficult for distributed-memory compilers. It is not a goal of the Fortran D compiler to be able to automatically parallelize these programs for distributed-memory machines. Requiring users to program in Fortran 90 can help prevent such poor programming practices, and is the approach taken by High Performance Fortran. However, as shown by the poor performance of the CM Fortran compiler, Fortran 90 syntax does not eliminate the need for advanced compile-time analysis and optimization.

9.5.4 Nature of Applications

We close by considering implications for future success of the Fortran D approach to data-parallel programming. We believe that problems with the immaturity of symbolic analysis, interprocedural analysis, run-time support, and the Fortran D compiler can be solved in the short term. No breakthroughs are required, simply a major investment in development effort.

Problems with dusty deck codes will remain. Simply adding data decomposition specifications to existing sequential, vector, or parallel programs does not ensure they will be compiled by the Fortran D compiler. Users of massively-parallel machines will eventually recognize that current compiler technology cannot automatically extract parallelism from such codes. They should be willing to rewrite important programs *once* in

Program	Data Size	Computation Amount	Computation Type	Communication Messages	Communication Size	Hand-Coded / Fortran D
DGEFA	$O(n^2)$	$O(n^3)$	Pivot	$O(n)$	$O(n)$	0.65–0.91
ERLEBACHER	$O(n^3)$	$O(n^3)$	Pipeline	$O(n)$	$O(n)$	0.50–0.59
SHALLOW	$O(n^2)$	$O(n^2)$	Parallel	$O(1)$	$O(n)$	0.95–0.99
DISPER	$O(n^4)$	$O(n^4)$	Parallel	$O(1)$	$O(n)$	0.97–1.03

Table 9.7 Inherent Communication and Parallelism in Applications

a "clean" machine-independent form using either Fortran 77 or Fortran 90, if advanced compiler techniques will ensure these programs can be executed efficiently across a wide range of machine architectures.

Compilation technology and dusty deck codes, however, are not the key limitations confronting the Fortran D compiler. Instead, it is the amount of parallelism and communication present in the input Fortran D program. Our experiences show that this is the most significant factor determining the success of the Fortran D compiler. Because the Fortran D compiler does not change the input algorithm or data decomposition, there is an inherent amount of communication and parallelism in a Fortran D program. The nature of the application thus dictates the performance achievable by the Fortran D compiler.

Consider the classification of programs and their communication requirements in Table 9.7. It categorizes the programs we examined by their data, computation, and communication requirements, then compares the effectiveness of the Fortran D compiler against hand-coded versions for the largest problems measured. As the amount of communication increases, the discrepancy between the Fortran D compiler and hand-optimized codes worsens. The table thus clearly demonstrates the correlation between the amount of communication performed and the success of the Fortran D compiler.

Our experiences with the prototype Fortran D compiler leads us to believe that within a few years, compilers for languages such as Fortran D or High Performance Fortran can match the performance of hand-optimized code for applications with high parallelism and low communication. However, prospects for programs with low parallelism or high communication remain unclear. These applications are much more sensitive to communication overhead, and it will be difficult for the compiler to automatically perform the optimizations programmers apply by hand to achieve high performance. Much additional research will be needed to develop automatic compilation techniques for these problems.

9.6 Discussion

This chapter describes compiler techniques developed in response to problems posed by linear algebra computations, large subroutines, and whole programs. The performance of the prototype Fortran D compiler is evaluated against hand-optimized programs on the Intel iPSC/860. Results show reasonable performance is obtained for stencil computations, though room for improvement exists for communication-intensive codes such as linear algebra and pipelined computations. The prototype significantly outperforms the CM Fortran compiler on the CM-5. The compiler requires symbolic analysis, greater flexibility, and improved optimization of pipelined and linear algebra codes. We believe the Fortran D compilation approach will be competitive with hand-coded programs for many data-parallel computations in the near future. However, additional effort is required before the compiler will be as effective for partially parallel computations requiring large amounts of communication.

Chapter 10

Fortran D Programming System

This chapter presents the design of other elements of the Fortran D programming system. A static performance estimator based on training sets provides machine-dependent details that fine-tune the compilation process. The automatic data partitioner derives Fortran D data decomposition specifications based on the original Fortran program. A shared-memory parallelizing and vectorizing compiler exploits multiple processors on a single node. The data locality optimizer restructures the node program for improved use of the memory hierarchy, backed up by an optimizing scalar compiler. A portable lightweight communication library improves performance. The programming environment provides program profiling, performance measurement and visualization, as well as support for debugging and accepting user feedback.

10.1 Introduction

The Fortran D compiler automates the complex task of translating a sequential shared-memory program into an efficient message-passing distributed-memory program. This translation is just the first step in providing a reasonable programming interface for MIMD distributed-memory machines. This chapter describes other important functions provided by the Fortran D programming system:

- Integrate compilers for parallelizing and/or vectorizing the node program produced by the Fortran D compiler, as well as scalar compilers that optimize data locality and instruction-level parallelism.
- Provide feedback on the estimated and actual performance of a Fortran D program for a given machine and problem size.
- Support automatic and interactive data decompositions selection through local and interprocedural analysis.
- Provide an efficient, portable run-time interface for the output of the Fortran D compiler. Utilize lightweight communication primitives to improve performance.
- Integrate tools for performance profiling, visualization, and debugging Fortran D programs.

To provide this functionality, the Fortran D programming system incorporates a number of other tools. Elements of the compilation system include a static performance estimator, automatic data partitioner, parallelizing and vectorizing compiler, data locality optimizer, and optimizing scalar compiler. Additional tools in the programming environment perform program profiling, performance visualization, interactive programming, and debugging. Other tools of the environment include standardized portable communication libraries, lightweight communication protocols. Finally, the Fortran D programming system can be targeted for other architectures such as distributed systems, distributed-shared memory multiprocessors, and heterogeneous systems.

10.2 Fortran D Compilation System

The Fortran D compiler automates most of the translation process required to translate sequential programs for execution on MIMD distributed-memory machines. The only additional information needed by the Fortran D compiler is a specification of the data decomposition. The obvious next step is for the Fortran D system to automatically derive the data decomposition specifications, or interactively assist the user in partitioning the data.

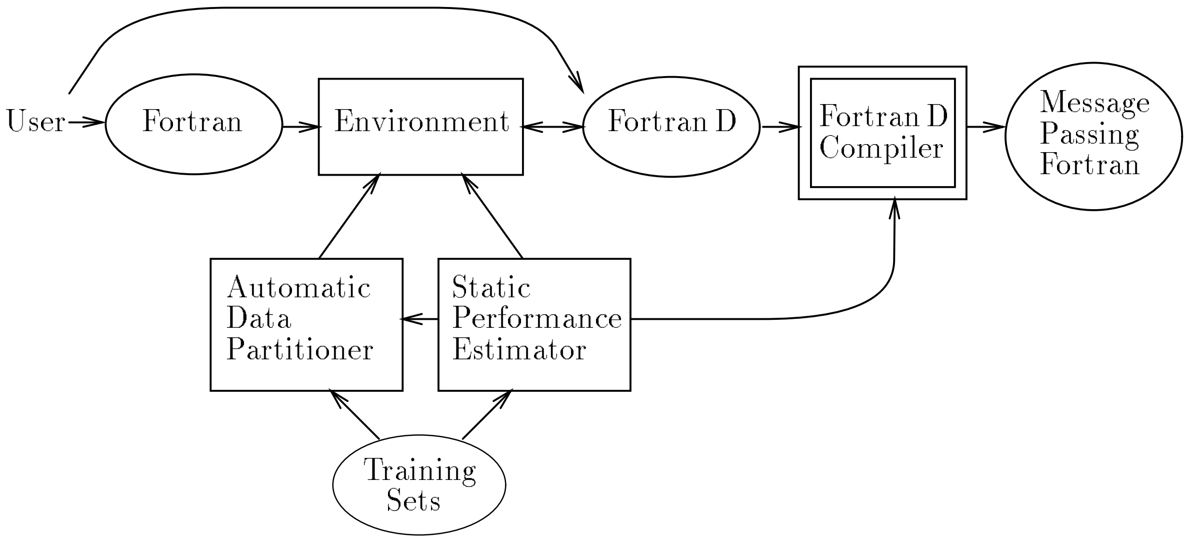


Figure 10.1 Fortran D Automatic Data Partitioner

10.2.1 Static Performance Estimator

The Fortran D static performance estimator is designed to support both automatic and interactive selection of efficient data decompositions. Comparing data decompositions without support from the Fortran D system is an expensive process. The programmer must first insert the appropriate data decomposition specifications in the program text, then compile and run the resulting program to determine its effectiveness. Comparing two data decompositions thus requires implementing and running both versions of the program, a tedious task at best. The process is prohibitively difficult without the assistance of a compiler to automatically generate node programs based on the data decomposition.

It is clearly inefficient to use dynamic performance information to choose between data decompositions. Instead, a *static* performance estimator is needed that can accurately predict the performance of a Fortran D program on the target machine. Also required is a scheme that allows the compiler to assess the costs of communication routines and computations. The static performance estimator in the Fortran D programming system caters to both needs.

