Optimizing Fortran90D/HPF for Distributed-Memory Computers

Gerald H. Roth

COMP TR97-276

April, 1997

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

RICE UNIVERSITY

Optimizing Fortran90D/HPF for Distributed-Memory Computers

by

Gerald H. Roth

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

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor
Computer Science

John Mellor-Crummey, Faculty Fellow
Computer Science

William W. Symes, Professor
Computational and Applied Mathematics

R. Gregg Brickner, Technical Staff Member
Los Alamos National Laboratory

Houston, Texas

April, 1997

Optimizing Fortran90D/HPF for Distributed-Memory Computers

Gerald H. Roth

Abstract

High Performance Fortran (HPF), as well as its predecessor FortranD, has attracted considerable attention as a promising language for writing portable parallel programs for a wide variety of distributed-memory architectures. Programmers express data parallelism using Fortran90 array operations and use data layout directives to direct the partitioning of the data and computation among the processors of a parallel machine.

For HPF to gain acceptance as a vehicle for parallel scientific programming, it must achieve high performance on problems for which it is well suited. To achieve high performance with an HPF program on a distributed-memory parallel machine, an HPF compiler must do a superb job of translating Fortran90 data-parallel array constructs into an efficient sequence of operations that minimize the overhead associated with data movement and also maximize data locality.

This dissertation presents and analyzes a set of advanced optimizations designed to improve the execution performance of HPF programs on distributed-memory architectures. Presented is a methodology for performing deep analysis of Fortran90 programs, eliminating the reliance upon pattern matching to drive the optimizations as is done in many Fortran90 compilers. The optimizations address the overhead of data movement, both interprocessor and intraprocessor movement, that results from the translation of Fortran90 array constructs. Additional optimizations address the issues of scalarizing array assignment statements, loop fusion, and data locality. The combination of these optimizations results in a compiler that is capable of optimizing dense matrix stencil computations more completely than all previous efforts in this area. This work is distinguished by advanced compile-time analysis and optimizations performed at the whole-array level as opposed to analysis and optimization performed at the loop or array-element levels.

Acknowledgments

> We shall become Christians on that morning when we laugh and sing for the right reasons, and when we weep not because we have lost something but because we were given so much.
>
> *Padovano*

I would like to begin by thanking my thesis committee. To Ken Kennedy, my advisor, goes my thanks for his continuous support; for his willingness to listen to my ideas, problems, and concerns; for his encouraging words; and for instructing me to see the big picture. My thanks to John Mellor-Crummey for challenging me to go the extra mile and solve the larger problem. Gregg Brickner introduced me to the issues involved in compiling stencils for distributed-memory architectures. And Bill Symes, whose initial problems with getting worthwhile performance from Fortran90 on a SIMD machine served as an impetus to the research presented in this dissertation.

And then there are the many outstanding graduate students that I have had the honor to work with over the past years. In particular, the following were willing to listen and discuss issues that I encountered in my research: Steve Carr, Cliff Click, Gina Goff, Reinhard von Hanxleden, Paul Havlak, Uli Kremer, Nat McIntosh, and Mike Paleczny. My special thanks to Chau-Wen Tseng who helped in directing me to my thesis topic.

I would like to thank Iva Jean Jorgenson and Ellen Butler, without whom I would not have survived as a long-distance graduated student during the last few years of my graduate career.

Much of my financial support came from the IBM Corporation via the Graduate Resident Study Program. Without this support I would not have been able to pursue a graduate career. I would like to thank the following managers from IBM's Santa Teresa Lab who made it possible: Ann Campbell, Al Kennedy, and Beverly Moncrieff.

And finally, I would like to thank my family, to whom this dissertation is dedicated. It has been their love, patience, and understanding that kept me going through the difficult times. To Lisa, my wife and best friend, and to my children, Patrick and Jennifer, goes all my love and gratitude.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Background

Since the dawn of computing there has been a need for high performance computers to solve computationally-intensive problems in a reasonable amount of time. Many different architectures have been used over the years to create such computers. As of today, distributed-memory parallel machines offer the greatest computing power and scalability for the lowest price/performance ratio and are thus *the* choice for solving today's most computationally-intensive and memory-intensive problems. However, programming such computers is still the principal obstacle to their general acceptance.

Fortran90 is currently the best numerically-oriented programming language for expressing data parallelism. Fortran90's array syntax and rich set of array intrinsic functions has raised the level of abstraction available to Fortran programmers, thus making it the programming language of choice in science and engineering.

High Performance Fortran (HPF), an outgrowth of Fortran D, has attracted considerable attention as a promising language for writing portable parallel programs for a wide variety of distributed-memory architectures. HPF offers a simple programming model that shields programmers from the intricacies of concurrent programming and managing distributed data. Programmers express data parallelism using Fortran90 array operations and use data layout directives to direct the partitioning of the data and computation among the processors of a parallel machine. It is then the responsibility of the compiler to generate efficient code for manipulating the distributed data. Thus HPF seems to be the best candidate for programmers to use in creating machine-independent data-parallel programs for execution on distributed-memory machines.

1.2 Thesis Statement

For Fortran90D/HPF to gain acceptance as vehicles for parallel scientific programming, they must achieve high performance on problems for which they are well suited. To achieve high performance on a distributed-memory parallel machine, a compiler for Fortran90D or HPF must do a superb job of translating Fortran90 data-parallel array constructs into an efficient sequence of operations that minimize the overhead associated with data movement and also maximize data locality. Unfortunately, current Fortran90D/HPF compilers fail to exploit the heightened level of abstraction present in Fortran90 array constructs and thus fail to optimize Fortran90 programs as fully as possible.

> **Thesis:** A Fortran90D/HPF compiler for distributed-memory parallel machines can produce more efficient code if it includes advanced analysis and optimization of the program at the whole-array level.

Fortran90/HPF has raised the level of abstraction of Fortran programs. We need a similar rise in the level of analysis and transformation if we expect to do the best job possible in optimizing it. Our work is distinguished by advanced compile-time analysis and optimizations performed at the whole-array level as opposed to analysis and optimization performed at the loop or array-element levels.

1.3 Contributions

The main goal of this work is not the design of an HPF compiler for distributed-memory machines. That effort has been addressed by others, including Compass, Thinking Machines Corp., MasPar, IBM, Digital Equipment Corp., the Portland Group, and Applied Parallel Research. But while there exist several HPF compilers, few are able to optimize programs as fully as would be expected. The goal of this work is to address the deficiencies of the current class of HPF compilers. We have accomplished that by developing a set of array-level optimizations that can be incorporated into other compilers to produce better code.

Here we present a summary of the contributions this dissertation makes in the area of compiling HPF for distributed-memory machines. It is also worthwhile to note that most of the optimizations presented are equally applicable to shared-memory and scalar machines.

Analysis of Fortran90 Array Constructs

Before we can optimize Fortran90 array constructs we must be able to analyze them to determine certain facts, and we must be able to represent those facts so they are accessible to the compiler. The analysis methods that are exploited by many advanced compilers include data flow analysis and dependence analysis. Methods used to represent derived information include control flow graphs, static single assignment, and dependence direction vectors. This dissertation shows how to extend these methods and representations to support the direct analysis of operations on array sections. In particular we classify a new genre of data dependences and present transformations that utilize them.

Offset Array Optimization

Interprocessor data movement on a distributed-memory parallel machine is typically far more costly than movement within the memory of a single processor. For this reason, much of the prior research on minimizing data movement has focused on the interprocessor case. However, although interprocessor data movement is more costly per element, the number of elements moved within the memory of a single processor may be much larger, causing the cost of local data movement to be dominant.

Our offset array optimization focuses on the problem of minimizing the amount of intraprocessor data movement when performing Fortran90 array operations. We have developed a set of necessary and sufficient criteria for when this local data movement can be avoided, and we have designed an algorithm that is capable of verifying the criteria over entire procedures. This algorithm is able to avoid such data movement in many more cases than the current class of HPF compilers.

Context Optimization

With distributed-memory SIMD machines there is a need to explicitly turn processors on and off. This is due to the fact that there is only a single instruction stream and not all processors are to execute each instruction. Only processors containing data related to the current instruction should execute it. If a processor is not to execute a set of instructions, it must be explicitly “masked out”. However, changing the processor mask, or *context*, is an expensive operation. Setting the machine context is an overhead that one must pay to execute on a SIMD architecture.

We have developed two separate optimizations that address this overhead by reducing the number of times that the machine context must be set. First, we rearrange the program statements so that during code generation as many statements as possible that execute under the same context are placed within the same loop nest. We call this optimization *context partitioning*. Second, we alter the order in which array elements are processed by performing loop transformations on the generated loop nests. These transformations allow us to hoist the context setting code out of the loops and thus reduce the number of context changes. We call this optimization *context splitting*.

Even though the context partitioning optimization was originally designed to optimize code for SIMD machines, it is also beneficial for MIMD machines. First, it generates loop nests for array statements that do not require conditionals when executed on distributed-memory machines. Second, the loop nests contain more statements and thus increase the opportunities for data re-use. Third, context partitioning groups together as many communication operations as possible which facilitates analysis and thus enables a reduction in the amount of interprocessor data movement.

Advanced Scalarization

Before an array statement of a data-parallel language can be executed on the target architecture, a compiler must rewrite it such that individual array elements are accessed one at a time. This is accomplished by replacing the array statement with a loop nest that contains a subscripted reference to the array. This translation is known as *scalarization*. We have developed a new algorithm to perform scalarization in a single pass over the code, as opposed to the standard two-pass algorithm. The new algorithm has the same ability as the two-pass algorithm for avoiding the generation of array temporaries or minimizing their size when temporaries are required.

Stencil Compilation

For many HPF programs performing dense matrix computations, the main computational portion of the program belongs to a class of kernels known as stencils. We have developed a strategy for optimizing such stencil computations, no matter how they are instantiated by the programmer. The strategy combines our offset array optimization, context partitioning optimization, and advanced scalarization algorithm with a new optimization, communication unioning, that minimizes interprocessor communication for such computations. The result is a compilation scheme that is more robust than all previous work in this area.

1.4 Overview

In the next chapter, we present the necessary background material to understand the issues being addressed in this dissertation. This includes an overview of the target architectures and programming languages. In Chapter 3 we discuss related efforts and their relevance to this work. Chapter 4 presents the compilation model on which the rest of the dissertation is built. It introduces many of the issues involved in compiling a data-parallel programming language for execution on a distributed-memory machine.

In Chapter 5 we begin the main contributions of this dissertation by describing extensions to common compiler analysis techniques and representations to support the array section operations found in Fortran90. In particular, we address data-flow analysis, dependence analysis, and static single assignment. The results of these analyses are used in the subsequent optimization phases of our compiler.

Chapter 6 describes the offset array optimization which reduces the cost of intraprocessor data movement that is associated with Fortran90 shift intrinsics. Chapter 7 discusses two context optimizations for SIMD architectures. We also discuss the relationship of these context optimizations to MIMD machines. Chapter 8 presents an algorithm that is able to perform scalarization more efficiently than the standard algorithm that is commonly used.

Chapter 9 introduces a new strategy for compiling dense matrix stencil computations for distributed-memory machines. This strategy exploits several of the optimizations presented earlier in the dissertation and introduces a new optimization, communication unioning, that reduces the cost of interprocessor data movement. This chapter also includes an extended example that demonstrates the power of this compilation strategy. We conclude in Chapter 10 with a summary of our contributions and a look forward to future work.

Chapter 2

Background

In this chapter we give a brief overview of distributed-memory multiprocessor architectures, the Fortran90 language, and the data distribution/alignment directives of Fortran D and High Performance Fortran. In addition we present an overview of two differing compilation models that Fortran90D/HPF compilers use to generate code for execution on distributed-memory machines. These descriptions introduce the concepts necessary to understand how the machines work, why the compilers must generate certain code sequences, and why the optimizations described in later chapters are beneficial.

2.1 Distributed-Memory Multiprocessor Architectures

This section introduces two classes of distributed-memory multiprocessor architectures. These classes comprise two of the four classes proposed in Flynn's taxonomy of computer architectures [70]. The first is *Multiple Instruction streams, Multiple Data streams*, or **MIMD** architectures. The second is *Single Instruction stream, Multiple Data streams*, or **SIMD** architectures. We give a brief overview of these two architectures in the following subsections. For a more complete discussion of general MIMD and SIMD architectures see an appropriate computer architecture book [88, 96].

2.1.1 Distributed-Memory MIMD Architectures

A MIMD computer contains many independent central processing units (CPUs) operating asynchronously, each executing its own instruction stream. The MIMD architectures in which we are interested associate some local memory with each CPU, from which the CPU fetches instructions and reads/writes data. The CPU along with its associated memory is referred to as a *processing element* (PE) or a *processing node.*

The collection of all PEs is called the *PE array.* The PE array may be treated as a linear array or as an array of higher dimensions; *e.g.*, a 16 PE array could be treated as a 16×1 grid, or an 8×2 grid, or a 4×4 grid, or even a $2 \times 2 \times 2 \times 2$ grid (equivalent to a 4-d hypercube). For simplicity, we limit our discussions to treating the PE array as a linear array or a two dimensional square grid.

The PEs are connected by an interprocessor communication network. This network allows messages to be sent from one PE to another. These messages can be used to allow PEs to share data with one another. The details of such a network are not important to this dissertation.

Examples of MIMD machines include the SP2 from IBM [4], and the Paragon from Intel [97, 67]. See Figure 2.1 for a simple schematic of such a machine.

2.1.2 Distributed-Memory SIMD Architectures

Distributed-memory SIMD machines are quite similar to MIMD machines in that they consist of an array of PEs connected by a communication network. However, unlike MIMD machines, the PEs of a SIMD machine operate synchronously, each executing the same instruction. In a SIMD machine each PE contains an Arithmetic Logical Unit (ALU) rather than a fully functional CPU. The instructions to be executed by the ALUs are received from a serial *front end* processor or *control unit*. The front end processor has three responsibilities. The first is to drive the PE array by

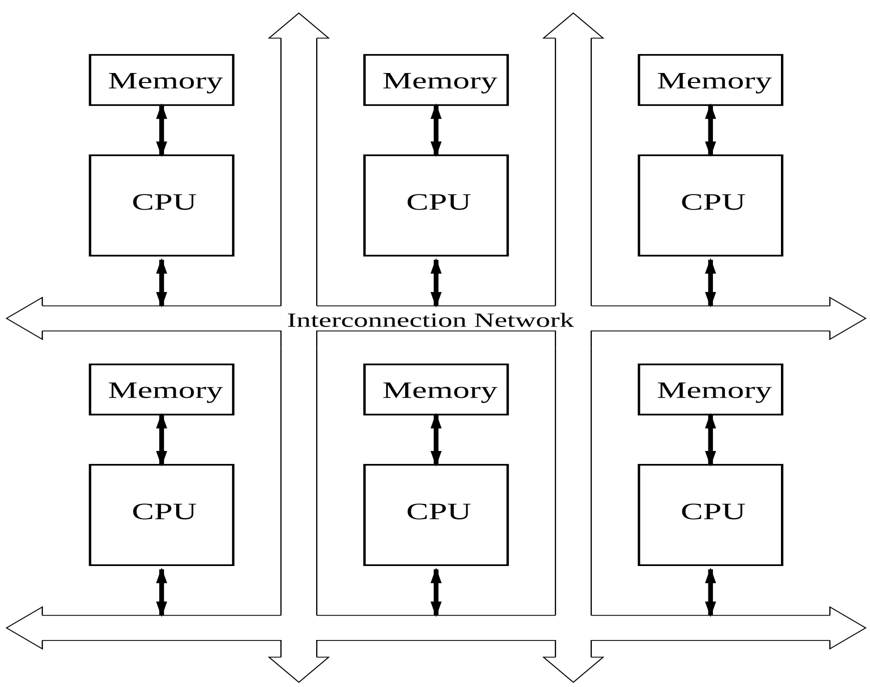


Figure 2.1 A distributed-memory MIMD architecture.

broadcasting instructions and related data to all PEs. The second is to perform all scalar computations and control flow operations. Third, the front end is the system interface to the external world.

The single instruction stream of a SIMD computer necessitates that each PE have an execution flag. This execution flag can be set on or off to indicate whether the PE should execute the current instruction. There are several reasons why a PE may be turned off during a computation. The two main reasons are that the current operation is being performed under a user specified mask (such as a Fortran90 WHERE statement), or that the PE does not have any local data on which to operate. When taken as a whole, the execution flags of all the PEs are said to determine the *context* of the PE array. Some instructions are executed regardless of the execution mask, for example instructions that reset the execution flag. It is the responsibility of the compiler to generate the code to set the correct PE context for all computations executed on the PE array.

Examples of SIMD machines include the CM-2/CM-200 from Thinking Machines Corporation [90, 148, 150], and the MP-1/MP-2 from MasPar [26, 130, 124]. A simple diagram of a SIMD machine can be seen in Figure 2.2.

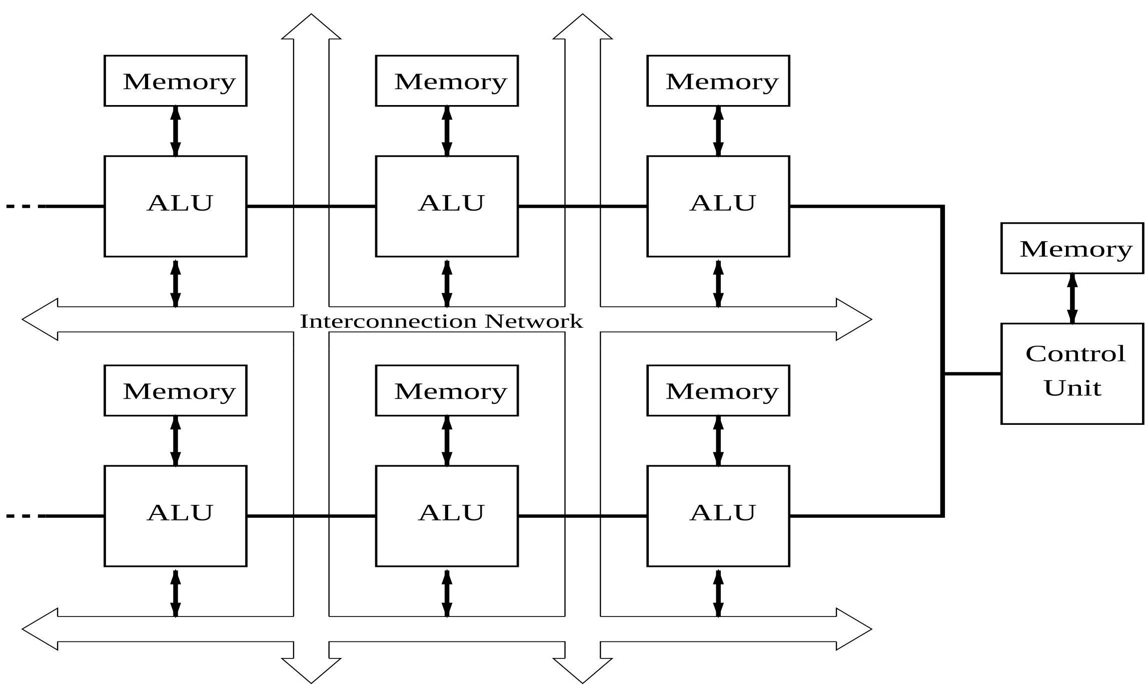


Figure 2.2 A distributed-memory SIMD architecture.

2.2 Fortran90

Fortran90 [18] adds a number of interesting and useful features to the Fortran language, the most popular being the array features. With Fortran90, entire arrays can be referenced by simply referring to the array name – no subscripts are necessary. Arrays or subsections of arrays can be specified by using *triplet* subscripts. A triplet specifies a range in the following form.

```
[lower bound]:[upper bound][:stride]
```

If the lower or upper bounds are not specified, the declared bounds of the array are assumed. The stride is 1 if not given. A *null triplet* is one in which only a single colon appears and thus refers to all the elements in the corresponding dimension of the array.

The use of arrays or array subsections in expressions and assignment statements greatly reduces the need for tedious and error-prone DO-loops. In Fortran90, arrays or array subsections are treated as unitary objects when referenced. Thus for an array assignment statement the semantics specify that all right-hand side array elements are read before any left-hand side array elements are stored.

Besides adding array references, Fortran90 added a large set of intrinsic functions that operate on or manipulate arrays. These intrinsic functions roughly fall into the following categories:

- Reduction operations: MAXVAL, MINVAL, SUM, COUNT, ALL, ANY, and PRODUCT.
- Inquiry functions: SIZE, SHAPE, LBOUND, and UBOUND.
- Array construction functions: SPREAD, MERGE, RESHAPE, PACK, and UNPACK.
- Array manipulation functions: CSHIFT, EOSHIFT, and TRANSPOSE.
- Location functions: MAXLOC and MINLOC.

Other array-related features of Fortran90 include allocatable arrays, pointers, and the WHERE statement. For more information on these or any of the above mentioned constructs, the reader is referred to an appropriate text [2, 127].

2.3 Fortran D and High Performance Fortran

Fortran D and High Performance Fortran (HPF) are versions of Fortran that have been designed to assist both the programmer and compiler in producing efficient data-parallel programs for distributed-memory machines. The main extensions found in these languages are directives for specifying how data arrays are to be distributed over the PE array.

Addressing the data distribution problem can be handled best by considering the 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's DECOMPOSITION statement or HPF's equivalent TEMPLATE statement specify an abstract problem or index domain; they do not require any storage. Each element of a decomposition represents a unit of computation. These statements declare the name, dimensionality, and size of a decomposition for later use.

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 across dimensions. The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition.

After arrays have been aligned with a decomposition, the DISTRIBUTE statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes for Fortran D are BLOCK, CYCLIC, and BLOCK_CYCLIC, whereas HPF has BLOCK and CYCLIC(i). The symbol "$*$" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine.

Entire descriptions complete with detailed examples for both Fortran D and HPF can be found elsewhere [71, 89, 112]. We use Fortran D directives in the examples throughout this dissertation with the knowledge that it is usually a trivial task to map them to the corresponding HPF directives.

2.4 Fortran90D/HPF Compilation Models

In compiling Fortran90D/HPF for execution on distributed-memory parallel architectures, most compilers use one of two compilation models. These two models use very different strategies for translating the user's program for execution on the parallel hardware. Each model has its advantages and its disadvantages. In each case we will assume that the generated code is in the form of an Fortran77 node program containing calls to a message-passing library which is then passed to the hardware vendor's compiler. We introduce these two models here, and then we use them to categorize the related projects that are discussed in Chapter 3.

2.4.1 Scalarizing Compilers

It is often the case that the quickest and easiest path to a completed software project is to leverage off of the use of existing software components. Compilers are no different. For this reason many Fortran90 compilers, and now Fortran90D/HPF compilers, are centered around pre-existing Fortran77 compilers. Such compilers consist of a Fortran90D/HPF front end that performs parsing and semantic analysis, followed by a scalarizer which translates the parallel Fortran90D/HPF code into equivalent sequential Fortran77D/HPF code, and then a Fortran77D compiler which performs program analysis, optimization, and code generation. See Figure 2.3 for an outline of a compiler which utilizes this model. This strategy is used by many of the HPF compilers for MIMD architectures [81, 85, 19], and it is not limited to compilers for distributed-memory machines [20].