The performance estimator in the Fortran D system employs the notion of a *training set* of kernel routines that measures the cost of various computation and communication patterns on the target machine. The results of executing the training set on a parallel machine are summarized and used to train the performance estimator for that machine. By utilizing training sets, the performance estimator achieves both accuracy and portability across different machine architectures [17, 114]. As discussed in Chapter 6, the Fortran D compiler also uses training set information to guide optimizations, particularly balancing communication and parallelism for coarse-grain pipelining.

10.2.2 Automatic Data Partitioner

The goal of the automatic data partitioner is to choose an efficient data decomposition. Several researchers have proposed techniques to automatically derive data decompositions based on simple machine models [88, 89, 109, 141, 175, 193]. However, these techniques are insufficient because the efficiency of a given data decomposition is highly dependent on both the actual node program generated by the compiler and its performance on the parallel machine. "Optimal" data decompositions may prove inferior because the compiler generates node programs with suboptimal communications or poor load balance. Similarly, marginal data decompositions may perform well because the compiler is able to utilize collective communication primitives to exploit special hardware on the parallel machine.

What the programming system must do is help the user to understand the effect of a given data decomposition and program structure on the efficiency of the *compiler-generated* code running on a given target machine. The Fortran D data partitioner, shown in Figure 10.1, supports this through the use of static performance estimation. Data decompositions are selected using standard heuristics, then evaluated using a combination of the Fortran D compiler and static performance estimation [17, 103, 113]. The resulting data decompositions should be efficient for both the compiler and parallel machine.

Note that even though it is desirable, to assist compilation the static performance estimator does not need to predict *absolute* performance. Instead, it only needs to accurately predict the performance of a program version *relative* to other versions. The prototype performance estimator has proved quite precise, especially in predicting the relative performances of different data decompositions [17]. We believe that for regular loosely synchronous problems written in a data-parallel programming style, the automatic data partitioner can determine an efficient partitioning scheme without user interaction. The automatic data partitioner can also be used to interactively suggest possible data decompositions and estimate their efficiency. This permits the user to apply knowledge not available at compile-time.

10.2.3 Additional Compilers

To yield the best performance, the Fortran D compiler must be used in conjunction with other compilation tools. Figure 10.2 displays the compilers in the in the Fortran D compilation system and how they interact with the Fortran D compiler. The data partitioner and the Fortran D compiler partitions a shared-memory program onto the nodes of a distributed-memory machine. This represents the coarsest level of parallelism. Another level of parallelism can be exploited if each node of the machine is a shared-memory multiprocessor and/or possesses vector units. In this case a parallelizing compiler can be used to parallelize/vectorize the node program. By treating each *send* as a use and each *recv* as a definition, the shared-memory parallelizer can perform program transformations while preserving the legality of the original message-passing program.

To achieve good scalar performance on a single processor, the compilation system must also pay close attention to improving data locality in the resulting node program. Selecting the right loop permutation is

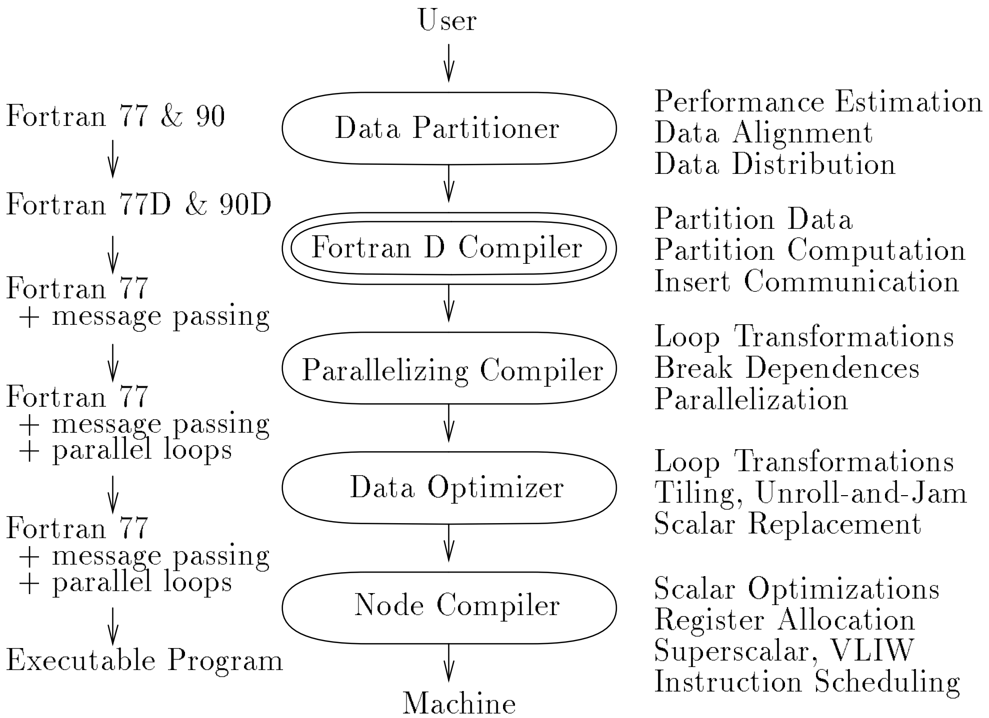


Figure 10.2 Fortran D Compilation System

a fundamental goal and should also be considered during the Fortran D and shared-memory parallelization process. The remaining optimizations, tiling for cache reuse and unroll-and-jam & scalar replacement for register usage, may be applied after previous parallel compilers have completed. Finally, an optimizing scalar compiler can perform classical scalar optimizations, register allocation, and exploit instruction-level parallelism for superscalar and VLIW processors through advanced instruction scheduling.

10.3 Fortran D Programming Environment

The Fortran D compilation system is designed to automatically translate input programs into highly efficient code. The components of the Fortran D programming environment assist the user when obstacles are encountered. Because the Fortran D compilation system may drastically change the structure of the original program, it is important that the programming environment helps the user understand the effect of changes to the original program.

The architecture of the Fortran D environment is shown in Figure 10.3. Information from executing training sets on the machine is provided to the compiler and environment. The Fortran D compiler calculates information on load imbalance and communication. The environment collects at run-time information on program events and data values. The programming environment uses this information to provide performance profiling, visualization, and debugging for the user.

10.3.1 Interactive Programming

Since the Fortran D programming system is built on top of ParaScope, it also provides program analysis, transformation, and editing capabilities that allow users to restructure their programs according to a data-parallel programming style.

The automatic data partitioner may be used as a starting point for choosing a good data decomposition. When invoked interactively for specific program segments, it responds with a list of the best decomposition schemes, together with their static performance estimates. If the user is not satisfied with the predicted overall performance, he or she can use the performance estimator to locate communication and computation intensive program segments. The Fortran D environment can then advise the user about the effects of program changes on the choice of a good data decomposition.

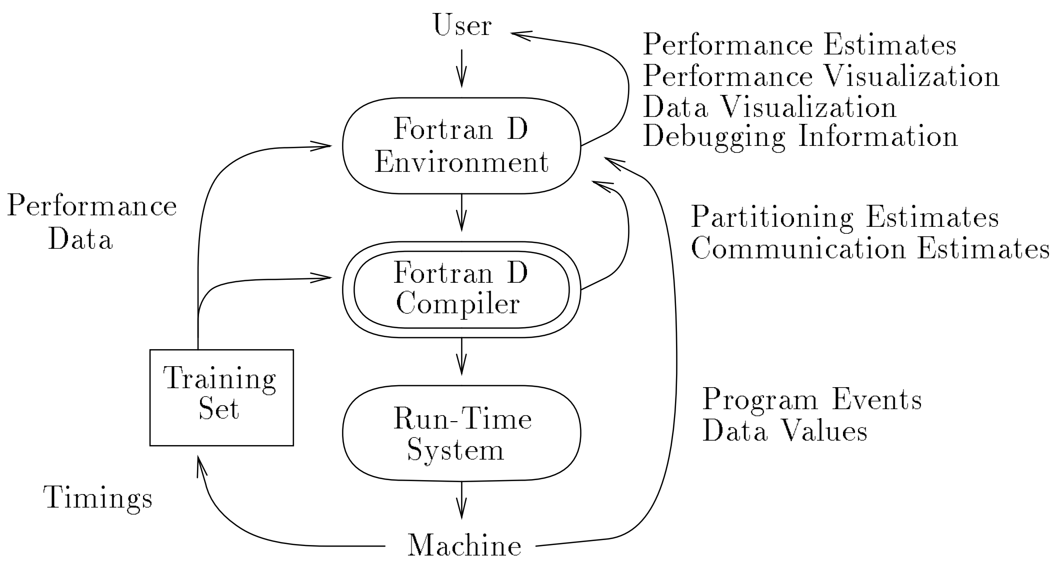


Figure 10.3 Feedback in the Fortran D Programming Environment

10.3.2 Run-Time System

The prototype Fortran D compiler generates calls to Intel NX/2 message-passing libraries, and will soon be extended to also target the Thinking Machines CMMD library. Future targets may include portable, efficient communication libraries such as Express [167] or PVM [195]. Because these communication libraries have similar message-passing interfaces, adding new targets is a straightforward task. Ongoing standardization efforts for message-passing libraries will ease the burden of the compiler writer.

10.4 Discussion

The Fortran D compiler automates the time consuming task of deriving node programs based on the data decomposition. Two other components of the Fortran D programming system, the static performance estimator and automatic data partitioner, support another important step in developing a data-parallel program—selecting a data decomposition. The Fortran D compiler cooperates with other compilation tools, such as a shared-memory parallelizing and vectorizing compiler, data locality optimizers, and traditional scalar compilers. To be useful, it must also takes part in providing feedback to the programmer other elements of the programming environment.

Chapter 11

Related Work

Supporting parallel programs on architectures with physically distributed memories is a challenging problem with potential for high payoffs. Here we briefly examine alternatives explored by other researchers, including parallel architectures & operating systems, programming models & languages, and shared-memory parallelizing compilers. We also describe other distributed-memory compilation systems in relation to Fortran D [57, 73, 93, 104, 105, 106, 107].

11.1 Parallel Architectures and Operating Systems

Because of high interprocessor communication costs, many researchers believe efficient general-purpose parallel computing on distributed-memory machines requires some form of architectural and operating systems support. Systems such as Alewife [42], April [2], DASH [137], KSR-1, and Willow [24] attempt to provide a coherent global address space through innovations in hardware and operating systems. Early experiences with these systems have been positive, but show that locality of reference is a major factor in determining performance.

Researchers are also developing techniques to provide a global address space on distributed-memory machines entirely through software. Distributed shared-memory systems such as Amber [48], Ivy [142], Munin [23], and Platinum [64] utilize the operating system to detect and expedite interprocessor data motion. These software-based systems incur significantly greater overhead, but have the advantage of not requiring additional specialized hardware. Because of their wide availability, researchers in this area have been particularly interested in targeting networks of high-performance workstations.

Both hardware and software approaches ease the task of parallel programming by eliminating separate address spaces and explicit communications. However, many significant problems remain. First, scientists are still required to write explicitly parallel programs that deal with synchronization and non-determinism. This requirement is undesirable because computational scientists are interested in solving numerical applications, not in learning the semantics of synchronization constructs or memory consistency models.

To reduce apparent memory latency, high-performance machines frequently provide data caches. Maintaining coherent caches for distributed memory architectures is an extremely difficult problem. For efficiency coherence is maintained at a reasonably large granularity, usually at the size of individual pages or long cache lines. *False sharing* may then cause extra data movement when logically separate data are mapped to the same memory unit. Existing distributed shared-memory systems also do not provide specialized data movement patterns that frequently occur in scientific codes; *e.g.*, array transposes, shift or broadcast along one or more dimension, various gather/scatter operations, or synchronized all-to-all transfers. In comparison, these patterns are easily recognized by the user or compiler based on program analysis.

Hardware and operating systems are fundamentally limited in that they can only react to accesses to nonlocal memory. They cannot hide memory latency by prefetching data before it is needed, or reduce data movement costs by fetching entire logical blocks at once. At best they can relax memory consistency, maintain a history of past accesses, and try to guess future reference patterns [112, 213]. Unlike compilers, operating systems are incapable of arranging data layout to avoid contention, reordering computation to improve data movement, or replicating computation to eliminate communication. Since data locality determines performance on distributed-memory architectures, even the best designed architecture and operating system will benefit from compiler optimizations to improve locality of reference.

11.2 Programming Models and Languages

The proliferation of parallel architectures has focused much attention on machine-independent parallel programming. Selecting an efficient level of parallel programming is an open research issue. Some researchers have proposed elegant architecture-independent programming models such as *Bird-Meertens formalism* [192] and the *Bulk-Synchronous* bridging model [201]. However, these parallel programming models are intended to guide the development of new data-parallel algorithms; they cannot be used to help scientists write parallel programs since no language or compiler support is provided.

High-level languages such as Delirium [147], Linda [41], PCN [71] and Strand [70] are valuable when used to *coordinate* coarse-grained functional parallelism, as are graphical programming languages such as CODE [159], HeNCE [21], and Schedule [68]. Both require the user to develop explicit parallel programs. In comparison, declarative languages such as Jade can exploit coarse-grain parallelism at run-time, using side effect information collected from user annotations [135]. However, all these parallel languages tend to be inefficient or burdensome for exploiting data-parallelism. As a result, many researchers have turned to advanced parallelizing compilers or lower-level parallel languages.

Though most parallel languages concentrate on specifying synchronization for task-level parallelism, we found several languages constructs useful for data-parallel programming. We have adopted many such features into Fortran D. In particular, we have been influenced by alignment specifications from CM FORTRAN [196], distribution specifications from KALI [127, 153], and structures to handle irregular distributions from PARTI [188]. We also incorporated the FORALL statement from Myrias [22] and CM Fortran [6]. The REDUCE statement in Fortran D is patterned after equivalent reduction functions in Fortran 90 [13].

11.3 Shared-Memory Compilers

Data-parallelism can usually be utilized as *loop-level* functional parallelism; it comprises most of the usable parallelism in scientific codes when synchronization costs are considered [52]. Shared-memory parallelizing compilers such as Parafrase [131, 171], PFC [9, 10], PTRAN [7], ParaScope [35, 59], and SUIF [197] use *data dependence* [130, 132] to detect and exploit parallel loops on MIMD shared-memory machines. Precise program analysis is needed to handle symbolics [91], procedure calls [62, 98, 143], and array reference aliases [19, 83, 133, 208]. Program transformations based on dependences may also be used to expose additional parallelism [117, 118, 119, 164].

Shared-memory parallelizing compilers can aid the programming process on distributed shared-memory machines, but possess several shortcomings. First, the main goal of parallelization techniques for shared-memory machines is to exploit parallelism. However, on distributed-memory machines data movement costs are much higher and must be taken into account when applying optimizations.

Compilation techniques have been developed to improve data locality on scalar and shared-memory machines. Program transformations can enhance *temporal* and *spatial* locality of scientific codes, improving the usage of higher levels of the memory hierarchy such as registers and cache [34, 40, 116, 173, 204]. Heuristics have been developed for managing multiprocessors cache coherence in software through the use of block cache invalidate, prefetch, and update instructions [65, 85]. Taken to the limit, these optimizations begin to resemble message vectorization and code generation for distributed-memory machines.

Researchers have proposed merging these data locality optimizations with parallelism information for distributed-memory machines. The basic premise is that since parallel loops exhibiting data locality are guaranteed to compute fairly disjoint data sets, the partition of parallel loop iterations among processors can also be used to assign data to each processor [66, 168, 190]. Where data accesses are not entirely disjoint, grouping methods can be applied to reduce communication between loop iterations [120, 191]. However, in order to take advantage of data locality, the compiler must take into account *affinity*, the interaction between data placement and task scheduling [149].

Though shared-memory compilation approaches are useful for reducing memory contention, in the end they prove insufficient for distributed-memory machines because the resulting data partition may be highly complex and frequently change between loop nests. In addition, no support is provided for generating or optimizing communications where needed. Distributed-memory compilation techniques are still required. In fact, even shared-memory machines may benefit. Experiments have shown that *non-uniform memory access* (NUMA) shared-memory machines can actually achieve improved performance when programmed using a distributed-memory programming model [144], since resulting programs have been restructured to reduce interprocessor contention and expensive global memory references.

11.4 Distributed-Memory Compilers

Compared with other approaches, distributed-memory compilers can achieve improved performance through compile-time analysis and optimization of both parallelism and interprocessor data movement. In this thesis we classified them as follows:

- **Transformation-driven**. First-generation systems that perform optimizing transformations on naive run-time resolution code.
- **Language-driven**. Systems that rely on language features to guide the compilation and optimization process, usually requiring extensive run-time support.
- **Analysis-driven**. Compilers that rely on compile-time analysis. Some also require significant run-time support.

The classifications are not fixed; a number of systems are evolving from simple language translators to tools that perform impressive compile-time analysis and optimization. In the following sections, we describe compilers in each of these groups.

11.4.1 Transformation-Driven Compilers

The earliest distributed-memory compilers, Callahan-Kennedy, SUPERB, and ID NOUVEAU, perform extensive analysis but adopt a rather cumbersome compilation approach. These systems first insert guards and element-wise messages to generate a naive program performing run-time resolution, then apply program transformations and partial evaluation in order to produce more efficient code. To perform optimizing transformations, these compilers must keep track of intermediate program versions. However, the presence of guards and explicit communication affects program semantics in ways that are difficult to determine, especially whether program restructuring would cause deadlock [82].

In comparison, the Fortran D compiler performs its analysis up front and uses the results to drive code generation. This approach is simpler and provides greater flexibility. For instance, the Fortran D compiler performs transformations on the original program without needing to consider potentially introducing deadlock due to message reordering.