The advantages of this model are fairly clear. By exploiting an existing Fortran77D compiler, a Fortran90D compiler can be created in a much shorter time span. In addition, the Fortran90D compiler gains from the years of effort that went into creating and optimizing the Fortran77/Fortran77D compiler. The scalarizer of the Fortran90D

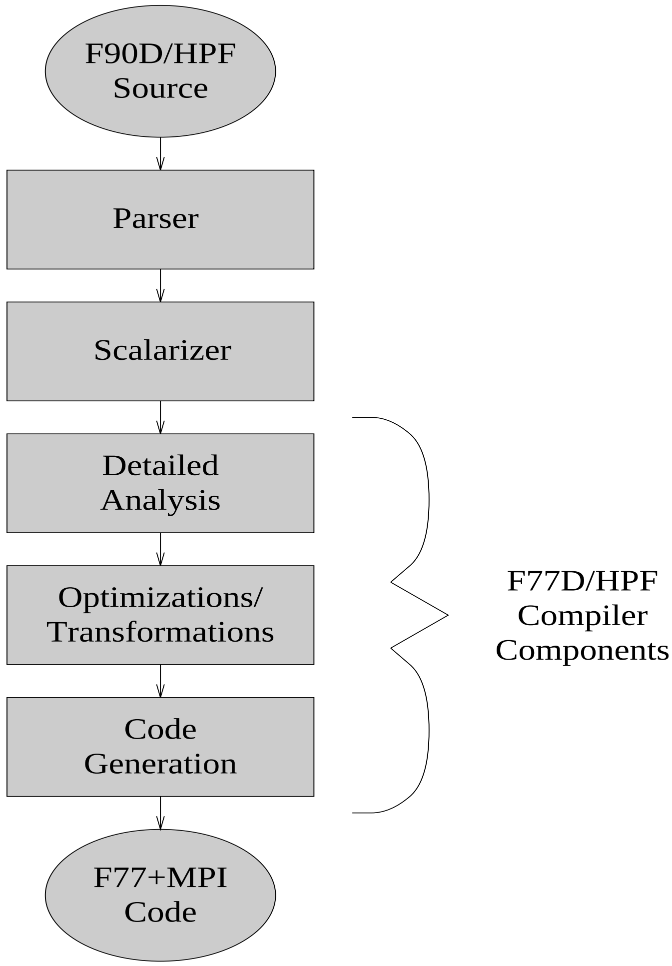


Figure 2.3 Components of a scalarizing Fortran90D/HPF compiler.

compiler does not need to be concerned with the alignment or distribution of arrays, or with the generation of communication to insure data is correctly passed between PEs – all of that is handled by the Fortran77D/HPF compiler.

And a final key advantage is that the Fortran77D compiler handles all of the incoming code uniformly, whether the code was originally written in a scalar style by the programmer or was translated into scalar code by the scalarizer. This allows for the parallelization and distribution of computation not only for code written with parallel constructs but also for parallelizable code that is not expressible using array syntax or FORALL; *e.g.*, pipelined computations [93, 152, 58].

The major drawback of this scheme is that the scalarization process can obfuscate the code, making it much more difficult to analyze and optimize than the original

Fortran90D code. Facts that were clear at the Fortran90 array level become obscured by strange loop bounds and subscript expressions. In many cases, whole-array optimizations that could have been performed on the Fortran90D program cannot be detected in the equivalent Fortran77D program. Missing the opportunity for such whole-array optimizations can lead to significant increases in both execution time and memory requirements due to compiler generated temporary arrays that could have otherwise been eliminated.

Another drawback is the increase in compile time. Since the Fortran77D/HPF compiler components only expect sequential programs as input, there are usually few methods for the scalarizer to transmit important information to it. Such lost information would include which constructs were originally written in parallel form. The Fortran77D compiler must then perform analysis to rediscover these facts so that it can produce efficient code for the parallel machine.

2.4.2 Array Operation Compilers

In contrast to the compilers that use the compilation model described in the preceding subsection, there are compilers created expressly for mapping the explicit parallelism of Fortran90 constructs to the parallel hardware. Such compilers are characterized by their ability to directly translate the data parallelism found in array expressions or FORALL statements for execution on the distributed-memory machine. This strategy is used in the majority of Fortran90D/HPF compilers for SIMD architectures [123, 140, 149], many of which are descendents of compiler technology developed by Compass [7, 9, 108, 109, 110], and is also employed by some MIMD compilers [28, 32]. We refer to such compilers as *array operation* compilers or *native* Fortran90 compilers.

The main phases of a compiler exploiting this model are depicted in Figure 2.4. After the input is parsed, the distributions and alignments of arrays are analyzed. This is used to generate required communication and to partition the computation among the PEs. And as communication operations are so expensive, an attempt is usually made to optimize them using methods such as message vectorization [21, 73, 91], message aggregation [120, 131, 152], and the exploitation of collective communication operations [120].

The major advantage of this model is its simplicity. The compiler takes the parallelism that is explicitly stated by the programmer and maps it to the parallel

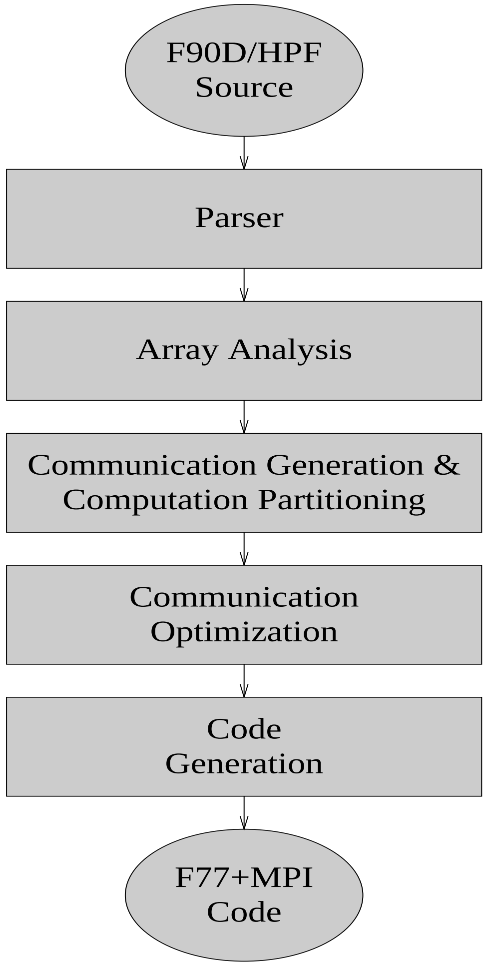


Figure 2.4 Components of an array operation Fortran90D/HPF compiler.

hardware. This typically only requires analysis of the distribution and alignment of arrays to determine that data which must be communicated prior to local computation. Often pattern matching is used to recognize and optimize the communication. However there is no need to perform in-depth analysis, such as dependence analysis, to discover parallelism.

This simplicity has its cost though, and is the major disadvantage of this compilation model. Without the information supplied by detailed analysis the compiler is unable to perform many advanced transformations or optimizations. And since the compiler only generates parallel code for parallel constructs, any code written in sequential mode is executed serially or duplicated on all processors. Thus the program must be written using array syntax, array intrinsics, or FORALL constructs wherever possible. This requires the programmer to translate all sequential code to parallel code, either by hand or using an automated tool [116, 13]. But some code sequences containing parallelism are not expressible with these constructs; *e.g.*, pipelined computations. In such cases a compiler must have dependence information to detect the available parallelism and to perform the necessary loop tiling transformations to effectively exploit it [93, 152].

2.5 Summary

In this chapter of background information we have introduced the Fortran90D and HPF languages, and have given an overview of the target distributed-memory architectures. We have also outlined two possible models that a compiler for these languages might utilize. This information provides the concepts necessary to understand the material presented in the rest of this dissertation.

Throughout the remainder of this dissertation any references to Fortran90D are meant to include HPF with data-parallel constructs, and any references to Fortran77D are meant to include HPF with only sequential constructs. As mentioned previously, all examples in this dissertation will use Fortran D syntax rather than HPF syntax.

Chapter 3

Related Work

The topic of compiling single address-space programs for execution on distributed-memory architectures has received an immense amount of attention over the past several years; both in the academic world and in industry. Initial efforts concentrated on the challenge of generating correct code. Later research tackled the task of generating efficient code.

Since this dissertation discusses the optimization of Fortran90 array constructs for execution on distributed-memory machines, we review in this chapter only those projects that also deal with the compilation & optimization of Fortran90 array features. However, it is important for the reader to remember that many of the issues involved in compiling Fortran90 for distributed-memory machines are the same for compiling Fortran77 for distributed-memory machines. In fact, most of these common issues were solved first by research projects concentrating on the compilation of Fortran77.

The interested reader is referred to any of the following projects which address the basics of distributed-memory compilers: the Fortran D compilation system at Rice University [83, 91, 92, 93, 94, 115, 152], the Parafrase-2 and Paradigm compilers at University of Illinois at Urbana-Champaign [78, 79, 80, 147, 22, 133], Vienna Fortran and the SUPERB-2 system at the University of Vienna [52, 50, 51, 68], and the SUIF project at Stanford University [16, 145, 157]. For the rest of this chapter we concentrate solely on projects whose main purpose is the compilation of Fortran90 constructs.

Compass Compilers

Compass (1961-1991) was an independent software house which was involved in the design and implementation of several SIMD compilers. The front end and the global optimizer from the original versions of both the CM Fortran and MasPar Fortran compilers were written by Compass; in fact, Compass wrote the entire Paris version of the CM Fortran compiler. The CM Fortran compiler is described in more detail below. The compilers they produced would be classified as array operation compilers as described in the preceding chapter.

The group at Compass, along with their associates at Thinking Machines Corp., were the first to investigate the challenges of compiling Fortran90-style data-parallel constructs for execution on distributed-memory machines [7, 8, 9, 110]. Together they created what many would consider to be the first, commercially viable, distributed-memory compiler.

In addition to the general SIMD compiler development effort, Compass did much of the ground-breaking research in the area of *data optimization* [108, 109, 107, 111, 122]. The purpose of data optimization is to automatically align data to improve locality and thus minimize interprocessor communication. Their method assumes an unlimited number of *virtual processors*. Then based on usage patterns, it maps arrays to the virtual processors, striving to align them so that communication costs are minimized. A later stage of the compiler then uses *strip mining* to map the virtual processors to the physical processors [156], also known as *array distribution*. This two stage approach makes each stage conceptually clean, but prevents them from interacting.

CM Fortran

Thinking Machines Corporation developed three generations of distributed-memory architectures, the first two being SIMD machines (the CM-1 and CM-2) [148] and the third being a MIMD machine (the CM-5) [132, 151]. CM Fortran, their Fortran derivative, was an implementation of Fortran77 augmented with array constructs from Fortran90. Their compiler for CM Fortran was also developed through three generations. The first generation was the *Paris*, or *fieldwise*, compiler which uses the bit-serial processors on the CM-1 and CM-2. The second generation CM Fortran compiler was the *slicewise* compiler [140]. The slicewise compiler ignored the bit-serial processors and used only the floating-point accelerator chips of the CM-2. The CM Fortran compiler can take advantage of the slicewise model of the machine in different ways to improve program performance for many engineering and scientific applications. The third generation compiler targeted the CM-5, and was basically an updated version of the slicewise compiler [139, 141]. As such, it treated the CM-5 as a SIMD machine and was thus not able to produce code that took full advantage of the architecture.

Even though the slicewise compiler gave improved performance, it also had several weaknesses. The compiler's shortcomings included the lack of transformations to increase the size of elemental code blocks, inefficient use of memory for compiler temporary arrays, and generation of poor code for communication along serial dimensions. Thinking Machines documented many of the shortcomings, and suggested methods that programmers may use to work around them [142].

Thinking Machines had also done some extensive work on compiling stencils [39]. A *stencil* is a computational pattern that calculates a new value for a matrix element by combining elements from neighboring matrix locations. The proper handling of stencils is very important for distributed-memory compilers, and can result in substantial performance gains. The performance gains produced by the CM-2 stencil compiler were obtained by using multiwire NEWS communication, eliminating memory-to-memory copying of data, and full exploitation of the floating-point registers. However, it was the responsibility of the programmer to identify a stencil computation, isolate it into a separate subroutine, and compile it with a special compiler.

NPAC's Fortran90D

The Fortran90D compiler developed by the Northeast Parallel Architectures Center (NPAC) at Syracuse University, like the CM Fortran compiler, is classified as an array operation compiler [28, 29, 30, 61]. The compiler only exploits the parallelism expressed in the data parallel constructs. It does not attempt to parallelize scalar constructs as would a Fortran77D compiler.

The basic structure of this compiler is composed of four major modules – parsing, partitioning, communication generation, and code generation. The parsing module translates all parallel constructs, such as array assignments and WHERE statements, into equivalent FORALL constructs [27]. In this way all subsequent modules need only deal with FORALL statements. The final result is a loosely synchronous single-program multiple-data (SPMD) program, structured as alternating phases of local computation and global communication.

Unlike most optimizing compilers, which rely upon program analysis to drive their transformations and optimizations, NPAC's Fortran90D compiler relies upon pattern matching:

> "The foundation of our design lies in recognizing commonly occurring computation and communication patterns. These patterns are then replaced by calls to the optimized run-time support system routines. The runtime support system includes parallel intrinsic functions, data distribution functions, communication primitives, and several other miscellaneous routines." [30]

This compiler's reliance on run-time support is evident in the fact that its run-time library contains over 500 routines. This shifting of responsibility from compile-time to run-time has its advantages and its disadvantages. The major advantage is that it simplifies the design and development of the compiler. Unfortunately it typically cannot generate as efficient code since it is difficult to perform optimizing transformations across the numerous call sites. The same can be said about relying upon pattern matching to drive transformations and optimizations. It simplifies the compiler, and when it works it works surprisingly well. But when it cannot match a pattern, the code produced is mediocre at best. An example of the vast differences in code quality produced by such compilers is presented later in this dissertation.

xlhpf

IBM's xlhpf compiler [81] is naturally classified as a scalarizing compiler as defined in Section 2.4.1. This is due to the fact that the first action taken after the intermediate representation is created is the scalarization of the array language into Fortran77 scalar form. Data dependence analysis is then performed to detect and exploit parallelism across all the code, whether it was originally written with Fortran90 parallel constructs or Fortran77 scalar constructs.

The xlhpf compiler's strengths come from its ability to perform deep program analysis. As previously mentioned the compiler uses dependence analysis to detect and exploit parallelism even in scalar code. Dependence information is also used for communication placement optimizations and selective loop distribution. The compiler uses symbolic analysis to generate efficient code even when the size of arrays or the number of processors is not known at compile-time. The analysis of data *availability* enables the elimination of redundant communication [82]. And static single assignment (SSA) form is exploited to produce an efficient mapping of scalar variables.

Taking just the opposite tack as the NPAC Fortran90D compiler, xlhpf actually in-lines library routines such as Fortran90 intrinsic functions. This enhances the opportunities for analysis and makes it possible to eliminate extra array temporaries and copying – an important consideration for any Fortran90 compiler which desires to achieve performance comparable to Fortran77 programs.

pghpf

The pghpf compiler, from The Portland Group Inc., is very similar in design to NPAC's Fortran90D compiler [32, 31, 126]. This is no surprise given the close association the two groups have had during recent years. The compiler still relies heavily on run-time routines, which enables it to target both shared-memory and distributed-memory machines. However the compiler does have additional analysis capabilities which enable it to perform optimizations not possible in NPAC's Fortran90D compiler.

The additional analysis performed by the compiler allows it to include optimizations such as heuristic-based data optimization, communication parallelization for replicated arrays, and communication unioning. Interprocedural analysis is performed which enables the global sharing of run-time data structures. By reusing run-time data structures used for communication scheduling the cost of creating and destroying such structures can be eliminated in many cases.

The compiler is also capable of recognizing and optimizing single-statement stencil computations. And finally, the compiler performs enhanced analysis and optimization of the resulting node program, including vectorizing loops for certain architectures.

xHPF

Applied Parallel Research Inc.'s xHPF translator is the most recent addition to the FORGE90 parallel programming environment [19, 146, 75]. The system is identical to the company's xHPF77 system, with the addition of a preprocessor that converts Fortran90 syntax into Fortran77. This structure classifies the compiler as a scalarizing HPF compiler, similar to IBM's xlhpf. Consistent with the design of other scalarizing HPF compilers, xHPF has the ability to produce parallel code for both Fortran90 and Fortran77 constructs.

The FORGE90 environment is an integrated system built around a set of databases. The databases hold the results of symbolic analysis, control flow analysis, and data flow analysis, as well as information regarding user-supplied directives. These databases contain information about all the procedures associated with an application. This allows the system to perform deep interprocedural analysis, thus enabling the parallelization of loops containing subroutine calls. The main limitation of the system is that the base distributed memory parallelization tool (DMP) does not attempt code restructuring transformations to enhance the detection or generation of parallel code.

Adaptor

Adaptor is an HPF-like compilation system that is structured very much like the CM Fortran compiler [33, 34, 35]. It only takes advantage of parallelism present in the explicitly parallel HPF constructs. It has no feature for automatic parallelization. In addition to batch compilation support, Adaptor also has a graphical user environment which allows the user to assist in directing the translation process.

Adaptor's compilation strategy can be split into three main phases. After the abstract syntax tree is generated and normalized, the compiler splits program statements in local and non-local operations. This step creates temporary variables and arrays as necessary. In the next phase, all parallel array operations are translated into an internal FORALL format and these resulting loops are analyzed. In the final phase, parallel loops are adjusted so that they only work on local data, communication statements are generated, and the final node program is produced. The system concentrates on producing correct code. It does not contain an optimization phase.

Fortran90-Y

The Fortran-90-Y compiler, developed at Yale University, is designed to support rapid prototyping of compilation and optimization techniques [57, 59, 58]. The compiler uses an abstract semantic algebra, *Yale Intermediate Representation* (YR), as its intermediate language. YR defines a series of semantic domains and sets of operators within each domain, and combines them with *shapes* that represent iteration spaces. The compiler optimizes a program by performing a sequence of source-to-source transformations over the YR code. It produces code for the CM-2 and the CM-5 that is comparable to that of the CM Fortran compiler.

Chapter 4

Compilation Model

In this chapter we describe in detail our overall compilation model for Fortran90D. This model is a hybrid of the two models presented in Chapter 2. As such we believe that it exploits the advantages of each model while minimizing their weaknesses. It accomplishes that by interleaving Fortran90 compilation issues with Fortran77 compilation issues.

This model, however, is not a central contribution of this dissertation. Instead, it gives us a framework on which we can apply and explain our optimizations. The optimizations themselves are not tied strictly to this model – they will work with either of the models presented in Chapter 2 or any hybrid of them.

4.1 Introduction

Figure 4.1 contains an outline of our compilation model for Fortran90D. The outline shows how some aspects of the scalarizing compiler have been combined with those of the array operation compiler to produce a unified compilation system.

Once the Fortran90D input program has been parsed, we use the Fortran D directives to determine how the data arrays are to be distributed across the PEs. After the distribution of the arrays, we look at the sequential code that exists in the program to determine which portions of it can be parallelized. Computation partitioning and communication generation come next. We then perform our high-level analysis and optimizations, which are followed by lower-level optimizations. Finally subgrid loops are generated as the last step before code generation. Each of these steps, with the exception of parsing, is explained in the remainder of this chapter.

Readers familiar with the SIMD compiler technology developed at Compass Inc. will notice that our model is an extended version of their Fortran90 compilation scheme. They were the first to use the array operation compilation model presented in Section 2.4.2 to create a compiler for distributed-memory machines. Their work greatly influenced subsequent research projects and commercial products (*e.g.*, the

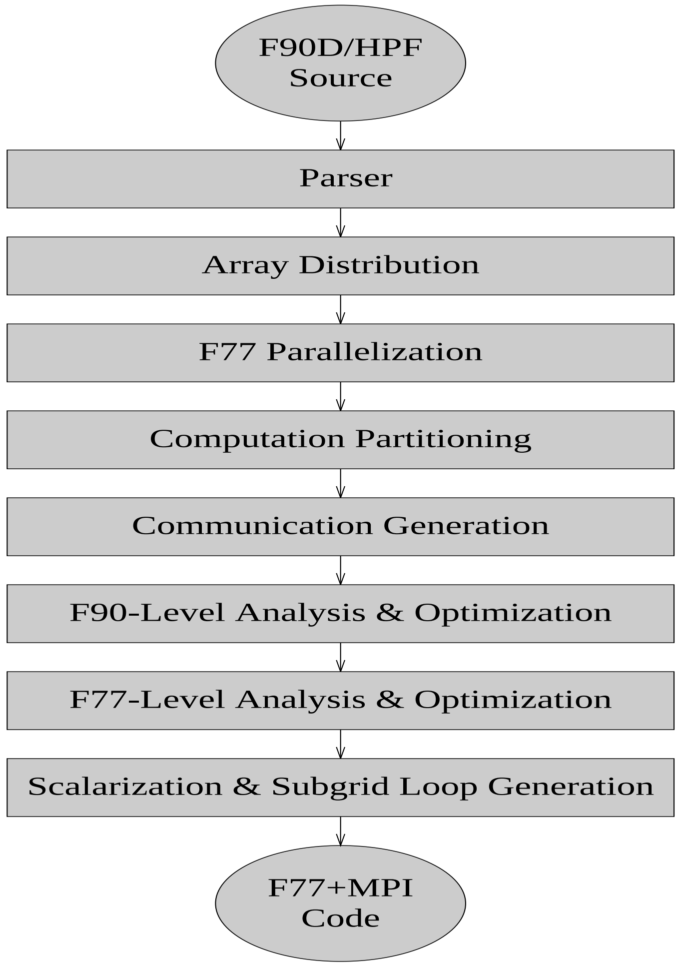


Figure 4.1 Our Fortran90D compilation model.

CM Fortran and MasPar Fortran compilers). Many of our optimizations were originally developed to address weaknesses found in these compilers.

4.2 Array Distribution

To exploit parallelism, the Fortran90D compiler distributes the data arrays across the PE array, giving each PE a chunk of the data to process. The manner in which arrays are distributed is very important for maximizing parallelism while minimizing expensive communication operations. As introduced earlier, Fortran D supplies three statements for specifying a data distribution: DECOMPOSITION, ALIGN, and DISTRIBUTE. When arrays are distributed across the PE array, each PE will locally allocate an equal-sized *subgrid* to hold its portion of the distributed array. The rank of the subgrid matches the rank of the distributed array. The extent of the i-th subgrid dimension is $Extent_i = \lceil N_i/P_i \rceil$, where N_i and P_i are the extents of the distributed array dimension and the PE array dimension, respectively. $P_i = 1$ for dimensions which are not distributed. The PE array itself is considered to have a rank equal to the number of distributed dimensions of the distributed array. To simplify our discussion, we limit the number of distributed dimensions to two.

The compiler uses a *distribution function* [92] to calculate the mapping of an array element to a subgrid location within a PE. Given an array `A`, the distribution function $\mu_A(\vec{\imath})$ maps an array index $\vec{\imath}$ into a pair consisting of a PE index $\vec{pid}$ and a subgrid index $\vec{\jmath}$. Inverse distribution functions, $\mu_A^{-1}(\vec{pid}, \vec{\jmath})$, give the reverse mapping. Table 4.1 shows examples of distribution functions and their inverses for one-dimensional arrays with either BLOCK or CYCLIC distributions. In this dissertation all arrays, whether user arrays or the PE array, use one-based indexing.

The Fortran D code in Figure 4.2 illustrates the concepts of data distribution. Given a distributed-memory machine with $P = 16$ PEs, the compiler would distribute array `X` as shown in Figure 4.3. Each PE would allocate a local subgrid $X'(16)$. The distribution function for `X` is:

$$\mu_X(i) \;=\; ((i-1) \bmod 16 + 1, \lceil i/16 \rceil).$$

On the same machine, array `Y` would be distributed by the compiler as shown in Figure 4.4. Notice how the PE array is now treated as a 4×4 matrix of PEs; *i.e.*, $P_1 = P_2 = 4$. Thus $Extent_1 = Extent_2 = 5$ and each PE would allocate $Y'(5,5)$ as the local subgrid. In this case the distribution function for `Y` is:

$$\mu_Y(i,j) \;=\; ((\lceil i/5 \rceil, \lceil j/5 \rceil), ((i-1) \bmod 5 + 1, (j-1) \bmod 5 + 1)).$$

```
REAL X(256), Y(20,20)
DECOMPOSITION A(256), B(20,20)
ALIGN X(I) WITH A(I)
ALIGN Y(I,J) WITH B(I,J)
DISTRIBUTE A(CYCLIC)
DISTRIBUTE B(BLOCK,BLOCK)
```

Figure 4.2 Fortran D code declaring two distributed arrays.