Callahan & Kennedy

Callahan & Kennedy proposed distributed-memory compilation techniques based on data-dependence driven program transformations [38]. These techniques were implemented in a prototype compiler in the ParaScope programming environment. In the compiler, standard and user-defined distribution functions are used to specify the data decomposition for sequential Fortran programs. The compiler inserts *load* and *store* statements to handle data movement, then applies program transformations such as loop interchange, distribution, and strip-mining to optimize guards and messages. The Callahan-Kennedy compiler prototype was eventually abandoned in favor of the greater flexibility of the Fortran D compiler.

SUPERB

SUPERB is a semi-automatic parallelization tool designed for MIMD distributed-memory machines [79, 80, 212]. It supports arbitrary user-specified contiguous rectangular distributions, and performs dependence analysis to guide interactive program transformations in a manner similar to the ParaScope Editor [129].

SUPERB originated overlaps as a means to both specify and store nonlocal data accesses. Once program analysis and transformation is complete, communication is automatically generated and vectorized utilizing data dependence information. Computation is partitioned via explicit guards, which may be eliminated by loop bounds reduction [81]. Unlike the Fortran D compiler, the original version of SUPERB did not support data alignment, cyclic distributions, automatic compilation, collective communications, dynamic data decomposition, and storage of nonlocal values in temporary buffers.

SUPERB performs interprocedural data-flow analysis of parameter passing. Each formal parameter is classified as unpartitioned or having a standard or nonstandard partition. A clone is produced for each possible combination of classification of the procedure parameters. The algorithm is similar to that employed by the Fortran D compiler, but less involved since SUPERB does not provide dynamic data decompositions. For local compilation, SUPERB modifies procedures so that arrays are always accessed according to their true number of dimensions, inserting additional parameters where necessary for newly created subscripts.

Id Nouveau

ID NOUVEAU is a functional language extended with single assignment arrays called *I-structures* [170, 178]. User-specified BLOCK distributions are provided. Initially, *send* and *recv* statements are inserted to communicate each nonlocal array access. Message vectorization is applied to combine messages for previously written array elements. Loop jamming (fusion) and strip-mining are used when writing array elements. Global accumulate & reduction operations are supported. Analysis is considerably simplified due to I-structures. Unlike other systems, program partitioning produces distinct programs for each node processor.

11.4.2 Language-Driven Compilers

A second group of language-driven distributed-memory compilers target programs containing explicit parallelism and possibly user-specified communication. Compilation is usually limited to extracting communication out of parallel computations, then partitioning the computation across processors. Communication selection is based on matching language features or user annotations to routines in the run-time system.

Some languages specify parallelism through a *local view* of computation, where the programmer specifies computation for an individual data point, relying on the compiler and run-time system to replicate the computation for all data points [96, 183]. In comparison, Fortran D supports a *global view* of computation, where the program specifies the overall computation, counting on the compiler to partition the computation.

In comparison to language-driven compilers, systems like Fortran D reduce the burden on the user through program analysis, the key to advanced optimization. By applying dependence analysis, the Fortran D compiler can exploit parallelism without relying on language features or annotations. It can also vectorize messages in sequential regions, such as those found in SOR or ADI integration. Many language-driven compilers are improving their compile-time analysis and optimization, especially CM FORTRAN.

Crystal

CRYSTAL is a high-level functional language [53, 138, 139, 140, 141]. Because it targets a functional language, the CRYSTAL compiler possesses markedly different program analysis techniques than compilers for imperative languages such as Fortran. However, it performs significant compile-time analysis and optimization, pioneering both automatic data decomposition and collective communications generation techniques. It is unclear whether the CRYSTAL language can express a wide range of scientific computations. Work in progress to adapt the CRYSTAL compiler for scientific Fortran codes will help answer this question.

During compilation, the CRYSTAL compiler first separates programs into *phases*, where each phase has a different computation structure, represented by an *index domain*. Heuristics are employed to align data arrays with the index domain, both within and across dimensions, then derive the control structure of the program. Communication patterns are synthesized syntactically from the computation, evaluated for a variety of block distributions, then matched with CRYSTAL collective communication routines. Later phases of the compiler generate message-passing C programs for the physical machine.

BLAZE, Kali

BLAZE is one of the first distributed-memory compilers [126, 152]. KALI, its successor, is the first compiler system that supports both regular and irregular computations on MIMD distributed-memory machines [121, 122, 123, 127, 124, 153]. Programs written for KALI must specify a virtual processor array and assign distributed arrays to BLOCK, CYCLIC, or user-specified distributions. Instead of deriving a functional decomposition from the data decomposition, KALI requires that the programmer explicitly partition loop iterations onto the processor array. This is accomplished by specifying an *on clause* for each parallel loop. Communication is then generated automatically based on the on clause and data distributions. Arguments to procedures are labeled with their expected incoming data partition. The user must ensure that the procedure is called only with the appropriately decomposed arguments. An *inspector/executor* strategy is used for run-time preprocessing of communication for irregularly distributed arrays [125, 155]. Major differences between KALI and the Fortran D compiler include mandatory on clauses for parallel loops, support for alignment, collective communications, and dynamic decomposition.

CM Fortran

CM FORTRAN is a version of Fortran 77 extended with vector notation, alignment, and data layout specifications [5, 196]. Programmers must explicitly specify data-parallelism through the use of array operations and by marking array dimensions as parallel. The CM FORTRAN compiler utilizes user-defined *interface blocks* to specify a data partition for each procedure. Array parameter are then copied to buffers of the expected form at run-time, eliminating the need for interprocedural analysis.

The first CM FORTRAN compilers treat the underlying machine *field-wise* as a collection of 1-bit processors, resulting in inefficient code. Later compilers improve performance by operating *slice-wise*, using 32 bit slices to take advantage of the 32-bit Weitek floating-point processors on the CM-2 [187]. The stencil compiler avoids unnecessary intra-processor data motion, inserting communication only for data located on a separate physical, rather than virtual, processor [31]. It resembles a highly specialized Fortran D compiler for stencil computations, including optimizations at the assembly code level to improve register usage. The CM FORTRAN compiler has been retargeted for the CM-5, but results show that it fails to fully exploit the CM-5 architecture [186].

C*, Dataparallel C

C* and DATAPARALLEL C are extensions of C similar to C++ that supports SIMD data-parallel programs [96, 97, 179]. They both provide a *local view* of computation. Data is labeled as *mono* (local) or *poly* (distributed). There are no alignment or distribution specifications; the compiler automatically chooses the data decomposition. Parallel algorithms are specified as *actions* on a *domain*, an abstract data type implementation based on the C++ class. Communications are automatically generated by the compiler. Despite their SIMD semantics, C* and DATAPARALLEL C can be efficiently compiled to both SIMD and MIMD architectures. Instead of generating virtual processors for each element of a domain, compile-time analysis enables *contraction*, emulating virtual processors via loops. Researchers have also examined synchronization problems encountered when translating SIMD programs into equivalent SPMD programs, as well as several communication optimizations [96, 174]. Experience will show whether SIMD languages such as C* provide sufficient flexibility for a wide class of scientific computations.

DINO

DINO is an extended version of C supporting general-purpose distributed computation [181, 182]. DINO supports BLOCK, CYCLIC, and special stencil-based data distributions with overlaps, but provides no alignment specifications. Like C*, it provides the programmer with a local view of the computation. A DINO program contains a virtual parallel machine declared to be an *environment*. They generate multiple processes per physical processor when large numbers of virtual processors are declared in the environment. Nonlocal memory references must be annotated with the "#" operator. The DINO compiler then translates these references into communications. Parallelism is specified by *composite functions*. Passing distributed data as parameters generates nonlocal memory accesses. The user labels parameters as IN or OUT to indicate whether their values are used and/or defined. Special language constructs are provided for reductions. DINO programs are deterministic unless special asynchronous distributed arrays are used.

DINO is a powerful and flexible language. Programmers can use it to specify optimizations such as coarse-grain pipelining and iteration reordering for pipelined and parallel computations [161]. However, its many features may prove burdensome to users. DINO2 proposes a large set of additional language features to support parallel task creation, data and computation mapping, synchronization, and communication [180]. DYNO provides support for irregular and adaptive numeric programs [203].

Paragon

PARAGON is a C-based programming environment targeted at supporting SIMD programs on MIMD distributed-memory machines [46, 177]. It provides both language extensions to C and run-time support for task management and load balancing. Data distribution in PARAGON may either be performed by the user or the system. Parallel arrays are mapped onto *shapes* that consist of arbitrary rectangular distributions. *Communication structures* can directly specify arbitrary mappings on parallel data. The location of each array element may be determined at run-time by checking the *distribution map* stored on each processor. Redistribution and replication of arrays and subarrays, as well as permutation and reduction mechanism are supported. PARAGON has been extended to handle the special class of *irregularly-coupled regular-meshes* [47]. It does not currently perform analysis or transformations to detect or enhance parallelism.