	$\mu(i)$	$\mu^{-1}(pid, j)$
BLOCK	$(\lceil i/Extent_1 \rceil, (i-1) \bmod Extent_1 + 1)$	$(pid - 1) * Extent_1 + j$
CYCLIC	$((i-1) \bmod P_1 + 1, \lceil i/P_1 \rceil)$	$(j-1) * P_1 + pid$

Table 4.1 Distribution functions and their inverses.

4.3 Parallelism Detection

One of the biggest drawbacks of the array operation compilation model presented in Section 2.4.2 was its inability to exploit the parallelism that is present in sequential code. To rectify this shortcoming, our compilation model includes a phase that identifies any serial operations on distributed arrays that are safe to execute in parallel. The search for parallelism is simplified by considering only those loop nests which iterate over the distributed dimensions of the arrays.

The methods used to detect and exploit data parallelism in sequential code have been well documented [12, 13, 25, 153, 161, 163] and will not be discussed any further here.

4.4 Computation Partitioning

The next step is to map the parallel operations to the processors. The Fortran D compiler uses the "owner computes" rule, where every processor only performs computations that update data it owns [45, 165]. In essence, the data distribution specified by the programmer is also a specification for distributing the computation. The compiler uses the distribution functions discussed in Section 4.2 to determine ownership.

The owner computes rule could be replaced by a *data optimization* phase, which may determine an alternate distribution for the computation and associated intermediate results, in an attempt to reduce the amount of communication [7, 56, 109, 119].

4.5 Communication Generation

Once data and computation distributions are finalized, the compiler must insert any necessary communication operations. These are required to move data so that all operands of an expression reside on the PE which performs the computation. A Fortran77D compiler [152] generates individual SEND and RECEIVE pairs for non-

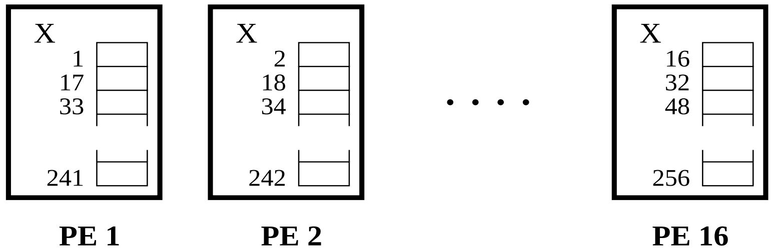


Figure 4.3 A one-dimensional array with 256 elements mapped in a CYCLIC manner onto a 16 PE machine.

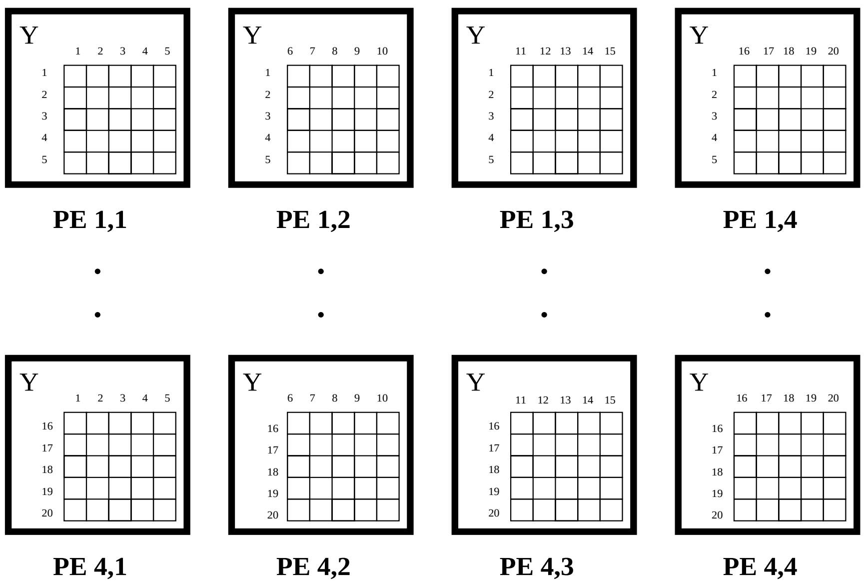


Figure 4.4 A 20×20 two-dimensional array mapped in a (BLOCK,BLOCK) fashion onto a 16 PE machine configured as a 4×4 matrix.

local data accesses, and then depend upon later compilation phases to optimize them. Our compiler model relies upon the same mechanism to handle serial source code.

However, Fortran90 array constructs supply additional information which can allow a compiler to directly recognize and exploit collective communication routines. Examples of collective communication routines include CSHIFT and TRANSPOSE. These routines have several features which provide compelling reasons for a Fortran90D compiler to exploit them:

1. *Performance:* These routines, implemented in the runtime library, are highly optimized for the target machine. Since these routines are written by hand they can often exploit architectural features that are not portable and thus not

used in higher level compilers, but which can provide significant performance advantages.

2. *Compiler complexity:* Since these routines move data in a collective manner, there is no need for subsequent compilation phases that attempt optimizations such as message vectorization or message aggregation. This greatly decreases the complexity of the compiler.

3. *Composition:* Collective communication operations may be combined to handle complex communication patterns. Such compositions usually execute faster than point-to-point communication operations generated by a compiler.

For these reasons a Fortran90D compiler should exploit collective communication routines whenever possible. This requires the compiler to recognize applicable patterns in the array syntax used in assignment statements. Our compiler uses a variant of the pattern matching techniques proposed by Li and Chen [120]. This requires an analysis of the array subscripts that are used, in conjunction with information about the array's alignment and distribution.

After the communication operations have been inserted, all computations reference data that are strictly local to the associated PEs. For example, the array assignment statement:

```
X(2:255) = X(1:254) + X(2:255) + X(3:256)
```

would be changed into the following three statements, where `TMP1` and `TMP2` are arrays that match the size and distribution of `X`:

```
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
X(2:255) = TMP1(2:255) + X(2:255) + TMP2(2:255)
```

Notice that in the third statement all the operands are "perfectly aligned" with each other and that there is no further communication required to compute the expression or store the result. This code is equivalent to the code produced by several other commercial and research compilers [34, 110, 139].

4.6 Fortran90-level Analysis and Optimization

In this phase of the compilation process, we perform array-level analysis and optimizations. The array-level analyses performed are those that are necessary to support our optimizations, and include data flow analysis, dependence analysis, and the generation of static single assignment form. These are discussed in Chapter 5.

The optimizations performed during this phase are the main contributions of this dissertation and are presented in Chapters 6 – 9. These optimizations address some of the overheads associated with compiling Fortran90 for distributed-memory computers, as well as addressing the performance of the compiler itself.

4.7 Fortran77-level Analysis and Optimization

After performing our high-level array optimizations, we turn to optimizing the portions of the program containing DO-loops and array-element references. As with the optimizations we performed at the array level, these optimizations address the many issues involved with generating efficient code for distributed-memory machines. These optimizations fall into several categories:

Reducing Communication: Here we perform optimizations that attempt to reduce the amount of communication. These include message vectorization [21, 73], message coalescing [95], message aggregation [120, 131], redundant communication elimination [17], and the exploitation of collective communication [120].

Hiding Communication: These transformations attempt to hide the cost of communication by overlapping communication and computation. Examples of such optimizations are communication placement [49, 84], message pipelining [136], vector message pipelining [95], and iteration reordering [113].

Improving Parallelism: Recognizing reductions and parallel-prefix scan operations [53, 114] can help to improve the available parallelism. Loop interchange and strip-mining [13, 160, 162] can be used to adjust the granularity of pipelined computations to balance parallelism and communication [93].

Storage Management: The use of overlap areas [73] and hash tables can ease the details of buffer management for certain types of computations. Message blocking [95] can be used in situations where buffer storage is limited.

For details on these optimizations see the individual citations or refer to Tseng's dissertation [152] where they are all discussed in terms of an optimizing Fortran77D compiler.

4.8 Scalarization and Subgrid Looping

As a final step in the compilation process the compiler must generate code that iterates over the subgrids allocated to the individual PEs. This step is composed of several sub-steps. First, array expressions are scalarized into serial DO-loops, and these loops are fused when possible. Next the loops bounds of all parallel loops are lowered so that they iterate only over the local subgrids. Then guard statements or context switching code is produced as needed depending upon the target architecture. Each of these sub-steps is discussed below. The result is a Fortran77 SPMD program that is ready for compilation and execution on the parallel machine.

4.8.1 Scalarization

The compiler translates the parallelism that is explicit in the Fortran90 array syntax into code that manipulates the arrays that have been distributed across the PEs. Since each PE is in fact a serial processor, the array expressions must be *scalarized*; *i.e.*, translated into serial code [12, 14, 161]. This process replaces the array expression with a loop nest containing array references with only scalar subscripts.

As an example, the array assignment statement

```
X(1:256) = X(1:256) + 1.0
```

would be translated into the following loop

```
DO I = 1, 256
   X(I) = X(I) + 1.0
ENDDO
```

which iterates over all 256 elements of the array `X`.

Unfortunately, the naive translation of array statements into serial loops is not always safe. Recall from Section 2.2 that the semantics for an array assignment statement specify that all right-hand side array elements are read before any left-hand side array elements are stored. Thus a naive translation of

```
X(2:255) = X(1:254) + X(2:255) + X(3:256)
```

into the following loop nest

```
DO I = 2, 255
   X(I) = X(I-1) + X(I) + X(I+1)
ENDDO
```

is incorrect, since on the second and subsequent iterations of the `I` loop the reference `X(I-1)` accesses the new values of the array `X` assigned on the previous iteration.

Fortunately, we never encounter this situation within our compilation model in its current instantiation. Recall from Section 4.5 that the generation of communication operations, with their associated temporary arrays, results in array expressions whose operands are all perfectly aligned. This property ensures that a naive scalarization of array assignment statements is always correct. The above example, after communication generation and scalarization, will look like:

```
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
DO I = 2, 255
   X(I) = TMP1(I) + X(I) + TMP2(I)
ENDDO
```

After the array expressions have been scalarized, the program contains many separate loop nests, each containing a single scalar assignment statement. Such programs exhibit poor temporal data locality and a high loop-overhead/computation ratio. To address these problems our compiler uses *loop fusion* [10] to merge multiple loop nests covering the same iteration space into a single loop nest.

The fusion of loops, however, is not always safe. A data dependence between two adjacent loops is called *fusion-preventing* if after fusion the direction of the dependence is reversed [1, 154]. The existence of such a dependence means that fusion is not safe. In our current model however, no such fusion-preventing dependences can exist between adjacent scalarized loops. This is due to the fact that the generation of communication causes all subgrid loops to operate on “perfectly aligned” data. Any fusion-preventing dependence that existed prior to communication generation is now carried through a communication operation and its compiler temporary. This communication operation prevents the Fortran90 array expressions (and their corresponding scalarized loops) from becoming adjacent and are thus not considered for loop fusion.

For example, given the following two array assignments

```
X(2:255) = X(2:255) + A(2:255)
B(2:255) = X(1:254) + B(2:255) + X(3:256)
```

communication generation would result in

```
X(2:255) = X(2:255) + A(2:255)
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
B(2:255) = TMP1(2:255) + B(2:255) + TMP2(2:255)
```

in which the fusion-preventing dependences are now carried by the temporary arrays. The definitions of the temporary arrays prevent the scalarized loops for the array assignments from becoming adjacent.

Due to this perfect alignment of data within array operations, our compiler can directly generate a single loop nest for adjacent Fortran90 array statements if they have identical distributions and cover the same iteration space. We call such array statements *congruent* [1]. This form of scalarization precludes the need for loop fusion. For example, when presented with the following array statements

```
X(1:256) = X(1:256) + 1.0
A(1:256) = X(1:256) ** 2.0
B(1:256) = X(1:256) + A(1:256) + B(1:256)
```

our compiler would directly generate the following loop during scalarization.

```
DO I = 1, 256
   X(I) = X(I) + 1.0
   A(I) = X(I) ** 2.0
   B(I) = X(I) + A(I) + B(I)
ENDDO
```

4.8.2 Subgrid Looping

After array statements have been scalarized into DO-loops, we need to adjust the bounds of the loops so that they iterate over the subgrid that is assigned to each PE. The resulting loop is known as the *subgrid loop*. Some Fortran90 compilers combine scalarization and loop bounds reduction into a single step [156]. We have separated them since our compiler needs to also lower the bounds of Fortran77 loops written by the programmer.

Continuing our previous example `X(1:256) = X(1:256) + 1.0`, where the array `X` is distributed as in Figure 4.3, scalarization generates the following loop.

```
DO I = 1, 256
   X(I) = X(I) + 1.0
ENDDO
```

[1] Congruence is a stronger restriction than conformance [142], which just considers shape and size.

Loop bounds reduction alters this loop to create the following loop,

```
DO I = 1, Extent₁        !  Extent₁ = 16
   X′(I) = X′(I) + 1.0
ENDDO
```

where $\mathtt{Extent}_1$ is the size of the subgrid and is calculated as described in Section 4.2. Recall that `X′` is the local subgrid instantiation of the distributed array `X`.

For MIMD architectures each PE executes the subgrid loop asynchronously. For SIMD architectures the execution of the subgrid loop is a cooperative effort between the FE and the PE array. The FE handles the control flow by executing the looping construct. For each iteration of the loop, the FE then broadcasts instructions to the PE array. The broadcast instructions result in each PE adding 1.0 to `X′(I)`, where the value of `I` is also broadcast from the FE.

4.8.3 Guard Statements and Context Switching

Up to this point all of our detailed examples have used arrays that, when distributed, gave an equal number of elements per PE, and all expressions involving the arrays have accessed the entire array. When either of these conditions is not met, the compiler has to modify the subgrid loop so that only the desired subgrid elements are processed.

Given the Fortran D declarations in Figure 4.2, assume we now encounter the array assignment statement `X(2:242) = X(2:242) + 1.0`, which increments 241 elements of `X` starting with the second element. As illustrated in Figure 4.3, PE 1 holds 15 of these elements in `X′(2:16)`, PE 2's full subgrid is involved, and PEs 3 through 16 each have affected elements in `X′(1:15)`. The subgrid loop for this statement must only modify the affected subgrid elements.

The manner in which this is done depends upon the target architecture. For MIMD machines we can further refine the subgrid loop bounds or introduce guard statements. For SIMD machines we need to insert code to set the execution flags of the PEs.

MIMD Loop Bounds Refinement and Guard Introduction

When the target architecture is a MIMD machine, each PE has the capability to determine the values of the loop bounds for the subgrid portion it owns. Thus in many cases the compiler needs only to generate code that dynamically sets the loop bounds on each PE. The formulae in Table 4.2 show the calculations necessary to

	BLOCK	CYCLIC
Lower Bound	max((pid$_i$-1)$*$Ext$_i$+1,L)-(pid$_i$-1)$*$Ext$_i$	lb$_i$ = ((L–1)/P$_i$)+1 if ((pid$_i$–1) $<$ MOD(L–1,P$_i$)) lb$_i$ = lb$_i$ + 1
Upper Bound	min(pid$_i$$*Ext_i$,U)-(pid$_i$-1)$*Ext_i$	ub$_i$ = ((U–1)/P$_i$)+1 if ((pid$_i$–1) $>$ MOD(U–1,P$_i$)) ub$_i$ = ub$_i$ – 1

Table 4.2 Loop Bounds Refinement

compute the loop bounds when a subsection of an array is being referenced. The values for `pid`$_i$, `Ext`$_i$ (an abbreviation for `Extent`$_i$), and `P`$_i$ are as defined in Section 4.2 Recall that we use 1-based indexing for all arrays, including the PE array and the local subgrids.

For the example of incrementing the elements `X(2:242)`, the following code is produced:

```
LB1 = ((2-1)/P1) + 1
IF ((pid1-1) < MOD(2-1,P1)) LB1 = LB1 + 1
UB1 = ((242-1)/P1) + 1
IF ((pid1-1) > MOD(242-1,P1)) UB1 = UB1 - 1
DO I = LB1, UB1
   X'(I) = X'(I) + 1.0
ENDDO
```

Although the reduction of loop bounds handles the majority of situations arising from the use of Fortran90 array syntax, there are times when it is insufficient. For such complex cases it may be necessary to compute the local iteration sets [54, 105] or it may require the insertion of explicit guards [45, 99, 136]. We do not discuss the details of these at this point.

SIMD Context Switching

Since SIMD machines have a single instruction stream, the subgrid iteration space must include the union of all the iteration spaces required by the individual PEs. The subgrid loop must then contain code to enable and disable different sets of PEs depending upon which subgrid element is being processed. This is accomplished by changing the context of the PE array by setting each PE's execution flag as described in Section 2.1.2.

To determine the set of PEs to enable we employ the inverse distribution functions described in Section 4.2. The context switching code added to our subgrid loop that increments X(2:242) is:

```
DO I = 1, Extent₁        !  Extent₁ = 16
   Set_Context(((I-1)*P₁ + pid₁ ≥ 2) .AND. ((I-1)*P₁ + pid₁ ≤ 242))
   X'(I) = X'(I) + 1.0
ENDDO
```

P_i is the number of processors, while pid_i and $Extent_i$ are as defined previously. The function Set_Context causes each PE to evaluate the logical expression and enable its execution flag if the result is true, otherwise the execution flag is disabled.

In a similar manner, operations on arrays that do not "evenly" fill the machine require context switching code to be inserted into the subgrid loop. Compare the Fortran D declarations in Figure 4.5 to those in Figure 4.2. On the same 16 processor machine, array Y2 would be distributed as seen in Figure 4.6. Looking at Figure 4.6, it can be seen that array Y2 is distributed such that:

- PEs (1:3,1:3) each contain a full subgrid of data,
- PEs (1:3,4) contain data only in the left portion of their subgrids,
- PEs (4,1:3) contain data only in the upper portion of their subgrids,
- PE (4,4) contains data only in the upper-left corner of its subgrid.

Since different PEs contain data in different subgrid locations, the subgrid loop must contain code to change the context depending upon which subgrid element is being processed. Again, the inverse distribution functions determine the active set of PEs. For example, the Fortran90 statement Y2 = ABS(Y2) would generate the following subgrid loop nest:

```
DO J = 1, Extent₂            !  Extent₂ = 5
   DO I = 1, Extent₁         !  Extent₁ = 5
     Set_Context (((pid₁-1) * Extent₁ + I ≤ 17) .AND.
                  ((pid₂-1) * Extent₂ + J ≤ 19))
     Y2'(I,J) = ABS ( Y2'(I,J) )
   ENDDO
ENDDO
```

$Extent_i$ and pid_i are the subgrid extent and PE index, respectively, along the i-th dimension.

```
REAL Y2(17,19)
DECOMPOSITION B2(17,19)
ALIGN Y2(I,J) WITH B2(I,J)
DISTRIBUTE B2(BLOCK,BLOCK)
```

Figure 4.5 Fortran D code declaring an odd-shaped array.

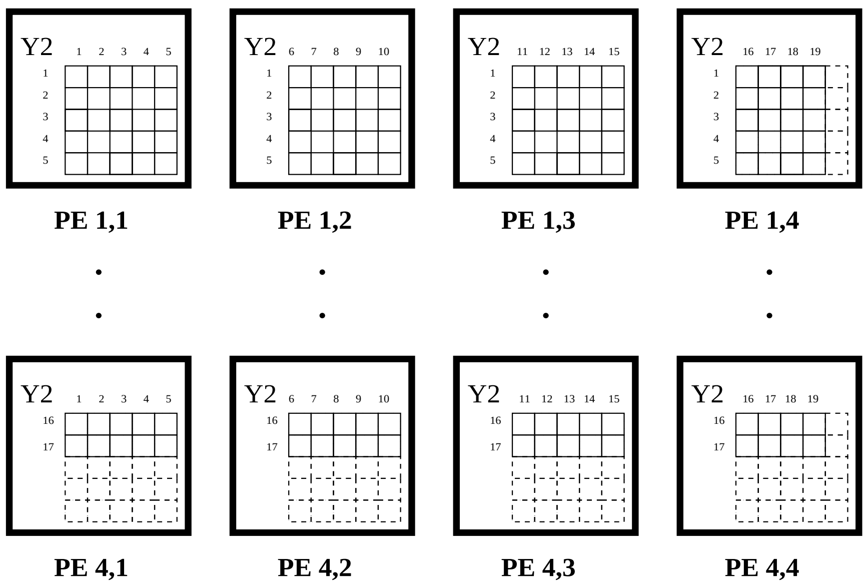


Figure 4.6 A 17×19 two-dimensional array mapped in a BLOCK fashion onto a 16 PE machine configured as a 4×4 matrix.

It should be obvious by now that the code required for changing the context can be quite complex. If the previous example had only operated on a subrange of the array `Y2`, for example, then the call to `Set_Context` would also have required a lower bound check for each dimension. This would double the number of logical operations.

We do have one mitigating factor. When a single subgrid loop nest is generated for a set of adjacent congruent array statements, we do not need to change the context for each individual statement in the subgrid loop; *i.e.*, they all operate under the same context. This is true by our definition of congruent array statements: they have the same iteration space and operate on arrays that have identical distributions.

4.9 Summary

In this chapter we have presented our compilation model for compiling Fortran90D programs for distributed-memory architectures. This model establishes a foundation on which we build our optimizations and transformations. However, it is important to reiterate the fact that our optimizations are quite general and can be applied to many different Fortran90 compilation models.

Chapter 5

Analysis of Fortran90 Array Operations

5.1 Introduction

Prior to optimizing a program, a compiler must analyze it to determine facts that can be used to validate transformations. Since our goal is the optimization of programs at the Fortran90 array level we need the ability to analyze programs at that level. In this chapter we present methods for extending scalar analysis algorithms to include the analysis of Fortran90 array operations. The analysis methods we investigate are data-flow analysis and dependence analysis, as well as the static single assignment intermediate representation. The results of these analyses are used by the optimizing transformations presented in the remainder of this dissertation.

5.2 Data-Flow Analysis

Data-flow analysis is a classical analysis technique which tracks the flow of data through the program's variables [11, 100]. Analyzing a program's data flow can provide information on reaching definitions, available expressions, and live variables, which in turn can enable optimizations such as global common subexpression elimination, loop invariant code motion, strength reduction, and global register allocation [5]. Data-flow analysis and associated optimizations have been standard components of optimizing compilers for many years.

In this section we discuss analyzing data-flow through Fortran90 array statements. Since data-flow analysis depends upon a correct control-flow graph (CFG), we first discuss generating CFGs for Fortran90 programs.

5.2.1 Control-Flow Graphs

Generating a CFG for Fortran90 is not much different than for Fortran77. Fortran90 contains only a single control construct that is not present in Fortran77: the CASE construct. The CASE construct has identical semantics to similar constructs found in other languages. Thus the CFG for the CASE construct is generated as with other languages.

Another construct added to the Fortran90 language is the WHERE construct. It is important to note that the WHERE construct is **not** a control construct. It is tempting to think of the WHERE construct as an array-level IF statement, but this is not the case. All the assignment statements in the WHERE construct are executed in a sequential manner, first those under the WHERE statement and then those under the ELSEWHERE statement.

5.2.2 Data-Flow Analysis of Array Expressions

Data-flow analysis is traditionally used to analyze the flow of values through scalar variables. It is too costly to compute data-flow information through each element of an array. However, there are times when we want to know information about an array as a single object. In those cases we can treat the array like a scalar variable, and use existing flow analysis techniques.

Once the CFG for a Fortran90 program is constructed, data-flow analysis proceeds just as with a Fortran77 program. In fact, our Fortran90 compiler uses the data-flow analysis engine that was written for the Fortran77D compiler. The challenge comes in generating the local access sets that are used as input for the analysis.

Local access sets for data-flow analysis, such as *use*, *def*, and *kill*, specify how each basic block affects the flow of values through the program. Extending these to include arrays is a fairly straight-forward exercise. If an array is referenced, regardless of whether it is a reference to a single element or a subsection of the array, it is a *use*. Similarly, any assignment to an array is a *def*. But only a *def* that defines the entire array can be considered a *kill*. This important distinction requires that we examine the subscript expression used for the array access. Subscripts containing only null triplets are easily identifiable as full array references. However symbolic analysis may be required to compare subscript expressions against the declared ranges of the array.

In our compiler, we use data-flow analysis to generate live-variable information for arrays. With this information we create an interference graph, which we use in our offset-array optimization presented in Chapter 6.

5.3 Dependence Analysis of Array Syntax

In this section we present a methodology for performing data dependence analysis directly on Fortran90 array-section references. We show how direction vectors can be extended to include the dependence information. We also introduce a special class of dependences that arise from array syntax, and discuss some of their properties.

The dependence information produced by the algorithms presented here are used in our compiler to perform context partitioning (see Section 7.2) and advanced scalarization (see Chapter 8).

Before beginning it is important to clarify some terminology that is used in this section. An *array reference* is a subscripted variable reference. A *subscript* is one element from a subscript list. A triplet, as defined in Section 2.2, is one type of subscript. It is assumed that whole array references, array references without a subscript list, are represented within the compiler to include a subscript list containing the appropriate number of null triplets.

5.3.1 Data Dependence

The theory of *data dependence* is well understood and is extensively used in advanced optimizing and parallelizing compilers. We say that a data dependence exists between two statements if there is an execution path from one to the other and both statements access the same memory location. Data dependence is fundamental to compilers that attempt reordering transformations since it specifies statement orderings that *must* be preserved to maintain program semantics [12, 161, 166].

There are four types of data dependence:

- *True dependence* (δ) occurs when one statement writes a memory location that another statement later reads.
- *Antidependence* ($\overline{\delta}$) occurs when one statement reads a memory location that another statement later writes.
- *Output dependence* (δ^o) occurs when one statement writes a memory location that another statement later writes.
- *Input dependence* (δ^i) occurs when one statement reads a memory location that another statement later reads. Input dependences are different than the others, in that they do not restrict the order of execution.

Dependence analysis is the process of determining whether a data dependence exists between two statements [24]. The principal focus of dependence analysis is to determine dependences that arise from subscripted array references that appear within loop nests, since it is not always easy to determine if such references access the same memory location.

5.3.2 Partition-based Dependence Testing

In the partition-based dependence testing algorithm [76] used in the analysis and transformation systems at Rice University, pairs of array references are classified before being tested. This enables us to choose the most efficient test for a given pair of references and lets us test the subscripts in the order of less expensive to more expensive. The classification system consists of two orthogonal criteria: *complexity* and *separability*.

Complexity refers to the number of distinct loop induction variables that appear within a subscript. Individual subscripts are first classified, and then those results are used during dependence testing to derive a classification for a subscript pair. Complexity classes include ZIV (zero index variables), SIV (single index variable), and MIV (multiple index variables).

Separability refers to whether or not different subscript positions contain common induction variables. A subscript position is *separable* if the indices it contains do not appear in other subscript positions [12, 41]. If different subscript positions contain the same index, they are said to be *coupled* [121]. The concept of separability is important when testing multidimensional arrays in that it allows dependence testing to proceed subscript-by-subscript without a loss of precision. In contrast, coupled subscripts must be tested as a group to obtain exact results.

The concepts of complexity and separability are combined in the partition-based dependence testing scheme to determine the most appropriate test to use for a given pair of references. An outline of the algorithm is given in Figure 5.1. This algorithm has been used with great success in the PFC compiler [13], the ParaScope programming environment [104], and the Fortran D compiler [93, 152].

5.3.3 Dependence Representation

Data dependences are often represented using *direction vectors* and/or *distance vectors* [161]. The direction vector is an ordering vector, containing $<$, $=$, $>$, or $*$, that

1. Partition the subscripts into separable and minimal coupled groups.
2. Label each subscript pair as ZIV, SIV, or MIV.
3. For each separable subscript pair, apply the appropriate single subscript test based upon the complexity of the subscripts.
4. For each coupled group, apply a multiple script test.
5. If any test yields independence, no dependences exist.
6. Otherwise merge all the direction vectors computed by the previous steps into a single set of direction vectors for the two references.

Figure 5.1 Partition-based dependence testing algorithm.

specifies the relation of the source and target iterations involved in the dependence. The distance vector contains the vector difference of the source and target iterations. These vectors are convenient methods for characterizing the relationship between the values of the loop indices of the two array references involved in the dependence. In this work we discuss only direction vectors, although the algorithms presented could easily be adapted to work with distance vectors.

Direction vectors are useful in determining if a dependence is *loop-carried* or *loop-independent* [13]. For loop-carried dependences, the direction vector also tells us which loop *carries* the dependence and in which direction. The vectors contain an element for each loop which encloses both statements involved in the dependence. The positions in the vectors from left to right correspond to the surrounding loop indices from outermost to innermost.

To extend direction vectors for array-section references, we add vector elements to account for the implied loops of the triplets. The number of elements added to a vector corresponds to the number of triplets that the two array references have in common. In most cases these vector elements are only considered when the two array references are congruent[2], in which case they have the same number of triplets. These new direction vector elements appear to the right of those elements corresponding to surrounding loops. We order the elements from left to right as they appear in the subscript list, although any consistent ordering will do. In fact some people may want

[2]See Section 4.8.1 for the definition of congruent arrays.

to use the opposite ordering since they want the rightmost direction vector position, corresponding to the innermost loop, to be associated with the leftmost subscript due to the column-major storage layout of Fortran arrays. We chose the left to right ordering for its ease of understanding since it matches the order in which the triplets appear in the program text.

Consider the code fragment shown in Figure 5.2. Any dependences among statements S_1 and S_2 due to the references to array A would have an associated direction vector containing three elements: the first corresponding to the I loop, the second corresponding to the first triplet, and the third corresponding to the second triplet. This fragment of code contains the following dependences: $S_1\overline{\delta}_{(=,>,=)}S_1$, $S_1\overline{\delta}_{(=,=,=)}S_1$, $S_1\delta_{(=,>,=)}S_2$, and $S_2\overline{\delta}_{(<,=,=)}S_1$.

5.3.4 Scalarization Dependences

Given this extension to the concept of a direction vector, there is a subclass of dependences that deserve some special attention: those dependences which have an "=" in all non-triplet direction vector positions. We call these dependences *scalarization dependences* [138]. Since scalarization dependences arise from parallel constructs in the Fortran90 program, they do not have the same behavior as non-parallel dependences. Note that it is valid for *any* of the three direction specifiers to appear in the triplet-related vector positions. Thus for scalarization dependences, it is no longer the case that a true dependence with a ">" as the first non-"=" direction is equivalent to an antidependence with the direction reversed, as has been previously noted by others [14, 40].

By definition, scalarization dependences are loop-independent with regard to surrounding loops. This has several implications. First, any such dependence of a statement on itself is always an antidependence (ignoring input dependences), whereas such a dependence from one statement to a subsequent one is either a true or output dependence. Next, scalarization dependences have no effect on the parallelization of

```
      DO I = 1, N-1
S1:     A(I,2:N-1,1:N) = A(I,1:N-2,1:N) + A(I,2:N-1,1:N)
S2:     B(I,2:N-1,1:N) = A(I,3:N,1:N) + A(I+1,2:N-1,1:N)
      END DO
```

Figure 5.2 Fortran 90 code fragment.

surrounding loops, regardless of what direction the triplet-related positions contain. Finally, it is especially important to point out that such dependences do not affect the ability to parallelize the DO-loops that get generated during the scalarization of the Fortran90 code. This is due to the fact that the array-section subscripts are explicitly parallel constructs.

But this does not mean that we can ignore scalarization dependences. These dependences play an important role when the compiler scalarizes the Fortran90 program into its Fortran77 equivalent. This aspect of the dependences is addressed in more detail in Chapter 8.

Consider again the code in Figure 5.2. This fragment of code contains three scalarization dependences: $S_1\overline{\delta}_{(=,>,=)}S_1$, $S_1\overline{\delta}_{(=,=,=)}S_1$, and $S_1\delta_{(=,>,=)}S_2$. The code also has the dependence $S_2\overline{\delta}_{(<,=,=)}S_1$ which is carried by the I loop.

5.3.5 Classification of Array-Section References

As introduced in Section 5.3.2, pairs of array references are classified before being tested by the partition-based dependence testing algorithm. This allows us to choose the most efficient test for a given pair of references. We now extend the concepts of complexity and separability to include array-section references.

Complexity

We have created a new complexity class for subscripts containing array syntax. We call this new class simply TRIPLET, corresponding to the triplet notation used in the subscript. Unlike the other complexity classes, a TRIPLET is also sub-classified to indicate the complexity of its components. A TRIPLET is sub-classified as SIV if the corresponding triplet subscript contains no index variables (the SIV sub-classification is due to the index variable implicit in the triplet notation). If the triplet contains one or more index variables from enclosing loops in any of its components, then the TRIPLET is sub-classified as MIV. When convenient, we use the shorthand TRIPLET$_{SIV}$ and TRIPLET$_{MIV}$ to represent the complexity and sub-classification of a triplet subscript.

Statement S_3 in Figure 5.3 has two array references, each containing two subscripts that are classified as TRIPLET. For each reference, the first triplet subscript is sub-classified as MIV due to the induction variable J, and the second triplet subscript is sub-classified as SIV.

```
      DO J = 1, N
S3:     A(J:N,K,1:N) = A(J:N,1:N,L) + ···
      END DO
```

Figure 5.3 Complexity and separability example.

Separability

The concept of separability is an important issue for a dependence testing algorithm that is interested in both precision and efficiency. It allows dependence testing to proceed subscript-by-subscript, thus breaking down the problem space into smaller pieces, without a loss of precision. Luckily, array-syntax subscripts can cause different subscript positions to become coupled in only one situation: if corresponding triplets for the two array references being tested are in different subscript positions, those positions must be coupled.

Considering again the two references to array A in Figure 5.3, we see that the second and third subscript positions become coupled since the second triplet appears in each of them. The first subscript position is separable. Note however, that if the induction variable J appeared in another subscript position, that position would be coupled with the position that contains the first triplet. That coupling would be due to J though and has nothing to do with the triplet.

5.3.6 Dependence Testing of Array Expressions

We have designed our dependence testing methodology to work with the partition-based testing scheme presented in Figure 5.1. We extend the algorithm to first determine the separability and complexity for triplet subscripts. In the following subsections we introduce the necessary decision algorithms that determine independence of array-section references, or determine dependence and produce the desired direction vectors. As with most dependence testing schemes, we assume that all expressions used in the subscripts and triplets are linear in the loop induction variables. If non-linear expressions are encountered we assume all direction vectors are possible.

In some situations that may occur, we exploit existing SIV and MIV tests. This necessitates the generation of the appropriate input parameters. To accomplish this we translate the triplet notation into a linear function of a pseudo-induction variable i. The pseudo-induction variable runs from 1 to $(ub - lb + st)/st$, where lb, ub, and

st are the lower bound, upper bound, and stride of the triplet, respectively. The linear function that is used in place of the triplet within the dependence tests is $st * i + (lb - st)$. When the lower bound and stride are both one, this simplifies to i. This translation does not need to be applied to the program representation; it is only needed to produce the necessary input parameters for existing tests when required.

Separable Triplet Subscript Tests

The dependence test used for a separable subscript pair in which one of the subscripts is a triplet depends upon the triplet's sub-classification as well as the complexity of the other subscript. When both subscripts in the pair to be tested are classified as TRIPLET_{SIV}, an SIV test is required. In this case we exploit the existing SIV test implemented in the system. To use the test we produce the appropriate input parameters by generating the linear function of a pseudo-induction variable, as explained previously.

However, there are two common cases which can be tested quite easily without requiring the conversion. The first case is when both triplets have a stride of one. In that situation the dependence distance is simply the difference of the lower bounds:

$$d = lb_1 - lb_2 \tag{5.1}$$

A dependence exists if and only if $|d| \leq ub_1 - lb_1$. The second case is when both triplets have the same non-unit stride, in which the dependence distance is:

$$d = \frac{lb_1 - lb_2}{st_1} \tag{5.2}$$

In this case a dependence exists if and only if d is an integer and $|d| < (ub_1 - lb_1 + st_1)/st_1$. Most triplet subscripts are expected to fall into one of these two special cases, and as can be see in Equations 5.1 and 5.2, the dependence tests for these cases are simple and easy to compute.

If the case where the triplet is classified as TRIPLET_{SIV} and the other subscript is ZIV, the pair has the form $\langle lb\!:\!ub\!:\!st, c_1\rangle$. For this situation, we define the *dependence point* to be:

$$p = \frac{c_1 - lb + st}{st} \tag{5.3}$$

A dependence exists if and only if p is an integer and is in the range $1\!:\!(ub - lb + st)/st$. For the common case where both lb and st are equal to one, p is simply equal to c_1.

All dependence directions are possible, except for when p equals lb or ub, in which case one direction can be eliminated.

In all other cases of separable subscript pairs that contain at least one triplet an MIV test is required. We rely upon the existing MIV tests implemented in the system in the same manner as we used the SIV test above, by converting the triplet into a linear function of a pseudo-induction variable.

Coupled Triplet Subscript Test

When triplet subscripts are coupled with other subscripts a multi-subscript test is necessary. We utilize the existing scalar testing algorithms available in the system by converting the triplets into linear functions of pseudo-induction variables. However, this conversion has a special consideration in the case of coupled subscripts. If the subscripts became coupled because corresponding triplets did not appear in matching subscript positions, then the linear functions generated for the corresponding triplets share the same pseudo-induction variable. Once the triplets have been translated, we exploit whichever multi-subscript test is available in our system [121, 134, 164].

5.4 Static Single Assignment Form

In recent years, Static Single Assignment (SSA) form [64, 65] and related intermediate representations have gained in popularity because of their efficiency in program analysis and transformations [15, 137, 155]. SSA is loosely characterized by the trait that each variable has only a single definition. This is achieved by creating a new instance of a variable, typically indicated by a subscript, each time the variable is assigned a new value. For example, the variable V will have a different instantiation, such as V_0, V_1, and V_2, at different definition points. When different instances of a variable come together at a control flow merge point, the instances are combined via a *ϕ-function* which defines yet another instance of the variable. See Figure 5.4 for a sample code segment and its equivalent SSA form including the use of a ϕ-function.

Often the SSA representation of a program is in the form of a sparse graph, in which nodes represent definition points of variables and edges connect a variable to all its uses. For the implementation of SSA that exists within the Fortran D compiler [87], the SSA graph of a program does not exist as a separate object but rather is built on top of the CFG. It is important to realize that we keep a close correspondence between the SSA graph and the CFG. This gives us the capability to use the strengths of each,

```
IF (A == 0) THEN
  B = 0
ELSE
  B = C/A
END IF

D = 2*B
```

(a) Original source.

```
IF (A0 == 0) THEN
  B0 == 0
ELSE
  B1 = C0/A0
END IF
B2 = φ(B0,B1)
D0 = 2*B2
```

(b) SSA version.

Figure 5.4 Example of SSA form.

and allows us to efficiently move between the two as needed. It is also worthwhile to mention for the benefit of those who use SSA for compiling serial programs that we do not delete/subsume copy operations that involve arrays[3].

There are several variants of SSA available. The variations can enhance the precision of different analysis algorithms. For our purposes, we have found that the pruned-SSA form [60] best fits our needs. In the pruned-SSA form, dead ϕ-functions have been deleted. We also employ both def-use and use-def edges in our graph. The only other change we make to the SSA representation deals with the handling of arrays.

5.4.1 SSA and Fortran90 Array Sections

Within the original design of SSA, uses and modifications of arrays are represented with ACCESS and UPDATE operators, respectively. To enable the analysis of Fortran90 programs, we have enhanced these operators to handle array sections by incorporating regular section descriptors (RSDs) [12]. For instance, the UPDATE operator takes three arguments: the array being modified, a section descriptor indicating the affected elements, and the new values to be stored in those locations. Figure 5.5 contains some examples of array section references and their corresponding SSA form.

The only other modification that we were required to make to the Fortran77D implementation of SSA so that it would properly handle Fortran90D programs was the identification of array kills. This was accomplished by a quick analysis of the array-section subscripts, similar to that performed for data-flow analysis as described

[3]We make this point since it has caused confusion when we've discussed the ideas presented in this dissertation with our comrades in the scalar compiler group.

```
··· = A(1:N)              ··· = ACCESS(A5,[1:N])
A(2:N-1) = ···            A6  = UPDATE(A5,[2:N-1],···)
··· = A(J)                ··· = ACCESS(A6,[J])
```

(a) Original source. (b) SSA version.

Figure 5.5 Example of SSA form of array-section references.

in Section 5.2.2. The identification of array kills is necessary to prevent the generation of useless def-use edges coming into the first argument of the UPDATE operator. Preventing such useless edges from being inserted into the SSA graph means that ϕ-functions that are created for whole-array definitions that appear inside a loop can be pruned away as useless if they were not referenced after the loop. This is very important since Fortran90 compilers often generate many array temporaries, and the interaction between the UPDATE operators and the ϕ-functions give the appearance that the temporaries have much larger live-ranges than they actually have.

Figure 5.6 contains an example of how this occurs. In the original source, the array TMP has a very short live range: it is defined in one statement and used in the next. However, the live ranges of the TMP_i arrays in the SSA version of the code span the entire DO-loop. This is because the UPDATE operator makes TMP_2 appear live and the ϕ-function defining TMP_2 causes TMP_1 to become live around the back-edge of the loop. Preventing the generation of the def-use edge from the ϕ-function to the UPDATE operator means that the ϕ-function is useless and will be pruned away. This in turn reduces the live-range of TMP_1 to be the same as TMP in the original program.

In our Fortran90D compiler we use SSA form, enhanced as described above, to perform the analysis required for our offset array optimization. The use of SSA in this instance permits advanced analysis and transformations to be performed in linear time. In addition, no possible applications of the optimization are missed. Full details of the offset array optimization and its use of SSA form are given in Chapter 6.

5.5 Summary

In this chapter we have presented methodologies for extending program analysis to handle Fortran90 array references. We have given algorithms to extend data flow analysis and SSA form as well as extending dependence testing to directly analyze

```
DO I=1,N

  ...
  TMP(:)  = ···
  ··· = TMP(:)
  ...
END DO
!  no other uses of TMP
```

(a) Original source.

```
DO I=1,N
  TMP₂ = φ(TMP₀,TMP₁)
  ...
  TMP₁ = UPDATE(TMP₂,[:],···)
  ··· = ACCESS(TMP₁,[:])
  ...
END DO
!  no other uses of TMPᵢ
```

(b) SSA version.

Figure 5.6 Interaction between ϕ-function and UPDATE operator.

data dependences arising from array-section references. An important characteristic of the testing procedures presented in this chapter is that they were designed to fit smoothly into the framework of existing optimizing compilers. These extensions give Fortran90 and HPF compilers analysis capabilities not previously available, allowing them to make decisions and perform transformations at the Fortran90 level before scalarizing the program into Fortran77 code.

Chapter 6

Offset Array Optimization

Eliminating the local data motion by separating the set of data that must move between nodes from the data that stays within local memory may yield a significant performance improvement. S. L. Johnsson [98]

6.1 Introduction

For Fortran90D or HPF to gain acceptance as a vehicle for parallel scientific programming, they must achieve high performance on problems for which they are well suited. To achieve high performance on a distributed-memory parallel machine, a Fortran90D compiler must do a superb job of translating Fortran90 data-parallel array operations into an efficient sequence of operations that minimize the overhead associated with data movement.

Interprocessor data movement on a distributed-memory parallel machine is typically far more costly than movement within the memory of a single processor. For this reason, much of the prior research on minimizing data movement has focused on the interprocessor case. Notable research in this area includes Knobe, Lucas and Steele's work on automatically aligning and partitioning arrays to minimize data movement associated with binary operations on arrays [109], and Chatterjee, Gilbert, Schreiber, and Teng's work on optimally mapping the computation of array expressions to processors to minimize data movement when the mappings of data to processors are fixed [55].

While interprocessor data movement is more costly per element, the number of elements moved within the memory of a single processor may be much larger, causing the cost of local data movement to be dominant. For example, consider applying the Fortran90 CSHIFT intrinsic to perform a circular left shift by one position of a 1D array of 1000 elements partitioned block-wise across four processors (each processor is responsible for a contiguous block of 250 elements). To perform the shift, each processor sends one element to its left neighbor in a ring, moves 249 elements one position left within the processor's local memory, and receives one element previously sent by the neighboring processor to the right. Here, each processor moves 249 elements on-processor for every one element moved off-processor. The cost of local data movement becomes more important for distributed arrays as the partition size per processor increases.

In this chapter, we focus on the problem of minimizing the amount of intraprocessor data movement when computing Fortran90 array operations. To make our technique as generally applicable as possible, we handle array assignment statements where the right-hand side consists of a call to a Fortran90 shift intrinsic. The technique can handle all such assignment statements, even when the definition of the array and its uses are separated by control flow. Such a technique supersedes those that are restricted to a single statement.

In the next section we briefly review the Fortran90 shift operators and their execution cost on distributed-memory machines. In Section 6.3, we describe the *offset array* strategy for reducing intraprocessor data movement associated with shift operations on arrays with BLOCK or BLOCK_CYCLIC distributions. We have developed two analysis algorithms for determining when offset arrays may be used. Section 6.4 presents an algorithm that performs analysis only at the basic block level. Section 6.5 describes a global SSA-based analysis algorithm that can exploit offset arrays over entire procedures. We close the chapter by presenting some empirical results and discussing related works.

6.2 Fortran90 Shift Operations

The Fortran90 circular shift operator `CSHIFT(ARRAY, SHIFT, DIM)` returns an array of the same shape, type, and values as ARRAY, except that each rank-one section of ARRAY crossing dimension DIM has been shifted circularly SHIFT times. The sign of SHIFT determines the shift direction. See Figure 6.1 for an example. The end-off shift operator `EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)` is identical to CSHIFT except for the handling of boundaries: array elements shifted off the end are lost and `BOUNDARY` values are inserted at the other end. Figure 6.2 has an example of an EOSHIFT. For the rest of the chapter we focus on optimizing CSHIFT operations although our techniques can be generalized to handle EOSHIFT as well.

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \qquad \begin{bmatrix} 7 & 1 & 4 \\ 8 & 2 & 5 \\ 9 & 3 & 6 \end{bmatrix}$$

(a) `SRC` (b) `CSHIFT(SRC,SHIFT=-1,DIM=2)`

Figure 6.1 Example of the CSHIFT intrinsic.

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 4 \\ 0 & 2 & 5 \\ 0 & 3 & 6 \end{bmatrix}$$

(a) `SRC` (b) `EOSHIFT(SRC,-1,0,2)`

Figure 6.2 Example of the EOSHIFT intrinsic.