ARF, PARTI

ARF is a compiler for irregular computations [209]. It provides BLOCK and CYCLIC distributions, and is the first compiler to support user-defined irregular distributions. The goal of ARF is to demonstrate that inspector/executors can be automatically generated by the compiler for user-specified parallel loops. It does not currently generate messages at compile-time for regular computations. ARF is designed to interface Fortran application programs with PARTI, a set of run-time library routines that support irregular computations on MIMD distributed-memory machines [188]. PARTI is first to propose and implement user-defined irregular distributions [155] and a hashed cache for nonlocal values [108, 156]. It is employed by a number of systems to support irregular computations, including the Fortran D compiler.

Pandore

PANDORE is a compiler for distributed-memory machines that takes as input C programs extended with BLOCK, CYCLIC, and overlapping data distributions [12, 203]. Distributed arrays are mapped by the compiler onto a user-declared *virtual distributed machine* that may be configured as a vector, ring, grid, or torus. The compiler then outputs code in the *vdm_l* intermediate language. Calls to the PANDORE communication library to access nonlocal data is also automatically generated by the compiler. Guard introduction and communications optimization techniques are under development.

Oxygen

OXYGEN is a compiler for the K2 distributed-memory machine [184, 185]. Unlike most systems, OXYGEN follows a functional rather than data decomposition strategy. Task-level parallelism is specified by labeling each parallel block of code with a *p_block* directive. Loop-level parallelism is specified by labeling parallel loops with either *split* or *scatter* directives. Decompositions are mapped onto the K2 architecture as *ring*, *rowwise*, or *colwise*. Distributed data arrays may be declared as *local*, *multicopy*, or *singlecopy*, corresponding to private, replicated, and distributed, respectively. Explicit communications directives for reductions and broadcast are also provided. The OXYGEN compiler converts Fortran code with user directives into C++ node programs with communications. Messages are inserted at points in the program called *checkpoints* to enforce coarse-grain synchronization. Work is in progress to automatically generate OXYGEN directives for functional and data decomposition.

SPOT

SPOT is a point-based SIMD data-parallel programming language [193, 194]. Distributed arrays are defined as *regions*. Computations are specified from the point of view of a single element in the region, called a *point*. Locations relative to a given point are assigned symbolic names by *neighbor* declarations. An *iteration index* operator allows the programmer to specify whether nonlocal values from neighbors are from the current or previous iteration. This stencil-based approach allows the SPOT compiler to derive efficient *near-rectangular* data distributions. The compiler then generates computation and communication by expanding the single point algorithm to cover all points distributed onto a node. No alignment and or distribution specifications are provided. It is not clear how SPOT will support computation patterns that cannot be described by stencils.

Vcode

VCODE is data-parallel intermediate language [49, 51]. It is designed to allow easy porting of data-parallel languages between parallel architectures. Initial implementations target the Thinking Machines CM-2, Encore Multimax, and Cray Y-MP. In Vcode, computation is performed as *segmented scans* on vectors [25]. Vcode demonstrates the usefulness of segmented scans, but possesses severe shortcomings. A major problem is that most information available in the original program is lost during the translation to Vcode, and must must be recaptured through difficult analysis to enable efficient compilation [50].

Additional Compilers and Systems

We briefly review the large number of distributed-memory compilers and systems that have been developed. The first group of compilers target Fortran 90. ADAPT [154] and ADAPTOR [27] both perform translations relying on run-time support from a portable library. Wu & Fox describe development of a Fortran 90D compiler developed and validated via a test-suite approach [210]. These compilers perform little analysis or optimization, extracting parallelism from Fortran 90 array syntax or parallel loop annotations.

The remaining systems are quite varied. BOOSTER provides a system to annotate data placement through user-defined shapes and views [163]. ORCA-1 supports explicit parallelism through phase abstractions [145]. MODULA-2* provides a superset of data-parallel constructs [169]. METAMP provides a high-level interface for explicit message-passing [162]. PYRROS statically partitions task graphs for multiprocessors [211]. Gupta & Banerjee develop a methodology for generating complex collective communication [90]. Wolfe introduces *loop rotation*, a combination of loop skew, reversal, and interchange to reduce contention for common data [207]. Ramanujam & Sadayappan tile loop nests for distributed-memory machines [176]. O'Boyle & Hedayat develop a linear algebraic framework for parallelizing SISAL [160].

11.4.3 Analysis-Driven Compilers

The final group of distributed-memory compilers are analysis-driven; they rely more on compile-time analysis than language features or user annotations. These compilers typically accept Fortran 77 or 90 programs with data decomposition annotations. They perform analysis to automatically detect parallel operations. Compared with the Fortran D compiler, these compilers shift much of the burden of parallelization to the run-time system. Calls to library routines are inserted at compile-time and invoked at run-time to calculate information such as local loop bounds, array indices, and the amount of data to communicate. This approach reduces the burden on the compiler and provides it with greater flexibility, allowing it to handle complex cases such as work arrays or array reshaping at procedure boundaries at run-time. However, it does limit the amount of optimization that may be applied at compile-time for simpler computations.

Forge90, MIMDizer

FORGE90, formerly MIMDIZER, is an interactive parallelization system for MIMD shared and distributed-memory machines from Applied Parallel Research [14, 100]. It performs data-flow and dependence analyses, and also supports loop-level transformations. Associated tools graphically display call graph, control flow, dependence, and profiling information. When programming for distributed-memory machines, users may interactively select BLOCK or CYCLIC distributions for selected array dimensions. *Code spreading* is applied interactively or automatically to loops to introduce parallelism. Distributed arrays are linearized in order to simplify ownership computation at run-time. FORGE90 automatically generates communications corresponding to nonlocal memory accesses at the end of the parallelization session.

ASPAR

ASPAR is a compiler that performs automatic data decomposition and communications generation for loops containing a single distributed array [110]. It utilizes collective communication primitives from the EXPRESS run-time system for distributed-memory machines [167]. ASPAR automatically selects BLOCK distributions; no alignment or distribution specifications are provided. ASPAR performs simple dependence analysis using *A-lists* to detect parallelizable loops. The structure of the loop computation may be recognized as a *reduction* operation, in which case the loop is parallelized by replacing the reduction with the appropriate EXPRESS *combine* operation. If the loop performs regular computations on a distributed array, a *micro-stencil* is derived and used to generate a *macro-stencil* to identify communication requirements. Communications utilizing EXPRESS primitives are then automatically generated. Like FORGE90, ASPAR performs less compile-time analysis and optimization, relying instead heavily on run-time support.

Vienna Fortran

SUPERB has recently been adapted for a new language called VIENNA FORTRAN [45]. Vienna Fortran does not provide a decomposition, but possesses alignment and distribution specifications similar to Fortran D. It supports explicit processor array declarations, irregular computations, performance estimation, and automatic data decomposition [29, 43, 69]. Dynamic data decomposition is permitted.

VIENNA FORTRAN allows the user to specify additional attributes for each distributed array [44]. *Restore* forces an array to be restored to its decomposition at procedure entry. *Notransfer* causes remapping to be performed logically, rather than actually copying the values in the array. *Nocopy* guarantees that its formal and actual parameters have the same data decomposition. No copies take place, but an error results if different decompositions are encountered. We attempt to achieve the same benefits in the Fortran D compiler through interprocedural analysis and optimization.

AL, iWarp

AL is a language designed for the WARP distributed-memory systolic processor [198, 199]. The programmer utilizes DARRAY declarations to mark parallel arrays. The AL compiler then applies *data relations* to automatically align and distribute each DARRAY, detect parallelism, and generate communication. Only one dimension of each DARRAY may be distributed, and computations must be *linearly related.* Its successor, the IWARP compiler, targets the difficult task of simultaneously supporting data-parallelism, coarse-grain task parallelism, and fine-grain systolic parallelism [86, 102, 134].

P^{3}C, VMMP

P^{3}C, the Portable Parallelizing Pascal Compiler, translates sequential Pascal programs into explicit parallel code [76]. The output program relys on VMMP [75], a portable software environment running on many multiprocessors. P^{3}C performs simple analysis to detect and parallelize parallel loops and reductions. Static execution time estimation is used to choose between sequential and parallel code. P^{3}C emphasizes portability rather than complex program analysis.

Fortran-90-Y

The FORTRAN-90-Y compiler is designed to apply formal specification techniques to generate efficient code for the CM-2 and CM-5 [54]. It is intended to support rapid prototyping of compilation and optimization techniques. The FORTRAN-90-Y compile uses YALE INTERMEDIATE REPRESENTATION as the basis for performing machine-independent program transformations. It achieves performance comparable to that of the production CM FORTRAN compiler.

The FORTRAN-90-Y compiler also supports research on optimization techniques for distributed-memory machines. A transformation strategy is presented to improve parallelism for *iterative spatial* Fortran 90 loops on the CM-2, using a combination of loop interchange, skew, and strip-mining [55]. These transformations are similar to Fortran D program transformations for optimizing pipelined computations on MIMD distributed-memory machines. Techniques are also developed for algebraic representation of data motion and data layout for YALE EXTENSIONS, a set of data layout declarations [56]. Variable references are translated into *communication expressions* and optimized using *communication algebra* through idioms.