6.2.1 Sources of Shift Operations

For Fortran90D, optimizing CSHIFT operations is important since CSHIFT operations are ubiquitous in stencil-based dense array computations for which Fortran90D is best suited. Besides CSHIFT operations written by users, compilers for distributed-memory machines commonly insert them to perform data movement needed for operations on array sections that have different processor mappings [110, 139, 34]. Section 4.5 describes how the communication generation phase of the compiler transforms certain array section references into calls to shift intrinsics that perform the necessary interprocessor data movement. For example, given the statement

```
X(2:255) = X(1:254) + X(2:255) + X(3:256)
```

the CM Fortran compiler would translate it into the following statement sequence, where `TMP1` and `TMP2` are compiler temporary arrays that match the size and distribution of `X`:

```
ALLOCATE TMP1, TMP2
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
X(2:255) = TMP1(2:255) + X(2:255) + TMP2(2:255)
DEALLOCATE TMP1, TMP2
```

For the rest of the chapter, we assume that all CSHIFT operations in a program are explicit (either user-written, or inserted where appropriate during an earlier phase of compilation), that each call to CSHIFT occurs as a singleton operation on the right-hand side of an assignment statement, and that each CSHIFT in our intermediate form is applied only to a whole array. All other occurrences of CSHIFT can be translated into the required form by factoring expressions and introducing array temporaries. For example, the statement:

```
X = X + CSHIFT(X,SHIFT=-1,DIM=1) + CSHIFT(Y(1:N:2),SHIFT=+1,DIM=1)
```

would be represented, internal to the compiler, by the following set of statements:

```
ALLOCATE TMP1, TMP2, TMP3
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = PACK(Y,MASK=(/ (MOD(I,2).EQ.1),I=1,N /))
TMP3 = CSHIFT(TMP2,SHIFT=+1,DIM=1)
X = X + TMP1 + TMP3
DEALLOCATE TMP1, TMP2, TMP3
```

Temporary arrays TMP1–TMP3 are specified such that they match the size and distribution of X.

6.2.2 Shift Operations on Distributed-Memory Machines

When a distributed array is shifted across a distributed dimension, two major actions take place:

1. Data elements that must be shifted across processing element (*PE*) boundaries are sent to the neighboring PE. This is the interprocessor component of the shift. The dashed lines in Figure 6.3 represent this data movement for arrays distributed in a BLOCK fashion.

2. Data elements that stay within the memory of the PE must be copied to the appropriate locations in the destination array. This is the intraprocessor component of the shift. The solid lines in Figure 6.3 represent this data movement.

Assuming a BLOCK distribution and that each PE contains a 2D subgrid of size $(g \times g)$, a shift amount of d, $d < g$, consists of an interprocessor move of d columns (of size g), and an intraprocessor move of $g - d$ columns. The cost of such a shift operation is described by the following model [69]:

$$T_{\text{shift}} = g\ (g - d)\ t_{\text{onpe}} + C_{\text{onpe}} + g\ d\ t_{\text{offpe}} + C_{\text{offpe}} \tag{6.1}$$

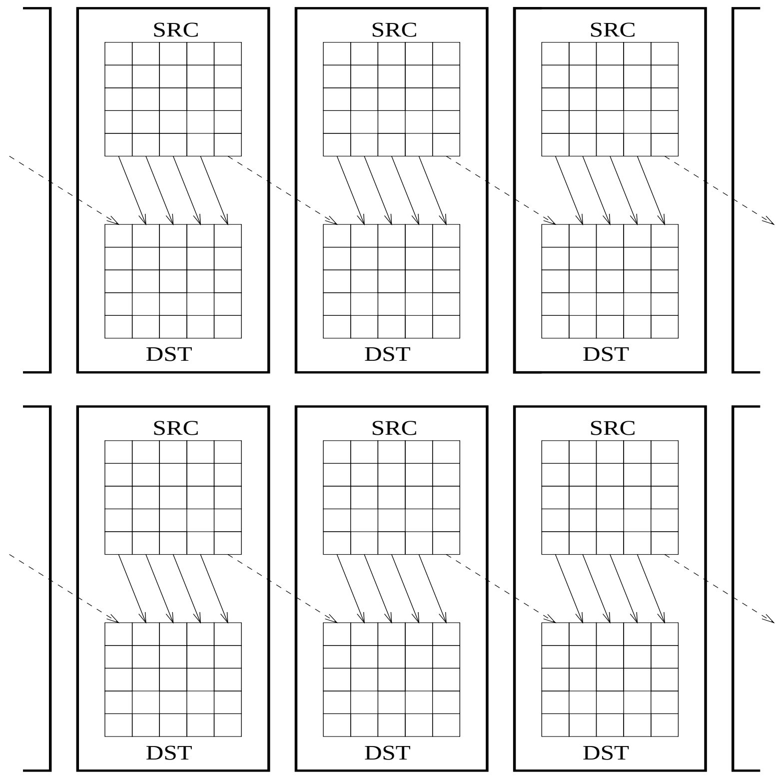


Figure 6.3 `DST = CSHIFT(SRC,SHIFT=-1,DIM=2)`

where t_{onpe} and t_{offpe} represent the time to perform an intraprocessor and interprocessor copy respectively, and C_{onpe} and C_{offpe} represent the startup time (or latency) for each type of copy. Table 6.1[4] presents measured values for each of the model's parameters for four different SIMD machines. Different models are required for the cases $d = g$ and $d > g$. BLOCK_CYCLIC distributions also require a different model which include a parameterization for the blocking factor.

The instances in which we are most interested occur when d is small compared to g. For such cases Equation (6.1) is $O(g^2\ t_{\mathrm{onpe}})$ and the execution time T_{shift} is dominated by the cost of the intraprocessor copies, even when $t_{\mathrm{onpe}} \ll t_{\mathrm{offpe}}$.

There are also additional costs that should be considered that are not associated directly to the execution of the shift operations. These costs relate to the temporary arrays that are often created to receive the results of the shifts. Not only must time be spent allocating and deallocating these arrays, but their additional memory

[4] From Fatoohi [69], ©1993 ACM.

Parameter	CM2	MPP	MP-1	610C
t_{offpe}	9.0	3.2	2.7	3.2
C_{offpe}	20.0	13.4	41.9	7.2
t_{onpe}	0.7	-	5.6	9.0
C_{onpe}	35.0	-	59.1	18.0

Table 6.1 Measured cost of communication parameters for a 32-bit word (in μsec).

requirements often limit the size of the problems which may be solved on a given machine.

6.3 Offset Arrays

The goal of the work described here is to eliminate the intraprocessor copying associated with a Fortran90 CSHIFT operation when it is safe to do so. When we can determine that the intraprocessor copying of a CSHIFT is unnecessary, we can transform the program to perform only the interprocessor copying and rewrite references to the shift's destination array to refer to the source array with indexing adjusted by the shift amount. We call such a destination array an *offset array*. In the following subsections, we present criteria for determining when offset arrays are safe and profitable and present the code transformations that avoid intraprocessor copying by exploiting offset arrays.

To hold the data that must move between PEs, we use *overlap areas* [73]. Overlap areas are subgrid extensions to hold data received from neighboring PEs. The advantage of this is that it allows us to access the data of the offset array, both the portion that is shared with the source array and the portion received from the neighboring PE, in a straight-forward and simple manner. The disadvantage of this method is that the storage allocated to the overlap areas is permanent. To limit the impact of allocating this permanent storage, we place an upper bound on its size. This also limits the size of the SHIFT amounts that may be optimized. If the shift does not fit into the overlap areas, then it cannot be an offset array. But since we are only interested in optimizing small SHIFT amounts, this is not a great concern. This upper bound should be set at compile-time by a heuristic that considers the machine characteristics along with the expected size of the subgrids.

6.3.1 Criteria for Offset Arrays

We have established a set of criteria to determine when it is safe and profitable to create an offset array. Given an assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` within our intermediate representation, the array DST may be treated as an offset array, and can thus share storage with array SRC, if the following criteria can be verified for this statement at compile time:

1. The source array SRC is not modified while this definition of DST is *live*.

2. The destination array DST is not partially modified[5] while SRC is *live*.

From the work on copy elimination in functional and higher-order programming languages [144], we know that the above two criteria are necessary and sufficient conditions for when the two objects can share the same storage. However, the sharing of storage may not always be profitable. To insure profitability, we add the following efficiency criteria:

3. The SRC array and the DST array are distributed in the same BLOCK (or BLOCK_CYCLIC) fashion and are aligned with one another.

4. The values SHIFT and DIM are compile-time scalar constants.

5. The amount of interprocessor data must fit within the bounds placed on the size of the overlap areas.

6. For each use of DST that is reached by the given definition, all the definitions of DST that reach that use are identical offset arrays of the same source array SRC.

These efficiency criteria may be relaxed if we are willing to generate multiple versions of code for statements that use the array DST, and then select the appropriate version depending upon run-time conditions. However, due to the drawbacks of multiple versions of code, in particular code growth, we consider these additional criteria to be important.

[5]Any partial modification requires a copy of the shifted array SRC and so we simply go ahead and make the copy at the point of the shift. Any full modification of DST which *kills* the whole array does not require the copy of SRC and thus DST may still be treated as an offset array up to the point of the killing definition.

6.3.2 Offset Array Optimization

Once we have determined that the destination array of the assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` may be an offset array, we perform the following transformations on the code. These transformations take advantage of the data that may be shared between the source array SRC and destination array DST and move only the required data between the PEs.

1. First we replace the shift operation `DST = SHIFT(SRC,SHIFT,DIM)` with a call to a routine that moves the interprocessor data into the appropriate overlap area: `CALL OVERLAP_SHIFT(SRC,SHIFT,DIM)`.

2. We then replace all uses of the array DST, that are reached from this definition, with a use of the array SRC. The newly created references to SRC carry along special annotations representing the values of SHIFT and DIM. In the examples that follow, the annotations are represented by a superscripted vector where the DIM-th element contains the value SHIFT; *e.g.*, $\mathrm{SRC}^{<\ldots,\mathrm{SHIFT},\ldots>}$.

3. Finally, when creating subgrid loops during the code generation phase, we alter the subscript indices used for the offset arrays. The array subscript used for the offset reference to SRC is identical to the subscript that would have been generated for DST with the exception that the DIM-th dimension has been incremented by the SHIFT amount.

It is possible, and actually expected, that offset arrays are themselves used in other shift operations. If these shift operations also meet all of the criteria to be an offset array then the above transformations can again be applied. We call such arrays *multiple-offset arrays*. If one dimension is shifted multiple times, the SHIFT amounts are simply added together. Figure 6.4 gives an example of a multiple-offset array.

As an example, consider the 5-point stencil routine in Figure 6.5(a). The expected intermediate representation in Figure 6.5(b) is achieved by separating the communication operations from the computational operations. Once we have determined that

```
t1 = cshift(a,shift=-1,dim=1)          CALL overlap_cshift(a,shift=-1,dim=1)
t2 = cshift(t1,shift=+1,dim=2)   ==>   CALL overlap_cshift(a<-1,0>,shift=+1,dim=2)
b = a + t2                             b = a + a<-1,+1>
```

Figure 6.4 A multiple-offset array.

```
 SUBROUTINE FIVE_PT(a,b,n)
 REAL, ARRAY(n,n) ::  a,b
 DECOMPOSITION decomp(n,n)
 ALIGN a,b with decomp
 DISTRIBUTE decomp(BLOCK, BLOCK)
 REAL cc,cn,ce,cw,cs
 COMMON cc,cn,ce,cw,cs

 b = cc * a
&  + cn * cshift(a,-1,1)
&  + cs * cshift(a,+1,1)
&  + cw * cshift(a,-1,2)
&  + ce * cshift(a,+1,2)

 RETURN
 END
```

(a) Original program

```
 SUBROUTINE FIVE_PT(a,b,n)
 REAL, ARRAY(n,n) ::  a,b
 REAL, ALLOCATABLE ::  t1,t2,t3,t4
 DIMENSION(:,:)  ::  t1,t2,t3,t4
 DECOMPOSITION decomp(n,n)
 ALIGN a,b,t1,t2,t3,t4 with decomp
 DISTRIBUTE decomp(BLOCK, BLOCK)
 REAL cc,cn,ce,cw,cs
 COMMON cc,cn,ce,cw,cs

 ALLOCATE(t1(n,n),t2(n,n),
&        t3(n,n),t4(n,n))
 t1 = cshift(a,-1,1)
 t2 = cshift(a,+1,1)
 t3 = cshift(a,-1,2)
 t4 = cshift(a,+1,2)
 b = cc * a
&  + cn * t1
&  + cs * t2
&  + cw * t3
&  + ce * t4
 DEALLOCATE(t1,t2,t3,t4)

 RETURN
 END
```

(b) Intermediate representation

```
 SUBROUTINE FIVE_PT(a,b,n)
 REAL, ARRAY(n,n) ::  a,b
 DECOMPOSITION decomp(n,n)
 ALIGN a,b with decomp
 DISTRIBUTE decomp(BLOCK, BLOCK)
 REAL cc,cn,ce,cw,cs
 COMMON cc,cn,ce,cw,cs

 CALL overlap_cshift(a,-1,1)
 CALL overlap_cshift(a,+1,1)
 CALL overlap_cshift(a,-1,2)
 CALL overlap_cshift(a,+1,2)
 b = cc * a
&  + cn * a<-1,0>
&  + cs * a<+1,0>
&  + cw * a<0,-1>
&  + ce * a<0,+1>

 RETURN
 END
```

(c) Offset array transformations

```
 SUBROUTINE FIVE_PT(a,b,n)
 REAL, ARRAY(n/p,n/p) ::  b
 REAL, ARRAY(0:n/p+1,0:n/p+1) ::  a
 REAL cc,cn,ce,cw,cs
 COMMON cc,cn,ce,cw,cs

 CALL overlap_cshift(a,-1,1)
 CALL overlap_cshift(a,+1,1)
 CALL overlap_cshift(a,-1,2)
 CALL overlap_cshift(a,+1,2)
 do j=1,n/p
  do i=1,n/p
   b(i,j) = cc * a(i,j)
&         + cn * a(i-1,j)
&         + cs * a(i+1,j)
&         + cw * a(i,j-1)
&         + ce * a(i,j+1)
  enddo
 enddo

 RETURN
 END
```

(d) Final node program

Figure 6.5 Offset array optimization on a 5-point stencil computation.

the temporary arrays T1–T4 can be offset arrays, we perform the above set of transformations. Figure 6.5(c) shows the program after the first two transformation steps have been completed. The third step is performed during subgrid loop generation and is shown in Figure 6.5(d). Notice that the declarations for array A and B have been changed to declare the local subgrid, and that the declaration for A includes the necessary overlap area. The node program can now be further enhanced by applying additional advanced optimizations as discussed in Chapter 9.

6.4 Local Offset Array Analysis

The last three criteria for determining if a shifted array may be an offset array can be very difficult to verify in the presence of complex control flow. For that reason we have an analysis algorithm that works on a basic block at a time. After describing the algorithm we analyze its complexity.

6.4.1 Local Analysis

Our local analysis algorithm requires that *live* information is available for SRC and DST arrays of the candidate shift operations. Such information tells us if these arrays are either *live* or *dead* at the end of each basic block.

Our local analysis algorithm is displayed in Figure 6.6. We begin by analyzing each basic block that contains a shift operation. The shift operations within a basic block are processed one at a time in the order that they occur. Processing the shift operations in execution order is important for the proper handling of multiply-offset arrays. For each shift operation encountered, we initially check criteria 3, 4, and 5 specified in Section 6.3.1. If they are satisfied we then check to see if the target array DST is *live* at the end of the basic block. If it is *live* then the local analysis routine cannot determine the safety of the offset array optimization without additional global information, and so the array is not offsetable within the scope of local analysis. However, if DST is *dead*, we scan forward from the shift operation checking the remaining criteria.

Since we can now limit the scope of the analysis to a single basic block, we can easily verify criteria 1 and 2 by scanning the rest of the block after the shift operation. Recall that the shift operation is a killing definition of the array DST, thus we need only look for definitions of DST or SRC that occur between the shift operation and the last reachable use of DST. If a killing definition of DST is encountered we can stop

```
Procedure Local_Offset_Arrays
Input: Set of basic blocks containing shift operations.
Output: Basic blocks optimized with offset arrays.

for each basic block bb in the input set do
   for each SHIFT operation stmt found in execution order of bb do
      dst = stmt.destination_array
      src = stmt.source_array

      if criteria 3, 4, or 5 is violated then iterate endif

      /* Scan rest of basic block bb checking criteria 1, 2, and 6. */
      current_statement = stmt.bblock_next
      offsetable = true
      list_of_references = nil
      src_modified = false
      while current_statement ≠ nil and offsetable do
         if current_statement references dst and src_modified then
            offsetable = false
            break
         elseif current_statement references dst and ¬ src_modified then
            add current_statement to list_of_references
         endif
         if current_statement defines dst then
            if current_statement is a killing definition of dst then
               break      /* No more reachable uses of dst. */
            elseif Is_Array_Dead_Here(bb, current_statement, src) then
               add current_statement to list_of_references
            else
               offsetable = false
            endif
         endif
         if current_statement defines src then src_modified = true endif
         current_statement = current_statement.bblock_next
      endwhile

      if offsetable then
         /* Apply the offset array transformations to statement stmt and */
         /* the statements in list_of_references that reference dst. */
      endif
   endfor
endfor
return

Boolean Function Is_Array_Dead_Here(bb,current_statement,src)
/* Determine if array src is dead at the point of current_statement */
/* in the basic block bb. This just requires a simple backwards scan from */
/* the end of the basic block to statement current_statement, starting with */
/* the pre-computed global live information. */
```

Figure 6.6 Local offset array optimization algorithm our search for any more reachable uses. If a partial modification of DST is found we must determine the liveness of SRC. Since we already have *live* information available at the basic block level, this only requires a backward scan starting from the bottom of the block. We annotate the block so that this scan need only be performed once per SRC array. If SRC is *dead* at the point of the partial modification then it is still possible for SRC and DST to share common storage elements, otherwise DST is not offsetable. If we find that SRC is modified during our scan, then DST is not offsetable if we find a subsequent reachable use of DST. If we complete our scan of the basic block and both criteria 1 and 2 were satisfied, then criterion 6 is satisfied implicitly for all reachable uses of DST that were encountered.

If all of the criteria are satisfied then the array is offsetable and we apply the transformations described in Section 6.3.2. We then proceed with the next shift operation in the current basic block, if one exists; otherwise we process the next basic block containing a shift operation. The algorithm is displayed in detail in Figure 6.6.

6.4.2 Cost Analysis

The local analysis algorithm simply scans each basic block that contains a shift operation. The basic block is scanned at most twice for each shift operation that it contains: a forward scan to verify the offset criteria, and a possible backward scan to determine the liveness of the SRC array. The complexity is thus $O(bb * s)$, where bb is the maximum size of a basic block, and s is the number of shift operations in the program. If we assume an upper limit on the size of basic blocks, the complexity is linear in the number of shift operations. We note here that the cost to compute *live* information is $O(n \log n)$.

6.5 Global Offset Array Analysis

Determining if it is valid for a shifted array to be an offset array across multiple basic blocks can be very difficult. This is due to the challenge of verifying the criteria specified in Section 6.3.1 over complex control flow patterns. In this section we present an algorithm capable of solving this problem in an efficient manner.

6.5.1 Global Analysis

If one looks closely at the criteria in Section 6.3.1 it can be seen that criterion 6 is the most difficult to verify. This is because of its inherent circularity: an array can be an offset array if it and all other reaching definitions of a use can be offset arrays. To be able to verify such a criterion requires an optimistic analysis framework; *i.e.*, one must assume all shifted arrays are offsetable, and then attempt to disprove the assumption. We have chosen to base our optimistic analysis framework on the SSA intermediate representation [65] extended as discussed in Section 5.4.

In addition to the SSA graph, we generate an interference graph [38]. The interference graph indicates those SSA variables with overlapping live ranges, and is used to check for violations of criteria 1 or 2. The *live*-ness of arrays is computed as described in Section 5.2. The graph is built in the usual manner, but with one exception: all ϕ-functions occurring at the same merge point are considered to be executed simultaneously. This prevents the detection of spurious interferences that may prevent the use of offset arrays across merge points. We use pruned-SSA to prevent live ranges from being extended unnecessarily, thus also preventing spurious interferences. We also enhance the accuracy of the interference graph by exploiting the RSDs in UPDATE operations to identify statements that *kill* the entire array.

6.5.2 Offset Array Algorithm

In this section, we present our algorithm for identifying offset arrays and transforming the program to reference them. In describing the algorithm, we typically refer to the SSA variable names rather than their CFG counterparts. The algorithm is shown in Figure 6.7.

We begin by traversing the CFG in a reverse depth-first order looking for shift operations. Upon encountering a shift operation which satisfies criteria 3–5, we check the interference graph to see if criterion 1 is not violated. If all the required criteria are satisfied, then it is safe for the destination array to be an offset array. Given such a shift operation, we rename the destination array DST_i by giving it the same SSA name as the source array SRC_j. The new name is annotated with SHIFT and DIM information as described in Section 6.3.2. By using the same name we do not violate the spirit of SSA. This is because the shift really does not create any new values but rather just specifies a new indexing method for existing values. We change the use of the SHIFT intrinsic into a use of the OVERLAP_SHIFT routine, and update the interference graph by renaming the changed variable.

After we make this change, we propagate the information in an optimistic manner. This insures that criterion 6 is satisfied wherever possible. The SSA framework allows

Procedure *Offset_Arrays*
Input: *CFG*, the control flow graph for the procedure.
Output: *CFG* optimized with offset arrays.
/* See Figure 6.8 for auxiliary routines. */

$SSA = Create\_SSA\_Form(CFG)$
$IG = Build\_Interference\_Graph(SSA, CFG)$

for each SHIFT operation *stmt* in a depth-first traversal of the *CFG* **do**
  push *stmt* onto stack S
endfor

while stack S is not empty **do**
  pop *stmt* off of stack S
  switch (*stmt*)

  case SHIFT operation: $Dst_{Dsub} = shift(Src_{Ssub}^{Sanot}, shift, dim)$:
    if criteria 3, 4, or 5 is violated **then break endif**
    if $Check\_Interferences(Dst_{Dsub}, Src_{Ssub}, IG)$ **then break endif**
    calculate new annotation *Nanot* from *Sanot*, *shift*, and *dim*
    replace *stmt* with $Src_{Ssub}^{Nanot} = overlap\_shift(Src_{Ssub}^{Sanot}, shift, dim)$
    call $Replace\_Uses(Dst_{Dsub}, Src_{Ssub}^{Nanot}, S)$
    call $Update\_Graphs(Dst_{Dsub}, Src_{Ssub}^{Nanot}, SSA, IG)$
    break

  case ϕ-function: $Dst_{Dsub} = \phi(Lvar_{Lsub}^{Lanot}, Rvar_{Rsub}^{Ranot})$:
    if criterion 3 is violated **then break endif**
    if $Lvar^{Lanot} \neq Rvar^{Ranot}$ **then break**
    elseif $Dst_{Dsub} = Lvar_{Lsub}$ or $Dst_{Dsub} = Rvar_{Rsub}$ **then break**
    elseif $Lvar_{Lsub}^{Lanot} = Rvar_{Rsub}^{Ranot}$ **then**
      $New_{Nsub} = Lvar_{Lsub}$
    else
      $New_{Nsub} = Find\_Phi(Lvar_{Lsub}, Rvar_{Rsub}, stmt)$
    endif
    if $Check\_Interferences(Dst_{Dsub}, New_{Nsub}, IG)$ **then break endif**
    replace *stmt* with $New_{Nsub}^{Lanot} = \phi(Lvar_{Lsub}^{Lanot}, Rvar_{Rsub}^{Ranot})$
    call $Replace\_Uses(Dst_{Dsub}, New_{Nsub}^{Lanot}, S)$
    call $Update\_Graphs(Dst_{Dsub}, New_{Nsub}^{Lanot}, SSA, IG)$
    break

  case UPDATE operation: $Dst_{Dsub} = Update(Lvar_{Lsub}^{Lanot}, section, values)$:
    if criterion 3 is violated **then break endif**
    $New_{Nsub} = Gen\_Next\_SSA\_Var(Lvar)$
    if $Check\_Interferences(Dst_{Dsub}, New_{Nsub}, IG)$ **then break endif**
    replace *stmt* with $New_{Nsub}^{Lanot} = Update(Lvar_{Lsub}^{Lanot}, section, values)$
    call $Replace\_Uses(Dst_{Dsub}, New_{Nsub}^{Lanot}, S)$
    call $Update\_Graphs(Dst_{Dsub}, New_{Nsub}^{Lanot}, SSA, IG)$
    break

  endswitch
endwhile

call $Insert\_Copies(CFG, SSA)$
return *CFG*

Figure 6.7 Global offset array optimization algorithm

```
Function Check_Interferences(Dst_Dsub, Src_Ssub, IG)
/* Return TRUE if there exists an interference between */
/* Dst_Dsub and some Src_i, i ≠ Ssub. */
for each N_i adjacent to Dst_Dsub in IG do
   if N = Src and i ≠ Ssub then
      return true
   endif
endfor
return false

Procedure Replace_Uses(Dest, Src, S)
/* Replace each reference to Dest with a reference to Src. */
/* If the use is in a φ-function or UPDATE operation, then */
/* push operation on stack S. */
for each use X of Dest do
   replace use of Dest at X with a use of Src
   if X is a φ-function or UPDATE operation then
      push X on stack S
   endif
endfor
return

Procedure Update_Graphs(Old, New, SSA, IG)
/* Replace node Old with node New in both SSA and IG. */
return

Function Find_Phi(Lvar_Lsub, Rvar_Rsub, stmt)
/* Find the φ-function merging Lvar_Lsub and Rvar_Rsub */
/* at the same merge point as stmt and return it. */
/* If it does not exist, return a new instance of Lvar. */
New_Nsub = Hash_Table_Lookup(Lvar_Lsub, Rvar_Rsub, stmt)
if New_Nsub = nil then
   New_Nsub = Gen_Next_SSA_Var(Lvar)
endif
return New_Nsub
```

Figure 6.8 Auxiliary procedures.

us to do this in a very efficient manner. This is accomplished by simply following the SSA def-use edges and replacing all uses of DST_i with uses of $\text{SRC}_j^{<\ldots,\text{SHIFT},\ldots>}$. Depending upon the type of use, further propagation may be possible. Several different cases must be handled during this propagation; we discuss them next. When the propagation of a change has completed, we continue the traversal of the program looking for the next offsetable array. The reverse depth-first traversal order is important so that multiple-offset arrays can be correctly handled in a single pass of the program.

As stated in the previous paragraph, we propagate the offset array information by changing all uses of the original destination array DST_i into uses of the new offset array $\text{SRC}_j^{<\ldots,\text{SHIFT},\ldots>}$. Since we are dealing with arrays, these uses can only occur in three places: an ACCESS operation, an UPDATE operation, or a ϕ-function. If this change is propagated into an array ACCESS operator, there are no more opportunities for propagation.

If the use is at an UPDATE operator we propagate the offset array through the operation when possible. It is valid to propagate through the UPDATE as long as it does not violate criterion 2 and it does not realign or redistribute the array. The propagation is accomplished by generating a new instance of the SRC array, call it SRC_k, to be the target of the UPDATE operation in place of the existing DST array instance. This new SRC_k receives the same annotations as $\text{SRC}_j^{<\ldots,\text{SHIFT},\ldots>}$, and then is propagated to all its uses in a similar manner. If it is not possible to propagate through the UPDATE operation then a copy of the offset array may be required prior to the UPDATE. This copy is inserted by a subsequent phase which we describe in the next subsection.

When propagating an offset array into a ϕ-function, in addition to verifying criteria 1 and 3, it is only valid to continue the propagation if the other input to the ϕ-function is an equivalent offset array (see criterion 6). Two offset arrays are equivalent if they are from the same SSA family and have identical annotations. The one exception to this rule is if a cycle has been created (*i.e.*; one of the inputs to the ϕ-function, when its annotation is removed, is the same SSA variable being defined by the ϕ-function).

When it is possible to propagate through the ϕ-function, we need to select the correct SSA variable to receive the definition of this ϕ-function. If the ϕ-function happens to be merging identical values, we simply use one of its inputs as the target variable. Otherwise we look for a ϕ-function at the same merge point whose inputs are the unannotated variables of the current ϕ-function. If found, we use the SSA variable that it defines, otherwise we generate a new instance of the SSA variable in the same manner as we did for UPDATE. In any case, the variable is annotated with the same annotation as the input variables and is propagated forward. If it is not possible to propagate through a ϕ-function, then array copy statements must be inserted on the appropriate branches leading to the ϕ-function. These copies are added by the *Insert_Copies* routine, which is the last function called by the *Offset_Arrays* procedure and which we describe next.

6.5.3 Inserting Array Copies

Once we have found all the offset arrays and propagated them as far as possible through the program, it may be necessary to insert some array copy statements to maintain the original semantics. The copy statements may be needed at points where an offset array is used to define, via an UPDATE operation or a ϕ-function, a non-offset array.

It is quite easy to determine the statements that may require a copy while we are propagating offset arrays in the *Offset_Arrays* procedure. To track these statements, we maintain a set of such statements (the code has been omitted from Figure 6.7). An UPDATE operation which is processed by the algorithm but determined not to be offsetable is added to the set. A ϕ-function which is determined not to be offsetable is also added to the set. If the ϕ-function is later determined to actually be offsetable (after the other input parameter has been processed), then it is removed from the set. The use of pruned-SSA form, where dead ϕ-functions have been eliminated, can greatly reduce the number of ϕ-functions added to the set.

After the propagation of offset arrays has completed, the procedure *Insert_Copies*, shown in Figure 6.9, is called to add the required array copies to the program. It examines each statement in the set that was produced to determine if an array copy is actually needed and to select the best placement for it. If the array copy is truly required, it takes as input the offset array and defines the array originally used by the statement. This copy statement performs all the intraprocessor data movement that was avoided at the shift operation. Note, however, that no interprocessor data movement is required.

Given an UPDATE operation from the set, an array copy statement is not required if the UPDATE is a *killing* definition. Otherwise, we insert an array copy immediately preceding the UPDATE.

For a ϕ-function which defines a non-offset array, an array copy statement must be generated for each input parameter that is an offset array. In general, the copy statements are placed on the appropriate branches leading to the merge point represented by the ϕ-function. It is possible to optimize this placement in the case of some loop structures. If a copy must be made for the array values coming around from the previous iteration but there is no use within the loop of the values defined by the ϕ-function, then it is possible to move the array copy out of the loop by placing it on the loop exit branch. The copy is moved to the shallowest nesting level such that it

Procedure *Insert_Copies*(*CFG*, *SSA*, *stmt_set*)
/* *stmt_set* is produced in *Offset_Arrays* as described */

for each *stmt* in *stmt_set* **do**
  switch (*stmt*)

  case UPDATE operation: $Dst_{Dsub} = Update(Lvar_{Lsub}^{Lanot}, section, values)$:
    if (*section* does not specify the entire array) **then**
      insert $Dst = Lvar^{Lanot}$ immediately preceding *stmt* in *CFG*
    endif
    break

  case ϕ-function: $Dst_{Dsub} = \phi(Lvar_{Lsub}^{Lanot}, Rvar_{Rsub}^{Ranot})$:
    if ($Lanot \neq$ nil) **then** /* *Rvar* is handled similarly. */
      insert $Dst = Lvar^{Lanot}$ on appropriate branch in *CFG*
      /* optimize placement when possible. */
    endif
    break

  endswitch
endfor
return

Figure 6.9 Algorithm to insert array copy statements.

still dominates all uses. This is advantageous in situations where it is allowable for an array to be an offset array inside a loop nest but not outside. The full copy is only performed when the loop nest is exited rather than on each iteration.

Consider the example shown in Figure 6.10. In the original program, the definition and use of array `B` inside the `DO`-loop can be made offset arrays, but `B` cannot be an offset array after the loop due to the modification of array `A`. In such a case the ϕ-function for array `B` at the loop header cannot be made an offset array, and normally the copy statement would be placed on the appropriate branch. But in this situation, that would result in the copy statement being placed in the loop. But since $\texttt{B}_1$ is not referenced within the loop body it is only necessary to create a full, shifted copy of array `A` on the loop exit branch, as is shown in Figure 6.10(c).

The insertion of array copy statements into the program raises a concern. We must answer the question of whether these inserted copy statements can generate more data movement than was specified in the original program. The following lemmas and theorem states that a copy statement created for an offset array cannot execute more times than the original shift operations that generated the offset array.

Lemma 6.1 Given a copy statement C created for an offset array SRC^{anot} which was generated by a set of OVERLAP_SHIFT operations $\{S_1, S_2, S_3, \ldots\}$,

```
REAL, ARRAY(n,n) ::  A,B
DO

  A = ···
  B = CSHIFT(A,+1,1)
    = B
ENDDO
A = ···
  = B
```

(a) Original program

```
REAL, ARRAY(n,n) ::  A0,B0
DO
  B1  = φ(B0,A2<+1,0>)
  A2  = ···
  CALL overlap_cshift(A2,+1,1)
    = A2<+1,0>
ENDDO
A3  = ···
  = B1
```

(b) Partial SSA representation

```
REAL, ARRAY(n,n) ::  A,B
DO
  A = ···
  CALL overlap_cshift(A,+1,1)
    = A<+1,0>
ENDDO
B = A<+1,0>  {copy statement placed here}
A = ···
  = B
```

(c) Optimized program

Figure 6.10 Optimizing the placement of the array copy statement.

any path from the beginning of the program (*Root*) to C must go through at least one S_i.

Proof Assume there exists a path P_1: *Root* $\xrightarrow{*} C$ that does not contain an OVERLAP_SHIFT operation S_i. Since C is an inserted copy statement for an offset array, there must exist an OVERLAP_SHIFT operation S_j and a path P_2: $S_j \xrightarrow{*} C$. Since both P_1 and P_2 end at C, and P_1 does not contain S_j (by assumption), then there must exist a merge point X that joins P_1 and P_2 prior to C. X must contain a ϕ-function which merges the values of SRC^{anot} generated at S_j with the other values of SRC that reach C along *Root* $\xrightarrow{*} X$. But our algorithm only propagates an offset array through a ϕ-function when the ϕ-function merges identical offset arrays. Thus there must exist an OVERLAP_SHIFT operation S_i identical to S_j on *Root* $\xrightarrow{*} X$ which contradicts our original assumption. $\square$

Lemma 6.2 C cannot be more deeply nested than all $S_i \in \{S_1, S_2, S_3, \ldots\}$.

Proof Assume C is contained in a loop which does not contain an OVERLAP_SHIFT operation S_i. Since C is an inserted copy statement for an offset array, there must exist an OVERLAP_SHIFT operation S_j outside the loop and a path P: $S_j \xrightarrow{*} C$. The path P must contain a ϕ-function to merge the values reaching C from S_j with those that reach C from the back edge of the loop. By the same argument used in the proof of Lemma 6.1, there must exist an OVERLAP_SHIFT operation S_i within the loop that reaches the back edge of the loop, thus contradicting our original assumption. □

> **Theorem 6.1** An inserted copy statement C is never executed more often than $\{S_1, S_2, S_3, \ldots\}$, the set of OVERLAP_SHIFT operations which generated the offset array for which the copy statement was required.

Proof The theorem follows directly from the preceding two lemmas. □

6.5.4 Cost Analysis

During our offset array algorithm each SSA def/use edge is processed at most once. Thus our algorithm is guaranteed to terminate. This also means that our algorithm is quite efficient. The cost of the algorithm is actually dominated by the cost of generating SSA form and building the interference graph, both of which are $O(n^2)$ in the worst case (although building SSA is $O(n)$ in practice [65]). Once these structures are built, the rest of the algorithm is linear. Finding offsetable arrays is $O(n)$ and their propagation through the program is $O(e)$. In addition, the checking of interferences is $O(i)$. Here n is the size of the program, e is the number if edges in the SSA graph, and i is the number of edges in the interference graph.

6.6 Code Generation Issues

6.6.1 Scalarization of Offset Arrays

It is important to note that the exploitation of offset arrays necessitates changes to our compilation model's scalarization process. The scalarization methodology described in Section 4.8.1 utilizes the fact that all operands of an array statement had been made perfectly aligned by an early phase of the compiler. Thus a naive scalarization of such a statement always produces correct code.

Unfortunately, the use of offset arrays invalidates the perfect alignment property of an array statement. In such cases it is possible for a naive scalarization of a statement to generate a loop-carried true dependence, and in doing so incorrectly change the semantics of the statement. Our scalarization process must be capable of handling such a situation. In Chapter 8 we present an advanced scalarization algorithm to address this problem.

6.6.2 EOSHIFT Offset Arrays

In this chapter we have not discussed the EOSHIFT operation, which is similar to CSHIFT except that new boundary values are given to array elements which wrap-around the end of the array. These new values create a conflict between EOSHIFT and CSHIFT as to what values should actually be placed in the overlap areas; *i.e.*, an overlap area cannot simultaneously hold both the new boundary elements specified by an EOSHIFT and the wrap-around elements of a CSHIFT.

To handle EOSHIFTs we modify our annotation vector to also include the boundary value for each dimension (this value is NULL for CSHIFTs). We then modify our codegen methodology to use buffers rather than the overlap areas for EOSHIFTs – this allows multiple EOSHIFTs and CSHIFTs to be offset simultaneously.

To efficiently use buffers one must apply loop peeling to peel off the iterations that reference the buffers. This allows us to hoist the conditionals that choose between subgrid elements and buffer elements out of the loop. The performance penalty for leaving the conditionals in the loop outweighs the performance gain we obtained by not making a full copy of the array. The peeling required to hoist these conditionals is identical to the context splitting optimization we exploit for optimizing SIMD context switches. This optimization is described in Section 7.3.

6.7 Experimental Results

To verify the usefulness of this optimization we have implemented it within the Fortran D system. Since the global offset array algorithm handles a strict superset of the cases the local algorithm can handle, we have only implemented the global algorithm. The implementation required that we enhance the system's data flow analysis and SSA representation so that it could properly handle Fortran90 array constructs as described in Chapter 5. The implementation is complete in that it performs the described analysis, propagation, transformations, and insertions of array copies when necessary. Unfortunately, the implementation is not fully integrated with the back-end of the Fortran D system, and thus it does not produce executable code. Instead it outputs modified Fortran90 code that contains annotations but which cannot be compiled.

Due to this limitation, we have split our experimental work into two phases. In the first phase we performed static experiments in which we ran our compiler on a set of Fortran90D/HPF codes and examined the results. In the second phase we then hand-translated some of the codes into executable form which exploits offset arrays. We then timed them on a variety of parallel architectures.

6.7.1 Static Experiments

To test our algorithm's ability to identify and exploit offset arrays, we modified our compiler to collect statistics and then we ran a set of Fortran90D and HPF codes through it. The majority of the codes we tested were taken from the High Performance Fortran Application (HPFA) project [86, 128] at the Northeast Parallel Architecture Center (NPAC), Syracuse University. [6] The HPFA project contains codes ranging from code fragments and benchmarking kernels to complete applications. Many of the codes are available in Fortran90 and HPF source, as well as Fortran77 and message-passing source. In addition to the HPFA codes, we also tested our compiler on the Advanced Regional Prediction System (ARPS) [66] weather prediction code from the Center for Analysis and Prediction of Storms (CAPS), University of Oklahoma. [7]

Our algorithm worked as expected and was able to handle some difficult codes. As an example, consider the before and after snap-shots of a section of code from an N-BODY kernel shown in Figure 6.11. As can be seen, our optimization was able to turn the array Q into an offset array. The uses of the array Q were changed and the algorithm was able to successfully propagate the offset array through several updates as well. The propagation stopped at the second CSHIFT which redefined the array Q. It was not necessary to insert a copy statement at that point since the definition is a killing definition. It was not possible to make the second CSHIFT an offset array since the shift amount was not a compile-time constant.

A summary of our results is given in Table 6.2. Within the 134 modules which we ran through our compiler we encountered 119 shift operations. Of these 119 shift

[6] See http://www.npac.syr.edu/hpfa for information on NPAC's HPFA project.

[7] For more information on ARPS and CAPS, see http://wwwcaps.uoknor.edu/.

```
q=CSHIFT(q,SHIFT=1,DIM=1)
k=npts/2
dx(0:k-1)=p(0:k-1,PX)-q(0:k-1,PX)
dy(0:k-1)=p(0:k-1,PY)-q(0:k-1,PY)
dz(0:k-1)=p(0:k-1,PZ)-q(0:k-1,PZ)
sq(0:k-1)=dx(0:k-1)**2+dy(0:k-1)**2
&          +dz(0:k-1)**2
dist(0:k-1)=SQRT(sq(0:k-1))
fac(0:k-1)=p(0:k-1,MM)*q(0:k-1,MM)
&          /(dist(0:k-1)*sq(0:k-1))
tx(0:k-1)=fac(0:k-1)*dx(0:k-1)
ty(0:k-1)=fac(0:k-1)*dy(0:k-1)
tz(0:k-1)=fac(0:k-1)*dz(0:k-1)
p(0:k-1,FX)=p(0:k-1,FX)-tx(0:k-1)
q(0:k-1,FX)=q(0:k-1,FX)+tx(0:k-1)
p(0:k-1,FY)=p(0:k-1,FY)-ty(0:k-1)
q(0:k-1,FY)=q(0:k-1,FY)+ty(0:k-1)
p(0:k-1,FZ)=p(0:k-1,FZ)-tz(0:k-1)
q(0:k-1,FZ)=q(0:k-1,FZ)+tz(0:k-1)

q=CSHIFT(q,SHIFT=npts/2,DIM=1)
p(:,FX)=p(:,FX)+q(:,FX)
p(:,FY)=p(:,FY)+q(:,FY)
p(:,FZ)=p(:,FZ)+q(:,FZ)
```

(a) Before offset array optimization

```
CALL OVERLAP_CSHIFT(q,SHIFT=1,DIM=1)
k=npts/2
dx(0:k-1)=p(0:k-1,PX)-q<1,0>(0:k-1,PX)
dy(0:k-1)=p(0:k-1,PY)-q<1,0>(0:k-1,PY)
dz(0:k-1)=p(0:k-1,PZ)-q<1,0>(0:k-1,PZ)
sq(0:k-1)=dx(0:k-1)**2+dy(0:k-1)**2
&          +dz(0:k-1)**2
dist(0:k-1)=SQRT(sq(0:k-1))
fac(0:k-1)=p(0:k-1,MM)*q<1,0>(0:k-1,MM)
&          /(dist(0:k-1)*sq(0:k-1))
tx(0:k-1)=fac(0:k-1)*dx(0:k-1)
ty(0:k-1)=fac(0:k-1)*dy(0:k-1)
tz(0:k-1)=fac(0:k-1)*dz(0:k-1)
p(0:k-1,FX)=p(0:k-1,FX)-tx(0:k-1)
q<1,0>(0:k-1,FX)=q<1,0>(0:k-1,FX)+tx(0:k-1)
p(0:k-1,FY)=p(0:k-1,FY)-ty(0:k-1)
q<1,0>(0:k-1,FY)=q<1,0>(0:k-1,FY)+ty(0:k-1)
p(0:k-1,FZ)=p(0:k-1,FZ)-tz(0:k-1)
q<1,0>(0:k-1,FZ)=q<1,0>(0:k-1,FZ)+tz(0:k-1)

q=CSHIFT(q<1,0>,SHIFT=npts/2,DIM=1)
p(:,FX)=p(:,FX)+q(:,FX)
p(:,FY)=p(:,FY)+q(:,FY)
p(:,FZ)=p(:,FZ)+q(:,FZ)
```

(b) After offset array optimization

Figure 6.11 Offset array optimization on an N-BODY code.

operations, only 5 appeared in stand-alone assignment statements (*e.g.*, like the two appearing in Figure 6.11); all the remaining shifts had started out as part of larger expressions and were hoisted out by our communication generator (see Section 4.5).

Our algorithm was then able to analyze 74 of the 119 shifts. The 45 shifts that we were not able to analyze appeared in the ARPS weather code. In that code most arrays are three dimensional and are distributed in a (BLOCK,BLOCK,BLOCK) manner. However, there are several arrays with an additional time-step dimension that is not distributed. The code contained 45 shifts of such arrays, where the fourth dimension contained a scalar subscript. Currently our implementation is limited to handling only the shifting of full arrays. Thus we were not able to analyze these shifts, and they were left unchanged. However, since the scalar subscript always occurs on a serial dimension, it is possible to apply the offset array optimization to these references. We did not make the adjustment to our algorithm due to additional constraints.

Of the 74 shifts analyzed, our algorithm was able to create 53 offset arrays which were referenced 70 times. Of the 53 offset arrays, 4 were multi-offset arrays. The

Parameter	Count
Modules processed	134
Shifts encountered	119
Shifts analyzed	74
Offset shifts	53
Multi-offset shifts	4
Uses of offset arrays	70
Offset-able updates	6
Offset-able ϕ-function	0
Inserted copies	1

Table 6.2 Results of offset array static experiments.

reason that 21 of the 74 shifts were not offsetable was because the operations contained shift amounts that were not compile-time constants; almost all of these occurring in the ARPS code. These 21 shifts could have been handled if we had decided to support the generation of multi-version code with an appropriate run-time check of the shift amount.

As was previously noted, almost all of the shift operations encountered were originally parts of larger expressions. Some of these were explicit calls to the Fortran90 intrinsics, while others were implicit in the use of array notation. In either case, it appears that a very localized analysis algorithm is all that is currently necessary to make this optimization effective. In fact, a single-statement recognizer performed before our communication generation phase would be able to handle 95% of the shifts. Note that such a single-statement recognizer would have to be able to handle explicit and implicit shifts equally well.

Since our compilation model includes the communication generation phase, which hoists communication operations out of expressions, the simplest analysis algorithm we would find acceptable would be the local analysis algorithm that works at the basic block level. Such an algorithm would be able to handle 99% of the shifts encountered in our study. We found only a single case where the definition and use of a shifted array were separated by control-flow. In that case the offset array was propagated into a ϕ-function, but that ϕ-function was determined not to be offsetable. Thus Table 6.2 reports that one copy statement was inserted.

It is possible and very likely that this situation will change. Most of the codes available for our test appear to simply be vectorized versions of Fortran77 codes. As Fortran90 and HPF gain in popularity more codes will be developed which directly exploit native Fortran90 constructs. The codes developed will become increasing complex, possibly even machine generated. If a programmer or the programmer's tool attempts to optimize the code by performing transformations at the array level such as common subexpression elimination, partial redundancy elimination or invariant code motion, it would be very possible that a global analysis algorithm, such as the one presented in Section 6.5 would be required to effectively exploit offset arrays.

6.7.2 Dynamic Experiments

To demonstrate the effectiveness of this optimization, we have compiled and executed the code from Figure 6.5 on an IBM SP-2 with 8 processors, a Thinking Machines CM-5 with 128 processors, and a MasPar MP-1 with 16K processors. Figures 6.12, 6.14, and 6.16 compare the execution times of the original program displayed in Figure 6.5(a) and the optimized program shown in Figure 6.5(d) for varying subgrid sizes. The figures show that the program exploiting offset arrays gives a speed-up of a factor of two or more for the larger subgrid sizes. The figures also display the time to execute the four CSHIFTs of Figure 6.5(b) and the four calls to OVERLAP_SHIFT of Figure 6.5(d). Figures 6.13, 6.15, and 6.17 shows the percent of total execution

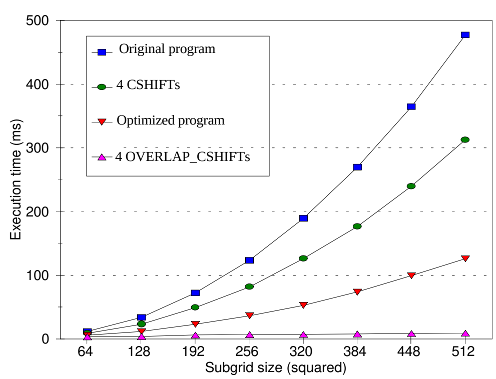


Figure 6.12 Timings for 5-point stencil computation on IBM SP-2.