Chapter 12

Conclusions

The Fortran D compiler demonstrates that with minimal language and run-time support, advanced compilation technology can produce efficient programs for MIMD distributed-memory machines. In this chapter we summarize the research embodied in the thesis—new compilation techniques, experimental results, and validation of a prototype compiler. We discuss the development of High Performance Fortran and our perspectives on the issues confronting compilers for distributed-memory machines, including language, compilation model, analysis, optimization, and long-term prospects. We conclude by considering areas for future work.

12.1 Compiling Fortran D

In this thesis, we identified data decomposition specifications as the key to automatically compiling programs for distributed-memory machines. Based on this premise, we designed a language, developed compilation technology, and applied both towards the construction a prototype Fortran D compiler. Preliminary experiences show that Fortran D programs written in a *data-parallel* programming style can be compiled into efficient parallel programs for MIMD distributed-memory machines, at least for parallel stencil codes.

The two major tasks facing the Fortran D compiler is to partition the program and introduce communication for nonlocal data accesses. To achieve high performance, the compiler must strive to minimize both load imbalance and communications costs. The Fortran D compiler utilizes a code generation strategy based on message vectorization that unifies and extends previous techniques. Its first applies scalar data-flow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependences. The compiler partitions data by analyzing Fortran D data decomposition specifications to calculate distribution functions. Computation is then partitioned across processors using the "owner computes" rule.

Once the work partition is computed, the compiler calculates the nonlocal data accessed by each processor. The compiler examines each nonlocal reference, using results of data decomposition, symbolic and dependence analysis to determine the legality of optimizations to improve parallelism and reduce communication costs. The compiler collects the extent and type of nonlocal data accesses to calculate the storage required for nonlocal data. Finally, the Fortran D compiler uses the results of analysis and optimization to generate the SPMD program with explicit message-passing that executes directly on the nodes of the distributed-memory machine. Array and loop bounds are reduced to instantiate the data and computation partition. Nonlocal data accesses result in the generation of calls to buffer, *send*, *recv* calls, and collective communication routines. The compiler applies run-time resolution to references not analyzed at compile time.

The Fortran D compilation process presents many opportunities for optimization. Program transformations modify the program execution order to enable optimizations. Communication optimizations reduce communication overhead by decreasing the number of messages or hide communication overhead by overlapping the cost of remaining messages with local computation. Parallelism optimizations restructure the computation or communication to increase the amount of useful computation performed in parallel. Cost models help guide both parallelism and communication optimizations.

Interprocedural analysis, optimization, and code generation algorithms limit compilation to only one pass over each procedure by collecting summary information after edits, then compiling procedures in reverse topological order to propagate necessary information. Delaying instantiation of the work partition, communication, and dynamic data decomposition enables interprocedural optimization. Both Fortran 77D and 90D may be compiled in a unified system by ordering loop fusion, partitioning, and sectioning. The Fortran D compiler must cooperatively interact with many other compilers and tools in a complete programming system. Empirical studies validate the performance of the prototype compiler and point out its strengths and weaknesses.

12.2 Contributions

This thesis contributes in three areas: improved compilation techniques, experimental evaluation of individual design choices, and empirical validation of the prototype Fortran D compiler.

12.2.1 Compilation Techniques

This thesis develops a number of compilation techniques that may be classified as new discoveries or extensions to existing methods. Novel algorithms introduced in this thesis include:

- Providing the DECOMPOSITION statement, allowing users to map different arrays to the same logical group.
- Applying vector message pipelining to hide communication costs without increase communication overhead.
- Identifying and optimizing parallelism in a class of pipelined computations.
- Cost models for guiding communication and parallelism optimizations.
- Efficient one-pass interprocedural compilation and optimization.

In addition, the Fortran D compiler incorporates a large number of adaptations and enhancements to compilation techniques discussed by previous researchers. These extensions include:

- Combining ALIGN, DISTRIBUTE, and extending FORALL for both SIMD and MIMD systems.
- Formulating a compilation model based on the owner computes rule. Applying distribution functions at compile-time and efficiently instantiating the program partition at run-time.
- Extending message vectorization into a complete code-generation strategy, including message coalescing and aggregation to reduce communication overhead.
- Adapting methods for relaxing the owner computes rule.
- Generalizing iteration reordering for unbuffered messages.
- Developing heuristics to guide program transformations such as loop interchange, fusion, distribution, and strip-mining.
- Unifying compilation of Fortran 77D and 90D.

12.2.2 Experimental Evaluation and Validation

In addition to new compilation techniques, this thesis contains a number of experimental studies that have provided valuable information. These studies fall into two groups. The first group of experiments verify the design of individual components of the compiler by:

- Evaluating the importance and interaction between different optimizations. Results indicate that exploiting parallelism is key, followed by optimizations that eliminate large numbers of messages, *e.g.,* message vectorization, coarse-grain pipelining, and collective communication.
- Establishing the importance of interprocedural optimization. Measurements show order of magnitude improvements in the optimized version of DGEFA, Gaussian elimination with partial pivoting.

- Verifying the importance of a unified compilation framework for Fortran 77D and 90D. Results show order of magnitude improvements for ADI integration due to loop fusion, and 30–50% improvements for kernels due to data prefetching.

The second group of experiments are designed to validate the complete prototype compiler. They compare the output of the prototype against:

- Hand-optimized message-passing kernels and programs on the Intel iPSC/860. On parallel stencil computations, the Fortran D compiler is slower by up to 50% for kernels but matches the performance of programs. On linear algebra and pipelined codes the prototype is slower by 50–100%.

- Kernels and programs compiled by the CM Fortran compiler for the TMC CM-5. The Fortran D compiler is faster by 2–17 times on parallel stencil codes, and over 20 times faster on linear algebra and pipelined kernels.

12.3 High Performance Fortran

Many researchers in the field have come to share the Fortran D view of data-parallel programming, as have most major vendors of parallel machines. Distribution of the initial Fortran D language specification quickly captured the interest of the high-performance computing community. It culminated a year later in the formation of the High Performance Fortran Forum, a coalition of vendors, research laboratories, academics, and users dedicated to defining an informal Fortran language standard for high-performance computing. Members include Convex, Cray Research, DEC, Fujitsu, IBM, Intel SSD, and Thinking Machines.

After much effort and a year of meetings, High Performance Fortran (HPF) was presented for public comment at Supercomputing '92 [99]. Many core features of Fortran D were adopted by HPF, along with numerous additional extensions and refinements. Several hardware vendors and software companies are in the process of developing commercial HPF compilers, some scheduled for distribution as early as 1993. We consider this wide-spread interest and acceptance additional validation of our thesis—that advanced compiler technology makes languages such as Fortran D and High Performance Fortran efficient and feasible for a variety of parallel architectures.

12.4 Perspectives

Our experiences in designing, implementing, and experimentally validating the prototype Fortran D compiler have provide us with a number of perspectives concerning languages such as Fortran D and High Performance Fortran.

12.4.1 Fortran D Language

First, we discovered that for most applications, very simple data decompositions suffice for good performance. In particular, only simple interdimensional array alignment proved useful for MIMD distributed-memory machines. Alignment offsets and options for overflow & range were not needed. Complex replication specifications were also unnecessary, variables were either distributed or replicated across all processors.

The data distribution attributes in Fortran D appear to be more than sufficient for generating good code. BLOCK distributions reduce communication for stencil computations, while CYCLIC distributions improve load balance for linear algebra computations. Higher dimensional data distributions are required to effectively exploit large number of processors for limited problem sizes. The FORALL loop, REDUCE statement, and ON clause were all unnecessary for regular dense-matrix computations. Dynamic data decomposition appears to be desirable only in a few rare cases.

12.4.2 Compilation Model

So far, our experiences with the Fortran D compilation model indicate the owner computes rule works quite well in practice. Given reasonable data decompositions, the rule partitions computation evenly while preserving data locality. Straightforward analysis can identify private variables and reductions, two occasions when the owner computes rule must be relaxed. Given the simple data decompositions and array subscripts encountered in programs thus far, distribution functions are easily applied and inverted as needed to enable compilation.

We also find that the SPMD model of computation used by the Fortran D compiler does not pose any restrictions to the flexibility or efficiency of Fortran D programs. Because the Fortran D compiler is designed for data-parallel codes, SPMD programs are sufficient to exploit all of the available parallelism. In fact, its control over SPMD output allows the Fortran D compiler compile-time control over scheduling and task-to-processor mapping not found in MIMD shared-memory fork-join programming models. Compilations systems that bypass SPMD to produce output specialized for particular processors avoid the need for run-time testing of processor identity. However, the computation avoided is purely local, and is so inexpensive it is unlikely to affect overall execution time.

12.4.3 Program Analysis

Just as in shared-memory parallelizing compilers, symbolic analysis is essential for providing sufficient information at compile-time. However, the penalty for missing information is much more severe for distributed-memory compilers. For well-written data-parallel codes, the Fortran D compiler does not seem to require symbolic analysis beyond the capability of mature parallelizing compilers. The difficulty lies in attempting to analyze "dusty deck" programs, which usually contain constructs too difficult for even state-of-the-art dependence analyzers. These codes are poor targets in any case for massively-parallel machines.