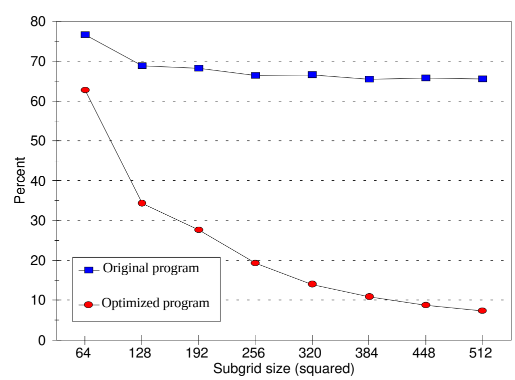


Figure 6.13 Percent of execution time spent performing shift operations.

time that is spent performing the shift operations for the respective machines. We can see that in many cases the amount of execution time that is spent performing the four CSHIFT operations is actually more than the time spent performing the desired computation. In fact, on the SP-2 the CSHIFT operations account for 65% of the total execution time for the largest subgrid. The corresponding number for the OVERLAP_SHIFT operations is 7.3%. Additional experimental results are presented in Chapter 9.

6.8 Related Work

6.8.1 The CM-2 Stencil Compiler

The stencil compiler [36, 39] for the CM-2 avoids the memory-to-memory copying for shift operations that occur within specific, stylized, array-assignment statements. These statements, or *stencils*, must be in the form of a weighted sum of circularly-shifted arrays. Not only does the compiler eliminate intraprocessor data movement for these statements, it also optimizes interprocessor data movement by using the CM-2's multidimensional and bidirectional interconnect, and exploits hand-optimized library microcode to minimize data movement between local memory and registers. However, use of this special-purpose compiler requires that the user identify these stylized assignment statements in the source program and separate them into their own subroutine.

Our compilation scheme is a superset of the stencil compiler. We hoist all shift operations, whether implicit or explicit, out of expressions and assign them to array temporaries. This allows us to handle all shift operations, whether part of a larger expression or not, in a uniform manner. Since hoisted temporaries have short life spans and thus never have conflicting uses, we will always be able to make them into offset arrays.

We present a more complete comparison between our work and the stencil compiler in Chapter 9 where we discuss our own stencil compilation strategy.

6.8.2 Scalarizing Compilers

Previous work on Fortran90D [62], like the stencil compiler, is capable of avoiding some intraprocessor data movement for stylized expressions. In this case, the expressions have to use array syntax. The compiler translates the array syntax expressions

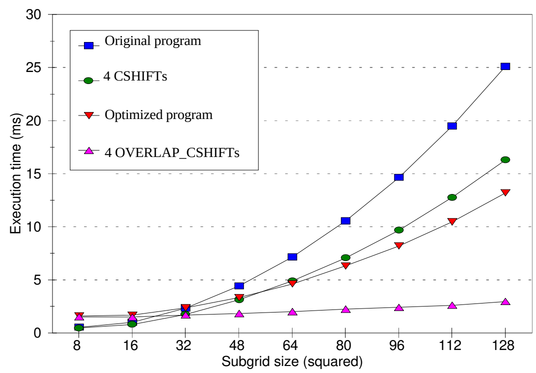


Figure 6.14 Timings for 5-point stencil computation on CM-5.

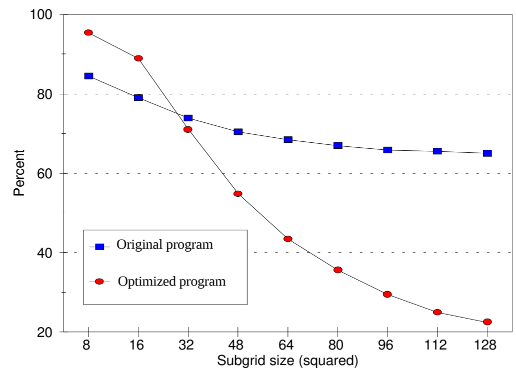


Figure 6.15 Percent of execution time spent performing shift operations.

into equivalent Fortran77D code using FORALL statements. It is then the job of the Fortran77D back-end, using dependence information, to determine the exact amount of interprocessor communication required. Unfortunately, any call to CSHIFT, whether in an assignment statement or as part of an expression, still makes a full copy of the array. As with the stencil compiler, our work is a superset of this work.

6.8.3 Functional and Single-Assignment Languages

Functional and many high-level languages have value semantics, and thus do not have the concept of state and variable as in Fortran. Naive compilation of such languages causes the insertion of many copy operations of aggregate objects to maintain program semantics. It is imperative that compilers for such languages eliminate a majority of the unnecessary copies if they hope to generate efficient code. This task is known as *copy optimization*. Although our problem differs somewhat from those faced by functional languages, it is worthwhile to review their work.

Schwartz [144] characterizes the task of copy optimization as the destructive use (reuse) of an object v at a point P in the program where it can be shown that all other objects that may contain v are *dead* at P. He then develops a set of *value transmission functions* that can be used to determine the safety of a destructive use within the language SETL.

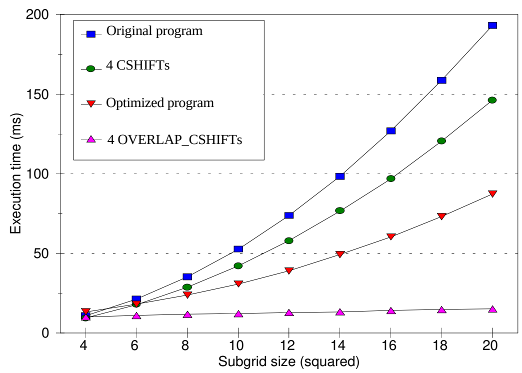


Figure 6.16 Timings for 5-point stencil computation on 16K MasPar MP-1.

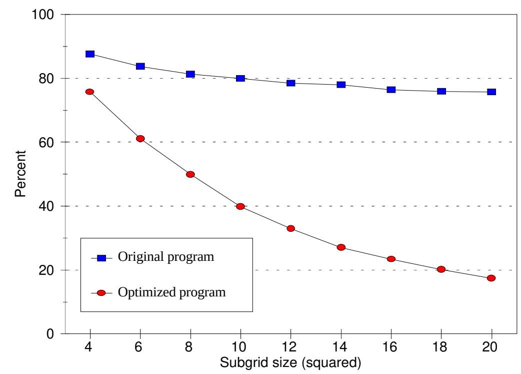


Figure 6.17 Percent of execution time spent performing shift operations.

Gopinath and Hennessy [77] address the problem of copy elimination by *targeting*, or the proper selection of a storage area for evaluating an expression, the goal of which is to reuse the storage of the input parameters. For the lambda calculus extended with array operation constructs, they develop a set of equations which, when solved iteratively to a fixpoint, specify targets for array parameters and expressions. Their approach can successfully target even divide and conquer algorithms. Unfortunately, solving their equations to a fixpoint is at least exponential in time. To avoid this cost within their implementation of the SAL language, they use type information to *"guess"* the fixpoint, which they can then verify in linear time.

Schnorf *et al.* [143] describe their efforts to eliminate aggregate copies in the single-assignment language SISAL. Their work analyzes edges in a data flow graph and attempts to determine when edges, representing values, may share storage. *Vertical substitution* attempts to link an output edge of a node with one of the input edges, thus letting them share storage. *Horizontal substitution* attempts to link sibling output edges. Our work is most closely related to the combination of vertical substitution and horizontal substitution. In their combined algorithm, they first attempt vertical substitution on a node and then horizontal substitution. Our work more or less attempts to perform vertical substitution given the constraint that our programming model implies horizontal substitution. That is we assume that all out-edges of an operation share the same storage (equivalent to horizontal substitution), and then try to find instances where an out-edge of a shift operation can share storage with the in-edge (equivalent to vertical substitution). They also rely upon pattern matching to catch a large percentage of unnecessary copies.

6.9 Summary

In this chapter we have presented a unified framework for analyzing and optimizing shift operations on distributed-memory multicomputers. The framework is capable of handling the majority of such operations, whether written by the user or generated internally by the compiler. This work supersedes prior work by others that only handled shifts embedded within expressions. In addition, the efficiency of the presented algorithms means they could replace the special-purpose pattern matching algorithms used by many compilers without drastically affecting compile time. And although this work has concentrated on distributed-memory machines, the optimizations presented are also applicable to scalar and shared-memory machines. In Chapter 9 we discuss an optimization designed to minimize the residual interprocessor communication of shift operations.

Chapter 7

Context Optimization

7.1 Introduction

SIMD machines offer impressive cost/performance ratios, and they are very well suited for a large body of engineering and scientific applications. However, current compilers for SIMD machines do not come close enough to exploiting the full potential of these machines. To obtain the best performance on such architectures, aggressive compilation techniques are required. One issue that SIMD compilers must address is generating code to change the machine context; *i.e.*, disabling processors not involved in the current computation.

As described in Section 2.1.2 there is a need with SIMD machines to explicitly turn processors on and off. This is due to the fact that there is only a single instruction stream and not all processors need to execute each instruction. Only processors containing data related to the current instruction should execute it. If a processor is not to execute a set of instructions, it must be explicitly "masked out". However, changing the machine state, or *context*, is an expensive operation. Setting the machine context is an overhead that one must pay to execute on a SIMD architecture.

As can be seen in the two simple examples given in Section 4.8.3, the code required to set the context of the PE array can include multiple logical and arithmetic operations. This code can be a significant portion of the work performed within a subgrid loop. It is our goal to reduce the impact of this overhead for programs compiled for SIMD machines.

Our compiler strategy has a two-pronged approach to minimize the expense of context switching. First, we rearrange the program statements so that as subgrid loops are generated, as many statements as possible that execute under the same context are placed within the same subgrid loop. Second, we alter the order in which subgrid elements are processed by performing loop transformations on the subgrid loops. These loop transformations will allow us to hoist the code that sets the machine context out of the loops and thus reduce the number of context changes.

These optimizations are described in detail in Sections 7.2 and 7.3, and some code generation issues are discussed in Section 7.4. We present some preliminary results in Section 7.5. Section 7.6 discusses related work by others. We conclude in Section 7.7 with a brief summary.

And although this chapter focuses on SIMD architectures, the optimizations presented are applicable to programs for MIMD architectures as well. Where appropriate, we discuss how these optimizations apply to MIMD machines.

7.2 Context Partitioning

As explained in Sections 4.8.1 and 4.8.3, a single subgrid loop nest is generated for adjacent Fortran90 array statements which are congruent, [8] and these statements all execute within the same context. However, unless an effort is made to make congruent array statements adjacent, many small subgrid loops may still be generated. Sabot recognized this problem, and recommended that users of the CM Fortran compiler rearrange program statements, when possible, to avoid the inefficiencies of such subgrid loops [142]. In order to alleviate this problem automatically, our compiler has an optimization phase that reorders statements within a basic block. The reordering attempts to create separate partitions of congruent array statements, scalar statements, and communication statements. We call this optimization *context partitioning.* The partitioning could group scalar statements and communication statements together; however, we prefer to separate them so that the communication operations can be further optimized by subsequent phases (see Section 9.5).

7.2.1 Context Partitioning Algorithm

To accomplish context partitioning, we use an algorithm proposed by Kennedy and McKinley [103]. While they were concerned with partitioning parallel and serial loops into fusible groups, we are partitioning Fortran90 statements into congruence classes. The algorithm works on the *data dependence graph* (DDG) [117] which must be acyclic. Since we apply it to a set of statements within a basic block, our dependence graph contains only *loop-independent* dependences [13] and thus meets that criterion. Besides the DDG, the algorithm takes two other arguments: the set of congruence

[8] Recall from Section 4.8.1 that congruent array statements have the same iteration space and operate on arrays that have identical distributions.

classes contained in the DDG, and a priority ordering of the congruence classes. We create a congruence class for each set of congruent array statements and then add separate classes for scalar statements and communication statements.

An outline of the context partitioning algorithm is shown in Figure 7.1. For each congruence class in priority order, the algorithm makes a single pass over the DDG. During the pass it greedily merges nodes for the given class. Two nodes from the same class may be merged if there does not exist a path between them that contains a node from another class. A path containing a node from another class is termed a *bad path*. When two nodes are merged, the DDG is updated to reflect it.

The strength of the algorithm comes in its ability to choose in constant time the correct node *pred* with which to fuse the given node n of the selected congruency class. The logic to do this is not shown in Figure 7.1 due to its complexity. Intuitively, n can fuse with all its ancestors from which there does not exist a bad path, but it cannot

---

Procedure *Context_Partitioning*
Input: *Stmts*, the set of statements to be partitioned.
Classes, the set of congruence classes.
Priority, the priority ordering for *Classes*.
Output: A linear list of *Stmts* that has been partitioned.

```
generate DDG, the data dependence graph for Stmts

for each c ∈ Classes in decreasing priority do
   for each node n ∈ DDG do
      if n is DATA-READY then add n to Worklist
   endfor

   while Worklist ≠ ∅ do
      remove arbitrary node n from Worklist

      if class(n) = c then
         find node pred, previously processed, with which n can be merged
         if pred ≠ 0 then
            fuse nodes n and pred in DDG
         endif
      endif

      for each edge (n,m) in DDG do
         visit node m
         if m is now DATA-READY then add m to Worklist
      endfor

   endwhile
endfor

topologically sort DDG to produce final output

end Context_Partitioning
```

Figure 7.1 The Context Partitioning algorithm.

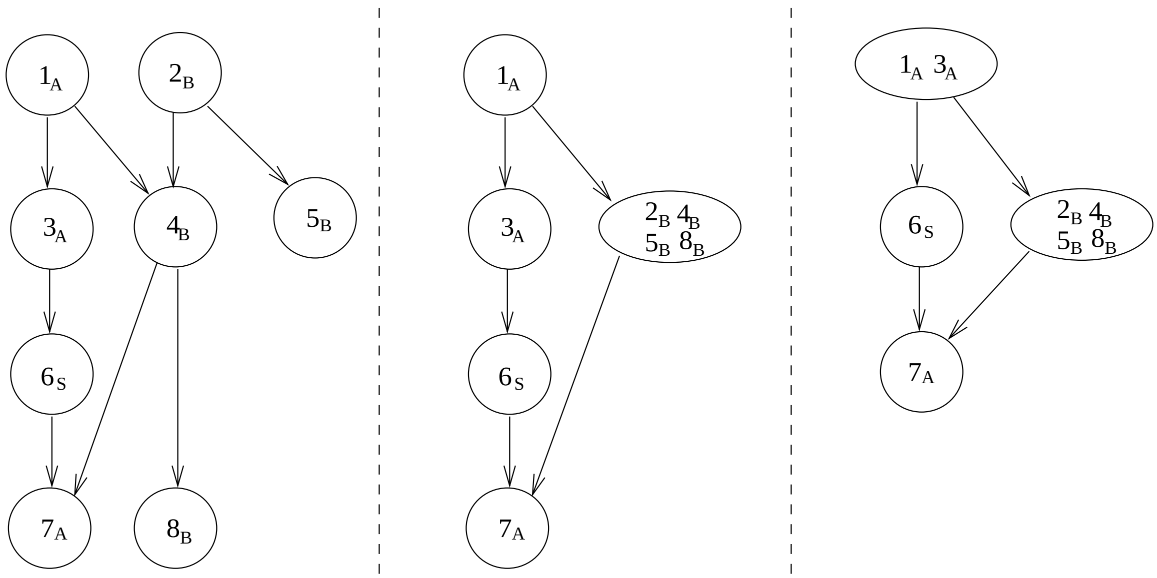
bypass any predecessor to fuse with an ancestor. It can also fuse with nodes which are neither ancestors nor descendants. As nodes are processed for a given class, the algorithm determines the highest numbered node m of class c from which there exists a bad path to n. Node n cannot fuse with m or any node with a lower number, where nodes are numbered breadth-first in their fusion group. The algorithm also computes the highest number of a direct predecessor k of n with which n can fuse. Node n cannot fuse with any node with a lower number than k. Thus the algorithm can compute the fusion node *pred* by taking the maximum of k and the next higher node than m. The interested reader is referred to the paper by Kennedy and McKinley for complete details of the algorithm [103].

Figure 7.2 displays how context partitioning would handle a block of eight statements. The statements are numbered to represent their textual order. The subscripts represent their congruence class. In this example, there are two congruence classes: A and B. Statement 6 is a scalar statement. Figure 7.2(a) shows the original source code, and Figure 7.2(b) shows the data dependence graph. Naive code generation would create six subgrid loops for these statements (only statements 4 and 5 would occur in the same subgrid loop). Figure 7.2(c) shows the data dependence graph after partitioning congruence class B, the larger of the two classes. Figure 7.2(d) displays the final result after partitioning class A. The final code is displayed in Figure 7.2(e). The modified code requires only three subgrid loops to be generated. (Note: Figures 7.2(b-d) show only true (flow) dependences; anti- and output-dependences have not been shown since they clutter the diagrams and add little benefit to the discussion, although they must be respected.)

Assigning a priority ordering to the congruence classes is required to handle class conflicts. A *class conflict* occurs when there exist dependences such that a pair of statements from one class may be merged during partitioning or a pair from another class, but not both since that would introduce a cycle in the DDG and thus make it unschedulable. The following contrived code segment, whose DDG is shown in Figure 7.3, gives an example of a class conflict:

```
s1:  A(1:100) = A(1:100) + 1.0
s2:  B(2:99) = B(2:99) * C(2:99)
s3:  C(1:100) = B(1:100)
s4:  D(2:99) = A(2:99)
```

It is possible to merge nodes *s1* and *s3* or nodes *s2* and *s4*, but we cannot merge both pairs. The priority ordering is used to determine which pair should be merged.



(b) dependence graph

(c) dependence graph after merging class B

(d) dependence graph after merging class A

1_A : X(1:256) = X(1:256) + 1.0
2_B : Z(1:100) = Z(1:100) + W(1:100)
3_A : V(1:256) = X(1:256) ** 2
4_B : W(1:100) = X(1:100) + Z(1:100)
5_B : U(1:100) = Z(1:100) + SCALAR1
6_S : SCALAR1 = SCALAR1 * V(I)
7_A : X(1:256) = W(1:256) * SCALAR1
8_B : Z(1:100) = SQRT(W(1:100))

(a) source code

1_A : X(1:256) = X(1:256) + 1.0
3_A : V(1:256) = X(1:256) ** 2
2_B : Z(1:100) = Z(1:100) + W(1:100)
4_B : W(1:100) = X(1:100) + Z(1:100)
5_B : U(1:100) = Z(1:100) + SCALAR1
8_B : Z(1:100) = SQRT(W(1:100))
6_S : SCALAR1 = SCALAR1 * V(I)
7_A : X(1:256) = W(1:256) * SCALAR1

(e) modified code

Figure 7.2 Partitioning congruent statements.

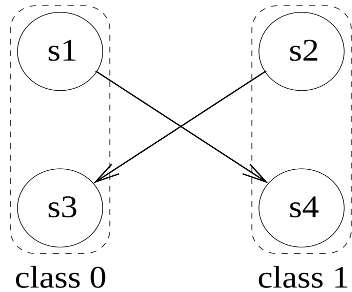


Figure 7.3 A context partitioning class conflict.

The algorithm merges pairs with a higher priority before those with a lower priority. Kennedy and McKinley have shown that choosing an optimal ordering of classes is NP-hard in the number of classes. However, since class conflicts are considered rare, a good heuristic for choosing an order should be effective. The heuristic that we have chosen is to order the array statement congruence classes by their size, largest to smallest for the given basic block, and to give the scalar and communication classes the lowest priority.

7.2.2 Context Partitioning for MIMD Machines

It should be noted that context partitioning is not a SIMD-only optimization. It is useful for Fortran90 compilers that target MIMD architectures as well. When compiling for MIMD machines, in fact any parallel architecture, loop fusion [10] is an important optimization as it can enhance the granularity of parallelism and increase the possibility of data reuse while reducing the overhead of parallel loop execution. This is particularly critical in Fortran90 compilation, where the scalarization of array statements generates many loop nests each containing a single assignment statement.

In previous work on loop fusion for parallel machines [42, 154, 159] two loops are candidates for fusion if their headers are *conformable* and there do not exist any *fusion-preventing dependences*. Two loop headers are conformable if they specify the same number of iterations and are both either parallel or sequential loops. A data dependence between two loops is fusion-preventing if after fusion the dependence becomes loop-carried and its direction is reversed.

While these criteria determine the safety of loop fusion and attempt to address its profitability on parallel machines, they are insufficient when considering loop fusion on distributed-memory architectures. This is due to the fact that they ignore the distribution and alignment of the arrays accessed within the loops, as well as the exact iteration space of the loops.

When the distribution and alignment of the arrays and the iteration space of the loops is not considered, it is possible for loops to be *over fused*. Loops are over fused when the code produced for the resulting parallel loops exhibits worse performance than the code for the separate parallel loops. As an example, consider the sample code in Figure 7.4. In Figure 7.4(a) we see two simple Fortran90 array assignment statements which operate on arrays with identical iteration spaces, but have different distributions. In Figure 7.4(b) we see the loops that would result from a naive scalarization. The Rice dHPF compiler [3] generates the code shown in Figure 7.4(c) for the two loops from Figure 7.4(b) when targeting a four processor machine. Since the scalarized loops have the same number of iterations and do not have any fusion-preventing dependences, many parallel compilers that support loop fusion would fuse the two loops into a single loop, such as seen in Figure 7.4(d). But even though the loops have the same iteration space, the arrays upon which they operate have different distributions. This causes problems for the code generator. As of April 1997, the Rice dHPF compiler produces the code seen in Figure 7.4(e) when the loops have been fused. When executed on an IBM SP-2, the performance of the code generated for the fused loops is 22% worse than for the code where the loops were not fused. This is due to the existence of the conditional and `IMOD` function within the second loop.

When faced with such loops, Tseng [152] proposes using loop distribution to avoid the complications of code generation. But this is actually doing double work: one phase of the Fortran90D compiler fuses loops only to have a later phase distribute them. The proper resolution is to avoid such loop fusion in the first place.

Others may claim that this problem can be avoided by delaying loop fusion until after subgrid loops have been generated and loop bounds have been reduced. This is dubious at best. The generation of subgrid loops often requires the creation of symbolic loop bounds which are defined at run-time. This obfuscation of the code makes it doubtful that conformable loop headers would be detected at compile-time.

To resolve this situation, loop fusion must consider the distribution and alignment of arrays as well as the iteration space of the prospective loops. These properties are those exactly captured by our definition of congruency!