Regular section descriptors appear to be adequate for describing index and iteration sets for parallel stencil programs. More complex descriptors such as simple sections [18] may be required for blocked linear algebra computations, as they frequently access data in trapezoidal sections. Kill analysis for scalars and array sections can aid the Fortran D compiler but are not essential; they seem more useful for guiding automatic data decomposition. Interprocedural analysis designed to optimize dynamic data decomposition through the calculation of *live decompositions* may be difficult, but the scarcity of opportunities for dynamic data decomposition render the problem academic.

12.4.4 Compiler Optimization

Our experiments show that recognizing and exploiting parallelism is the key optimization for the Fortran D compiler. The impact of exploiting parallelism increases with both problem size and the number of processors. The overall effect of communication optimizations, on the other hand, is limited by the portion of execution time devoted to performing communication. Because of the relatively high cost of initiating each message, communication optimizations that can eliminate large numbers of messages appear to be most significant. Additional communication optimizations seem worthwhile only for applications where communication is a large percentage of total execution time.

12.4.5 Algorithmic Complexity

Though not considered in detail in this thesis, the complexity of most Fortran D compilation and optimization algorithms appear to be strictly polynomial in the number of variable references in each loop nest. Dependence edges must be examined, and pair-wise comparisons of variable references or RSDs occur, but combinatorial search is rare. Most algorithms simply apply affine functions to directly calculate a solution, such as applying distribution functions to an array subscript to discover the local index set. The only algorithm where exponential search may potentially occur is in vector message pipelining, which resembles instruction scheduling in the presence of constraints. The prototype compiler avoids combinatorial search by applying a greedy algorithm to select communication placement.

12.4.6 Prospects

Despite its successes, our experiences with the Fortran D compiler shows that it is not a panacea for all the problems with parallel programming. In particular, the Fortran D compiler cannot be effectively used to parallelize "dusty deck" Fortran programs, even those written for vector or parallel machines. Significant user effort is still required to write programs in a clean data-parallel style before the Fortran D compiler can successfully analyze and compile the result. However, once programs have been rewritten in such a manner, the Fortran D compiler fulfills its promise of providing an efficient data-parallel programming model.

High Performance Fortran finesses the problem of dusty decks by choosing Fortran 90 as its base language. HPF thus forces programmers to write in a data-parallel style with explicit parallelism. This approach greatly simplifies the task of the HPF compiler as compared to the Fortran D compiler, and makes the relatively quick release of HPF compilers achievable. However, the poor performance of the CM Fortran compiler on the CM-5 demonstrates that explicit parallelism and language features do not entirely substitute for advanced compiler technology.

12.5 Future Work

We conclude by pointing out some areas for continuing the research begun in this thesis.

12.5.1 Extensions to the Fortran D Compiler

First, to increase its usefulness as a programming tool, the Fortran D compiler must be improved. We found that the prototype compiler was unable to compile many scientific programs because of its inflexibility and immaturity. For some cases better symbolic or array data-flow analysis is required for the compiler to understand complex programming constructs. In other instances better run-time support would have enabled the prototype to handle boundary conditions that take up only a small percentage of total computation time. There are also complex computation patterns such as FFTs and particle-in-cell that the compiler fails to recognize and compile due to its immaturity. The compiler also proved to be less effective for computations requiring extensive communication, such as ADI integration and LU decomposition using Gaussian elimination with partial pivoting. More advanced optimizations need to be developed and implemented for these communication-intensive programs.

12.5.2 Shared-Address Space Architectures

We intend to adapt Fortran D compilation techniques for emerging shared-address space architectures (*e.g.,* DASH, KSR-1). On these scalable parallel systems processors share the same global address space, freeing the compiler from the chore of handling explicit address conversion. However, locality of reference, latency avoidance & tolerance, and block data movement are still key to achieving good performance. In addition, the compiler must insert synchronization to must prevent undesirable data-races. Techniques to improve locality on these systems have typically evolved from shared-memory compilers, consisting of program transformations, affinity scheduling, and software cache coherence. In comparison, we plan to evaluate compiler optimizations adapted from distributed-memory compilers.

The main advantage of compilers like Fortran D is that they can more precisely determine the location of data and computation. By retaining the SPMD programming model, these compilers can also retain control over both the computation partition and mapping of tasks relative to data. The Fortran D compiler may partition both data and computation at compile-time as before in order to maximize locality of reference. Run-time task scheduling is not needed for regular computations. However, the compiler can no longer rely on explicit message passing to provide both synchronization and data movement. Instead, it will need to insert explicit barrier synchronization using shared-memory compilation techniques. In addition, *send* and *receive* messages can be translated into hints for optimizing inter-processor data movement using block data prefetch and update.

12.5.3 Low-level Communication Primitives

Another improvement we plan for the Fortran D compiler is to exploit low-level communication primitives such as *active messages* [200]. Standard message-passing libraries incur significant software overhead due to their generality and fault tolerance. Lower-level communication primitives, on the other hand, can avoid much of the overhead of a standard typed message-passing communication layer by making optimistic assumptions or limiting generality. Experience has show this can yield integer factors of improvement in performance [180].

By utilizing such communication primitives, the Fortran D compiler can generate programs with better performance. However, greater effort is required on the part of the compiler to ensure these primitives are used correctly. For instance, a major source of improvement in low-level communication primitives is the elimination of unnecessary data buffering. Instead of first copying data to a system buffer, it is sent or received in place in the users' address space. To use these primitives, the Fortran D compiler must perform compile-time analysis and generate appropriate synchronization to ensure that data is not overwritten. Unlike unbuffered messages provided, active messages allows the Fortran D compiler to exploit compile-time knowledge to reduce system overhead.

12.5.4 Irregular Computations

This thesis has focused on the compilation of regular dense-matrix computations. Though these still comprise the majority of scientific applications, there is an accelerating trend in the computational science community towards irregular computations involving sparse matrixes and adaptive algorithms [72]. Fortunately, researchers have been investigating techniques for efficiently handling such computations through a combination of compile-time and run-time approaches [125, 155]. There is an ongoing effort to develop and improve support for such irregular computations in the framework of the Fortran D compiler and programming system [58, 95, 172].

12.5.5 Support for Parallel Input/Output

Finally, one aspect of automatic parallelization that most researchers prefer to avoid is that of providing support for parallel I/O. The current Fortran D compiler inserts guards to ensure all I/O is performed on a single processor. More advanced systems will need to deal with parallel I/O is a more comprehensive manner [28].

Bibliography

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[3] I. Ahmad, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Implementation and scalability of Fortran 90D intrinsic functions on distributed memory machines. Technical Report SCCS-256, NPAC, Syracuse University, March 1992.

[4] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[5] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[6] E. Albert, J. Lukas, and G. Steele, Jr. Data parallel computers and the FORALL statement. *Journal of Parallel and Distributed Computing*, 13(2):185–192, October 1991.

[7] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, June 1987.

[8] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, April 1983.

[9] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[10] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[11] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[12] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[13] ANSI X3J3/S8.115. Fortran 90, June 1990.

[14] Applied Parallel Research, Placerville, CA. *Forge 90 Distributed Memory Parallelizer: User's Guide*, version 8.0 edition, 1992.

[15] V. Balasundaram. Translating control parallelism to data parallelism. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[16] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[17] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[18] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[19] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[20] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1979.

[21] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[22] M. Beltrametti, K. Bobey, and J. Zorbas. The control mechanism for the Myrias parallel computer system. *Computer Architecture News*, 16(4):21–30, September 1988.

[23] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[24] J. Bennett, S. Dwarkadas, J. Greenwood, and E. Speight. Willow: A scalable shared memory multiprocessor. In *Proceedings of Supercomputing '92*, Minnesota, MN, November 1992.

[25] G. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, MA, 1990.

[26] S. Bokhari. Complete exchange on the iPSC-860. ICASE Report 91-4, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[27] T. Brandes. Efficient data parallel programming without explicit message passing for distributed memory multiprocessors. Internal Report AHR-92-4, High Performance Computing Center, GMD, September 1992.

[28] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent file operations in a High Performance Fortran. In *Proceedings of Supercomputing '92*, Minnesota, MN, November 1992.

[29] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[30] P. Briggs, K. Cooper, M. W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-147, Dept. of Computer Science, Rice University, December 1990.

[31] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[32] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, to appear 1993.

[33] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[34] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[35] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[36] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[37] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

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

[39] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[40] S. Carr, K. Kennedy, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. Technical Report TR92-195, Dept. of Computer Science, Rice University, November 1992.

[41] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[42] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[43] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[44] B. Chapman, P. Mehrotra, and H. Zima. Handling distributed data in Vienna Fortran procedures. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[45] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[46] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[47] C. Chase, K. Crowley, J. Saltz, and A. Reeves. Compiler and runtime support for irregularly coupled regular meshes. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[48] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[49] S. Chatterjee and G. Blelloch. VCODE: A data-parallel intermediate language. In *Frontiers'90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[50] S. Chatterjee, G. Blelloch, and A. Fisher. Size and access inference for data-parallel programs. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[51] S. Chatterjee, G. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[52] D. Chen, H. Su, and P. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[53] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[54] M. Chen and J. Cowie. Prototyping Fortran-90 compilers for massively parallel machines. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[55] M. Chen and Y. Hu. Optimizations for compiling iterative spatial loops to massively parallel machines. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[56] M. Chen and J. Wu. Optimizing FORTRAN-90 programs for data motion on massively parallel systems. Technical Report YALE/DCS/TR-882, Dept. of Computer Science, Yale University, December 1991.

[57] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C. Tseng. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. In *Frontiers'92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[58] T. Clark, R. v. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott. Evaluating parallel languages for molecular dynamics computations. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[59] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, to appear 1993.

[60] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[61] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[62] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[63] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[64] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with Platinum. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, December 1989.

[65] E. Darnell, K. Kennedy, and J. Mellor-Crummey. Automatic software cache coherence through vectorization. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[66] E. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[67] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide.* SIAM Publications, Philadelphia, PA, 1979.

[68] J. Dongarra and D. Sorensen. SCHEDULE: Tools for developing and analyzing parallel Fortran programs. In D. Gannon, L. Jamieson, and R. Douglass, editors, *The Characteristics of Parallel Algorithms.* The MIT Press, Cambridge, MA, 1987.

[69] T. Fahringer, R. Blasko, and H. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[70] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[71] I. Foster and S. Tuecke. Parallel programming with PCN. Technical Report ANL-91/32, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, September 1991.

[72] G. Fox. Achievements and prospects for parallel computing. *Concurrency: Practice & Experience*, 3(6):725–739, December 1991.

[73] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[74] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[75] E. Gabber. VMMP: A practical tool for the development of portable and efficient programs for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):304–317, July 1990.

[76] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[77] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[78] G. Geist and C. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal of Scientific Stat. Computing*, 9:639–649, 1988.

[79] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[80] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice & Experience*, 2(3):171–193, September 1990.

[81] M. Gerndt. Work distribution in parallel programs for distributed memory multiprocessors. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[82] M. Gerndt. Program analysis and transformations for message-passing programs. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[83] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[84] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 228–234, August 1984.

[85] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[86] T. Gross, S. Hinrichs, G. Lueh, D. O'Hallaron, J. Stichnoth, and J. Subhlok. Compiling task and data parallel programs for iWarp. In *Proceedings of the Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, Boulder, CO, October 1992.

[87] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[88] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[89] 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, March 1992.

[90] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication for multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[91] M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[92] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, April 1991.

[93] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[94] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[95] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[96] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers.* The MIT Press, Cambridge, MA, 1991.

[97] 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.

[98] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[99] High Performance Fortran Forum. High Performance Fortran language specification, version 0.2. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, September 1992.

[100] R. Hill. MIMDizer: A new tool for parallelization. *Supercomputing Review*, 3(4):26–28, April 1990.

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

[102] S. Hinrichs and T. Gross. Utilizing new communication features in compilation for private memory machines. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[103] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[104] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[105] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[106] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[107] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[108] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12(4), August 1991.

[109] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[110] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[111] A. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[112] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[113] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Dept. of Computer Science, Rice University, April 1991.

[114] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation in a parallelizing compiler. Technical Report TR91-174, Dept. of Computer Science, Rice University, December 1991.

[115] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Technical Report TR92-189, Dept. of Computer Science, Rice University, August 1992.

[116] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[117] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[118] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[119] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice & Experience*, to appear 1993.

[120] C. King and L. Ni. Grouping in nested loops for parallel execution on multicomputers. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[121] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[122] C. Koelbel. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, pages 101–110, Albuquerque, NM, November 1991.

[123] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[124] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[125] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[126] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic domain decomposition in BLAZE. In S. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987. Pennsylvania State University Press.

[127] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[128] P. Kogge and H. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[129] U. Kremer, H. Zima, H.-J. Bast, and M. Gerndt. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, 7:387–393, 1988.

[130] D. Kuck. *The Structure of Computers and Computations, Volume 1.* John Wiley and Sons, New York, NY, 1978.

[131] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. The structure of an advanced retargetable vectorizer. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, October 1980.

[132] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[133] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[134] H. T. Kung and J. Subhlok. A new approach for automatic parallelization of blocked linear algebra computations. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[135] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[136] B. Leasure, editor. *PCF Fortran: Language Definition, version 3.1.* The Parallel Computing Forum, Champaign, IL, August 1990.

[137] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[138] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[139] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers'90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[140] 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.

[141] 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.

[142] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computer Systems*, 7(4):321–359, November 1989.

[143] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[144] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[145] C. Lin and L. Snyder. Data ensembles in Orca-1. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[146] J. Loeliger, R. Metzger, M. Seligman, and S. Stroud. Pointer target tracking: An empirical study. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[147] S. Lucco and O. Sharp. Parallel programming with coordination structures. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[148] S. Lundstrom and G. Barnes. Controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, St. Charles, IL, August 1980.

[149] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[150] K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Rice University, April 1992.

[151] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[152] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[153] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[154] J. Merlin. ADAPTing Fortran-90 array programs for distributed memory architectures. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[155] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[156] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[157] M. Mu and J. Rice. Row oriented Gauss elimination on distributed memory multiprocessors. *International Journal of High Speed Computing*, 4(2):143–168, June 1992.

[158] E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[159] P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming language. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[160] M. O'Boyle and G. Hedayat. A transformational approach to compiling Sisal for distributed memory architectures. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[161] D. Olander and R. Schnabel. Preliminary experience in developing a parallel thin-layer Navier Stokes code and implications for parallel language design. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[162] S. Otto and M. J. Wolfe. The MetaMP approach to parallel programming. In *Frontiers'92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[163] E. Paalvast, A. van Gemund, and H. Sips. A method for parallel program generation with an application to the Booster language. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[164] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[165] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[166] Parallel Computing Forum. PCF: Parallel Fortran extensions. *Fortran Forum*, 10(3), September 1991.

[167] Parasoft Corporation. *Express User's Manual*, 1989.

[168] J. Peir and R. Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, August 1989.

[169] M. Philippsen and W. Tichy. Compiling for massively parallel machines. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[170] K. Pingali and A. Rogers. Compiling for locality. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, June 1990.

[171] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[172] R. Ponnusamy, J. Saltz, R. Das, C. Koelbel, and A. Choudhary. Embedding data mappers with distributed memory machine compilers. In *Proceedings of the Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, Boulder, CO, October 1992.

[173] A. Porterfield. *Software Methods for Improvement of Cache Performance*. PhD thesis, Rice University, May 1989.

[174] M. Quinn and P. Hatcher. Data parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, September 1990.

[175] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[176] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[177] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NY, June 1990.

[178] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[179] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[180] M. Rosing. *Efficient Language Constructs for Complex Data Parallelism on Distributed Memory Multiprocessors*. PhD thesis, Dept. of Computer Science, University of Colorado, November 1991.

[181] M. Rosing, R. Schnabel, and R. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[182] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[183] M. Rosing, R. Schnabel, and R. Weaver. Scientific programming languages for distributed memory multiprocessors: Paradigms and research issues. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[184] R. Rühl. Evaluation of compiler-generated parallel programs on three multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[185] R. Rühl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[186] G. Sabot. A compiler for a massively parallel distributed memory MIMD computer. In *Frontiers'92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[187] G. Sabot. Optimized CM Fortran compiler for the Connection Machine computer. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, Kauai, HI, January 1992.

[188] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice & Experience*, 3(6):573–592, December 1991.

[189] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[190] W. Shang and J. Fortes. Independent partitioning of algorithms with uniform dependences. In *Proceedings of the 1988 International Conference on Parallel Processing*, St. Charles, IL, August 1988.

[191] J. Sheu and T. Tai. Partitioning and mapping nested for-loops on multiprocessor systems. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[192] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12), December 1990.

[193] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[194] D. Socha. Compiling single-point iterative programs for distributed memory computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[195] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[196] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 1.0 edition, February 1991.

[197] S. Tjiang, M. E. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[198] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[199] P.-S. Tseng. A systolic array parallelizing compiler. *Journal of Parallel and Distributed Computing*, 9(2):116–127, June 1990.

[200] T. v. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[201] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[202] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, January 1984.

[203] R. Weaver and R. Schnabel. Automatic mapping and load balancing of pointer-based dynamic data structures on distributed memory machines. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[204] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[205] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[206] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[207] M. J. Wolfe. Loop rotation. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.

[208] M. J. Wolfe and C. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[209] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[210] M. Wu and G. Fox. A test suite approach for Fortran 90D compilers on MIMD distributed memory parallel computers. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[211] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[212] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[213] R. Zucker and J.-L. Baer. A performance study of memory consistency models. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.