We thus propose a MIMD compilation scheme that includes context partitioning prior to scalarization, and that subsequent loop fusion be restricted to consider only

```
REAL, ARRAY(100) ::  A,B
DECOMPOSITION DECOMP1(100)
DECOMPOSITION DECOMP2(100)
ALIGN A WITH DECOMP1
ALIGN B WITH DECOMP2
DISTRIBUTE DECOMP1(BLOCK)
DISTRIBUTE DECOMP2(CYCLIC)

A(:)  = A(:)  + 1
B(:)  = B(:)  + 1
```

(a) Original program

```
REAL, ARRAY(100) ::  A,B
DECOMPOSITION DECOMP1(100)
DECOMPOSITION DECOMP2(100)
ALIGN A WITH DECOMP1
ALIGN B WITH DECOMP2
DISTRIBUTE DECOMP1(BLOCK)
DISTRIBUTE DECOMP2(CYCLIC)

DO i=1,100
  A(i) = A(i) + 1
ENDDO
DO i=1,100
  B(i) = B(i) + 1
ENDDO
```

(b) After scalarization

```
REAL, ARRAY(25) ::  A,B

DO i=25*mypid-24,25*mypid
  A(i-25*(mypid-1)) = A(i-25*(mypid-1)) + 1
ENDDO
DO i=mypid,mypid+96,4
  B((i+3)/4) = B((i+3)/4) + 1
ENDDO
```

(c) Generated code without loop fusion

```
REAL, ARRAY(100) ::  A,B
DECOMPOSITION DECOMP1(100)
DECOMPOSITION DECOMP2(100)
ALIGN A WITH DECOMP1
ALIGN B WITH DECOMP2
DISTRIBUTE DECOMP1(BLOCK)
DISTRIBUTE DECOMP2(CYCLIC)

DO i=1,100
  A(i) = A(i) + 1
  B(i) = B(i) + 1
ENDDO
```

(d) After loop fusion

```
REAL, ARRAY(25) ::  A,B

DO i=mypid,25*mypid-28,4
  B((i+3)/4) = B((i+3)/4) + 1
ENDDO
DO i=25*mypid-24,25*mypid
  A(i-25*(mypid-1)) = A(i-25*(mypid-1)) + 1
  IF (imod(i-mypid,4).eq.0) THEN
    B((i+3)/4) = B((i+3)/4) + 1
  ENDIF
ENDDO
DO i=25*mypid+4,mypid+96,4
  B((i+3)/4) = B((i+3)/4) + 1
ENDDO
```

(e) Generated code with loop fusion

Figure 7.4 Example of over fusing loops for distributed-memory machines.

blocks of loops generated from congruent array statements. As an example, consider the simplified program fragment from the Shallow weather prediction benchmark code shown in Figure 7.5(a) where all the arrays are perfectly aligned and distributed. Even though the two array statements have the same number of iterations, they form two congruence classes since their iteration spaces are slightly askew. Following our proposed strategy, the loops generated for these two statements would not be fused even though they have the same number of iterations and there are no fusion-preventing dependences. The reasons for this can be seen in the rest of the figure. Figure 7.5(b) shows the statements after scalarization, and Figure 7.5(c) shows how loop bounds reduction could be used to generate an efficient SPMD node program. If on the other hand the compiler had fused the two loops, as shown in Figure 7.5(d), the compiler can no longer use loop bounds reduction to instantiate the computation partition. This is due to the fact that each PE does not process the same number of elements for each statement. Instead the compiler inserts guards around each statement, as is seen in Figure 7.5(e). This is less efficient since the guard statements must be evaluated at run-time, once per iteration of the loop. For this reason, we only fuse loops that are generated by congruent array statements.

We address the issues of context partitioning and loop fusion in more detail in the chapters that discuss scalarization and stencil compilation.

7.2.3 Cost Analysis

Given the chosen priority ordering, the context partitioning algorithm is incrementally optimal; *i.e.*, for each class c, given a partitioning of classes with higher priority, the partitioning of c results in a minimal number of partitions. The algorithm partitions the DDG in $O((N+E)C)$ time, where N is the number of statements, E is the number of dependence edges, and C is the number of congruence classes.

During subgrid loop generation, all statements in a partition are placed in the same subgrid loop. The number of subgrid loops which operate over statements with the same context is thus minimal, given the chosen priority ordering.

7.3 Context Splitting

In the first example in Section 4.8.3, which incremented `X(2:242)`, the PE array has the same context for iterations 2 through 15 of the subgrid loop; during these iterations all the PEs are active. To take advantage of this invariance, we modify

```
REAL, ARRAY(256,256) ::  X,Y,Z
DECOMPOSITION DECOMP(256,256)
ALIGN X,Y,Z WITH DECOMP
DISTRIBUTE DECOMP(BLOCK,*)

X(2:256,1:255) = F1(Z(2:256,2:256),Z(2:256,1:255))
Y(1:255,2:256) = F2(Z(2:256,2:256),Z(1:255,2:256))
```

(a) SHALLOW: weather prediction code

```
REAL, ARRAY(256,256) ::  X,Y,Z

DO j=1,255
 DO i=2,256
  X(i,j) = F1(Z(i,j+1),Z(i,j))
 ENDDO
ENDDO
DO j=2,256
 DO i=1,255
  Y(i,j) = F1(Z(i+1,j),Z(i,j))
 ENDDO
ENDDO
```

(b) After scalarization

```
REAL, ARRAY(64,256) ::  X,Y,Z

lb = max(mypid*64,65)-(mypid*64)+1
DO j=1,255
 DO i=lb,64
  X(i,j) = F1(Z(i,j+1),Z(i,j))
 ENDDO
ENDDO
ub = min(mypid*64,255)-64*(mypid-1)
DO j=2,256
 DO i=1,ub
  Y(i,j) = F1(Z(i+1,j),Z(i,j))
 ENDDO
ENDDO
```

(c) SPMD code without loop fusion

```
REAL, ARRAY(256,256) ::  X,Y,Z

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) = F1(Z(i+1,j+1),Z(i,j+1))
 ENDDO
ENDDO
```

(d) After loop fusion

```
REAL, ARRAY(64,256) ::  X,Y,Z

lb = max(mypid*64,65)-(mypid*64)
ub = min(mypid*64,255)-64*(mypid-1)
DO j=1,255
 DO i=lb,ub
  IF (i.lt.64) THEN
   X(i+1,j) = F1(Z(i+1,j+1),Z(i+1,j))
  ENDIF
  IF (i.gt.0) THEN
   Y(i,j+1) = F1(Z(i+1,j+1),Z(i,j+1))
  ENDIF
 ENDDO
ENDDO
```

(e) SPMD code with loop fusion

Figure 7.5 Example of over fusing loops for the SHALLOW weather prediction code.

the subgrid loop by performing *loop splitting*, also called *index set splitting* [23, 161]. By splitting the iteration space into disjoint sets, each requiring a single context, we can safely hoist the context setting code out of the resulting loops. We call this optimization *context splitting*.

For the reference `X(2:242)`, the iteration space of the subgrid loop is split into 3 sets: {1}, {2:15}, and {16}. Applying context splitting to the subgrid loop produces the code shown in Figure 7.6. Compared to the original subgrid loop, this code has hoisted the call to `Set_Context` out of the loop, eliminated several arithmetic and logical operations, and the PE array context is now set only three times compared to the original 16 times. As we can see from this simple example, context splitting significantly reduces the number of times that the PE array context must be set for a subgrid loop. Whereas the original subgrid loop set the context once per iteration, it is now set only three times, no matter how large the subgrid is.

Unlike context partitioning, context splitting is a SIMD-only optimization. For MIMD machines, compilers can often side-step the issue that is addressed by context splitting. Since each PE in a MIMD machine also has control logic, a compiler can generate a program such that each PE determines the loop bounds for its own subgrid loop. By reducing the loop bounds, the compiler can often avoid iterations for which the PE has no work and thus does not need to introduce any guard statements into the subgrid loop body in those cases [74, 152].

We now discuss the details of context splitting. To simplify the discussion, we first discuss one-dimensional CYCLIC, BLOCK, and BLOCK_CYCLIC distributions, and then show how to combine one-dimensional splitting to handle multidimensional cases.

```
Set_Context(iproc ≥ 2)
X′(1) = X′(1) + 1.0
Set_Context(.TRUE.)
DO I = 2, 15
   X′(I) = X′(I) + 1.0
ENDDO
Set_Context(iproc ≤ 2)
X′(16) = X′(16) + 1.0
```

Figure 7.6 Context splitting applied to subgrid loop for `X(2:242)`.

Our canonical example in the following presentation will be the statement `X(N:M) = X(N:M) + 1.0`. Context splitting of subgrid loops for full arrays that do not evenly fill the machine is treated simply as a special case, where `N = 1`.

7.3.1 Context Splitting a CYCLIC Distribution

With a standard CYCLIC distribution, element `X(N)` may reside on any processor relative to the processor holding element `X(M)`. However, the offset of `X(N)` within the subgrid $\mathtt{X}'$ must be less than or equal to the offset of `X(M)`. That is given $\mu_X(N) = (iproc_N, j_N)$, $\mu_X(M) = (iproc_M, j_M)$, and $N \leq M$, then $j_N \leq j_M$ must hold. We cannot make any statement regarding the relationship of $iproc_N$ and $iproc_M$, except that $j_N = j_M$ implies $iproc_N \leq iproc_M$.

Using this information and our knowledge of CYCLIC distributions, we know that all PEs should be enabled for the subgrid iterations $j_N + 1$ to $j_M - 1$. This naturally divides the iteration space into three sets: $\{j_N\}$, $\{j_N{+}1{:}j_M{-}1\}$, and $\{j_M\}$. Figure 7.7 depicts this situation. When $j_N = j_M$, the second iteration set is empty and the first and third set are merged into a single set. The subgrid loop after context splitting now looks like that shown in Figure 7.8.

The code can be simplified if `N` and/or `M` are known constants. When `N` and `M` are both constants, the `IF`-test can be evaluated at compile-time and only the code for the appropriate branch needs to be generated. In the case where the operation is over the full array but the array does not evenly fill the machine, we know that `N = 1`. In this situation the `IF`-test is unnecessary and the pre-loop statements can be merged into the `DO`-loop. This is equivalent to peeling off the last iteration of the subgrid loop and hoisting the context setting code accordingly. The result is shown in Figure 7.9.

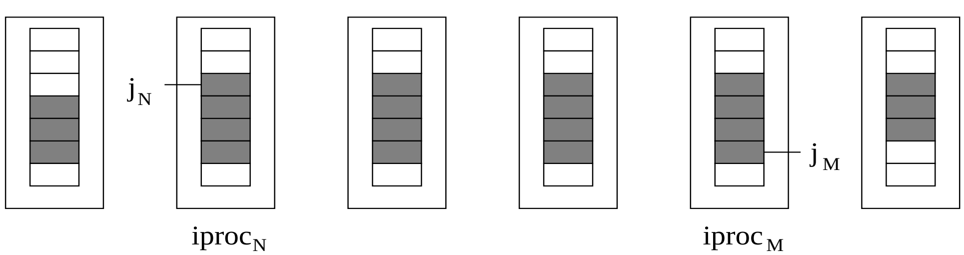


Figure 7.7 X(N:M) when X has a CYCLIC distribution.

```
iproc_N = (N-1) mod P + 1
iproc_M = (M-1) mod P + 1
j_N = ⌈N/P⌉
j_M = ⌈M/P⌉
IF (j_N = j_M) THEN
   Set_Context(iproc ≥ iproc_N .AND. iproc ≤ iproc_M)
   X′(j_N) = X′(j_N) + 1.0
ELSE
   Set_Context(iproc ≥ iproc_N)
   X′(j_N) = X′(j_N) + 1.0
   Set_Context(.TRUE.)
   DO I = j_N+1, j_M-1
     X′(I) = X′(I) + 1.0
   ENDDO
   Set_Context(iproc ≤ iproc_M)
   X′(j_M) = X′(j_M) + 1.0
ENDIF
```

Figure 7.8 Context splitting a CYCLIC distribution.

```
iproc_M = (M-1) mod P + 1
j_M = ⌈M/P⌉
Set_Context(.TRUE.)
DO I = 1, j_M-1
   X′(I) = X′(I) + 1.0
ENDDO
Set_Context(iproc ≤ iproc_M)
X′(j_M) = X′(j_M) + 1.0
```

Figure 7.9 Context splitting a CYCLIC distribution where N=1.

7.3.2 Context Splitting a BLOCK Distribution

Given a BLOCK distribution, the element `X(N)` always reside on a processor that has a number less than or equal to the processor holding the element `X(M)`; that is, $iproc_N \leq iproc_M$. Figure 7.10 shows the affected elements of array `X` for the assignment `X(N:M) = X(N:M) + 1.0` when `X` has a BLOCK distribution. As can be seen, all processors between $iproc_N$ and $iproc_M$ are enabled for all subgrid elements, whereas all processors outside that range are disabled for all subgrid elements. Processor $iproc_N$ is enabled at iteration j_N and subsequent iterations. Processor $iproc_M$ is enabled only for iterations up to and including j_M.

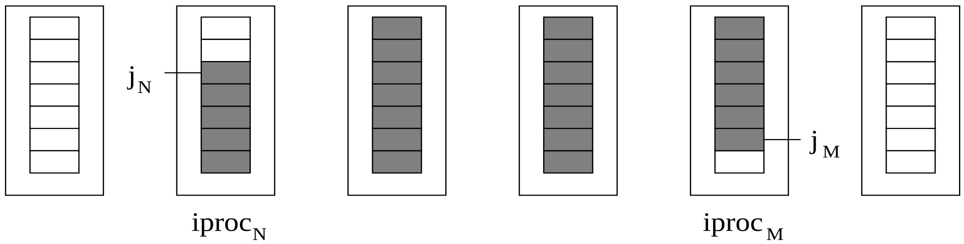


Figure 7.10 X(N:M) when X has a BLOCK distribution.

The difficulty in context splitting for a BLOCK distribution comes from distinguishing the case where $j_N \leq j_M$ from the case where $j_N > j_M$. If we let `LO = min(`$j_N, j_M + 1$`)` and `HI = max(`$j_N - 1, j_M$`)`, then the iteration space is naturally split into the following three sets: {1:`LO`−1}, {`LO:HI`}, and {`HI`+1:`Extent`}. The context for the first iteration set is the processor set $\{iproc_N + 1 : iproc_M\}$. The processor set for the second iteration set includes both $iproc_N$ and $iproc_M$ if $j_N \leq j_M$ holds, otherwise it excludes both. The context for the third iteration set is processors $\{iproc_N : iproc_M - 1\}$. Figure 7.11 shows the subgrid loop after context splitting.

This code can be greatly simplified if both `N` and `M` are compile-time constants, in which case both `IF` expressions can be eliminated. In addition, if `LO = 1` or `HI = Extent` then the body of the first or last `DO`-loop, respectively, is not executed. That `DO`-loop and its associated call to `Set_Context` can then be safely eliminated. In the case where the operation is over the entire array but the array does not evenly fill the machine, the code can be simplified to that shown in Figure 7.12.

7.3.3 Context Splitting a BLOCK_CYCLIC Distribution

Context splitting for a BLOCK_CYCLIC distribution is more complex. Here we show how context splitting can be used to optimize a subgrid loop operating over such a distribution.

BLOCK_CYCLIC is similar to CYCLIC but it takes a parameter b. It first divides the dimension into contiguous chunks of size b, then distributes these chunks in the same manner as CYCLIC. Figure 7.13 shows the distribution function used to map an array index to the PE index/subgrid index pair for a BLOCK_CYCLIC distribution. Also shown is the inverse function. It is interesting to compare these to the functions for the standard BLOCK and CYCLIC distributions, given in Table 4.1. Recall that it is the inverse function that is used in the calls to `Set_Context` that occur in the

```
iprocN = ⌈N/Extent⌉
iprocM = ⌈M/Extent⌉
jN = (N-1) mod Extent + 1
jM = (M-1) mod Extent + 1
IF (jN ≤ jM) THEN
   LO = jN
   HI = jM
ELSE
   LO = jM + 1
   HI = jN - 1
ENDIF
Set_Context(iproc > iprocN .AND. iproc ≤ iprocM)
DO I = 1, LO-1
   X′(I) = X′(I) + 1.0
ENDDO
IF (jN ≤ jM) THEN
   Set_Context(iproc ≥ iprocN .AND. iproc ≤ iprocM)
ELSE
   Set_Context(iproc > iprocN .AND. iproc < iprocM)
ENDIF
DO I = LO, HI
   X′(I) = X′(I) + 1.0
ENDDO
Set_Context(iproc ≥ iprocN .AND. iproc < iprocM)
DO I = HI+1, Extent
   X′(I) = X′(I) + 1.0
ENDDO
```

Figure 7.11 Context splitting a BLOCK distribution.

```
iprocM = ⌈M/Extent⌉
jM = (M-1) mod Extent + 1
Set_Context(iproc ≤ iprocM)
DO I = 1, jM
   X′(I) = X′(I) + 1.0
ENDDO
Set_Context(iproc < iprocM)
DO I = jM+1, Extent
   X′(I) = X′(I) + 1.0
ENDDO
```

Figure 7.12 Context splitting a BLOCK distribution where N=1.

naive subgrid loops. In this case, the inverse function is so complex that it adds a significant amount to the overhead of the subgrid loop.

Let's consider once again our example `X(N:M) = X(N:M) + 1.0`. Figure 7.14 shows the affected elements given a block size $b = 3$ (dark lines indicate logical block boundaries). It can be easily seen that at any given subgrid index different sets of PEs contain elements from the range $N : M$. On closer inspection, it can be found that the sets of PEs that must be enabled form a logical progression. It is this progression that establishes the iteration sets for context splitting.

To ease the specification of the iteration sets we not only need to know the values of j_N and j_M, the subgrid indices of $X(N)$ and $X(M)$ respectively, we also need to know the logical blocks in which they occur. We label these blocks B_N and B_M. We use one-based indexing for the blocks just as we do for indexing subgrids and PEs.They can be calculated using the following formula:

$$B_k \;=\; \left\lfloor \frac{k-1}{b*P_1} \right\rfloor + 1 \;=\; \lceil j_k/b \rceil, \quad for \;\; k = N, M$$

Once we have determined the values of j_N, j_M, $iproc_N$, $iproc_M$, B_N, and B_M, we split the iteration space into the following five sections and determine the set of active PEs as shown:

$\{(B_N - 1) * b + 1 \;:\; j_N - 1\}$	Turn on PEs where $iproc > iproc_N$.
$\{j_N \;:\; B_N * b\}$	Turn on PEs where $iproc \geq iproc_N$.
$\{B_N * b + 1 \;:\; (B_M - 1) * b\}$	Turn on all PEs.
$\{(B_M - 1) * b + 1 \;:\; j_M\}$	Turn on PEs where $iproc \leq iproc_M$.
$\{j_M + 1 \;:\; B_M * b\}$	Turn on PEs where $iproc < iproc_M$.

Depending upon the values of N and M, one or more of the five iteration sets may be empty. In particular, when $N = 1$ the first set is empty and the second and third set enable the same PEs and can thus be merged together. It is important to notice how context splitting has reduced the computations required for the calls to

$$\mu_{block\_cyclic(b)}(i) \;=\; \left(\left\lceil \frac{(i-1) \bmod (b*P_1)+1}{b} \right\rceil, \left\lfloor \frac{i-1}{b*P_1} \right\rfloor b + i \bmod b + 1 \right)$$
$$\mu^{-1}_{block\_cyclic(b)}(iproc, j) \;=\; \left\lfloor \frac{j-1}{b} \right\rfloor (P_1 * b) + (iproc - 1) * b + (j-1) \bmod b + 1$$

Figure 7.13 BLOCK_CYCLIC distribution function and its inverse.

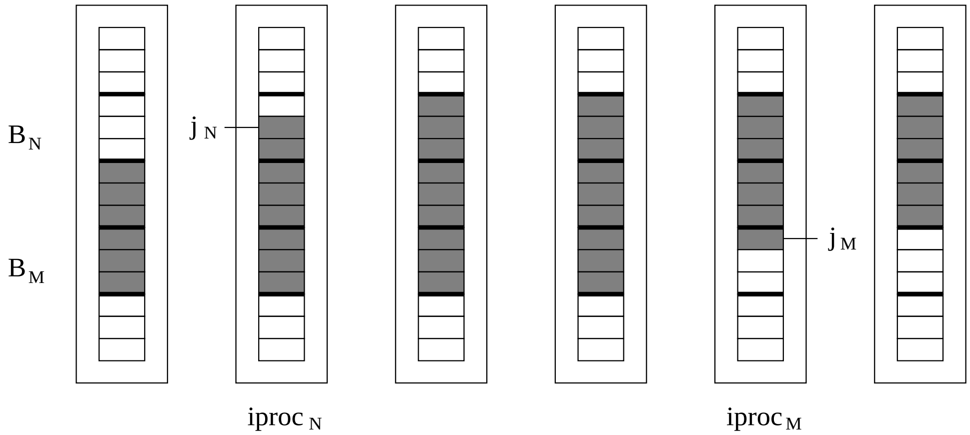


Figure 7.14 X(N:M) when X has a BLOCK_CYCLIC distribution.

`Set_Context` to simple logical comparisons; the expensive operations of the inverse mapping function are no longer required.

With a BLOCK_CYCLIC distribution, there is one special case which we must handle when performing context splitting. It occurs when $B_N = B_M$. In this situation, all the affected elements of $X(N : M)$ are at the same logical block level. Furthermore, the elements are distributed across this block level in a normal BLOCK distribution. To properly handle this case we could apply the techniques presented in Section 7.3.2, with the modification that the subgrid loop ranges from $(B_N - 1) * b + 1$ to $B_N * b$ rather than from 1 to $Extent$. Alternatively, if the block size b is small, we could simply generate the naive subgrid loop, since we know that the loop performs at most b iterations.

The subgrid loop for the statement `X(N:M)=X(N:M)+1.0` is shown in Figure 7.15. For simplicity, we have used the naive subgrid loop for the case where $B_N = B_M$. The variable `GlobalIndex` holds the value of the inverse distribution function, μ^{-1}, computed on each PE for each subgrid location.

7.3.4 Context Splitting a Multidimensional Distribution

To perform context splitting on a multidimensional distribution, we apply loop splitting on each dimension separately. This produces a set of imperfectly nested `DO`-loops. We then apply *loop distribution* [117, 129] to produce a set of perfectly nested `DO`-loops, each of which operates under a single context. The context for each loop nest is the intersection of the contexts produced for each dimension.

```
iproc_N = ⌈((N-1) mod (b*P) + 1)/b⌉
iproc_M = ⌈((M-1) mod (b*P) + 1)/b⌉
j_N = ⌊(N-1)/(b*P)⌋ b + i mod b + 1
j_M = ⌊(M-1)/(b*P)⌋ b + i mod b + 1
B_N = ⌈j_N/b⌉
B_M = ⌈j_M/b⌉
IF (B_N = B_M) THEN    !!  perform naive subgrid looping
   DO i = (B_N - 1) * b + 1, B_N * b
     GlobalIndex = ⌊(i-1)/b⌋*(P*b) + (iproc-1)* b + (i-1) mod b + 1
     Set_Context(GlobalIndex ≥ N .AND. GlobalIndex ≤ M)
     X'(i) = X'(i) + 1.0
   ENDDO
ELSE
   Set_Context(iproc > iproc_N)
   DO I = (B_N-1)*b+1, j_N-1
     X'(I) = X'(I) + 1.0
   ENDDO
   Set_Context(iproc ≥ iproc_N)
   DO I = j_N, B_N*b
     X'(I) = X'(I) + 1.0
   ENDDO
   Set_Context(.TRUE.)
   DO I = B_N*b+1, (B_M-1)*b
     X'(I) = X'(I) + 1.0
   ENDDO
   Set_Context(iproc ≤ iproc_M)
   DO I = (B_M-1)*b+1, j_M
     X'(I) = X'(I) + 1.0
   ENDDO
   Set_Context(iproc < iproc_M)
   DO I = j_M+1, B_M*b
     X'(I) = X'(I) + 1.0
   ENDDO
ENDIF
```

Figure 7.15 Context splitting a BLOCK_CYCLIC distribution.

Let's consider again the array Y2 as declared and distributed in Figures 4.5 and 4.6 on page 35. Performing context splitting on the statement `Y2 = ABS( Y2 )`, we first perform loop splitting on each dimension. For the first dimension, the iteration space is divided into the sets {1:2} and {3:5}, while the second dimension produces the sets {1:4} and {5}. After splitting the loops we use loop distribution, which produces these sets of two-dimensional iteration spaces: {1:2,1:4}, {3:5,1:4}, {1:2,5}, and {3:5,5}. The result is the code shown in Figure 7.16, which sets the context four times compared to the 25 times of the naive subgrid loop presented in Section 4.8.3:

7.4 Code Generation Issues

A close evaluation of the context splitting optimization reveals two possible concerns: loop overhead and code growth. Both of these occur because context splitting takes a single subgrid loop nest and generates multiple loop nests (with reduced loop bounds) each with a copy of the loop body (minus context setting code). We will address each of these concerns separately.

```
Set_Context (.TRUE.)
DO J = 1, 4
   DO I = 1, 2
     Y2'(I,J) = ABS ( Y2'(I,J) )
   ENDDO
ENDDO
Set_Context (iproc1 < 4)
DO J = 1, 4
   DO I = 3, 5
     Y2'(I,J) = ABS ( Y2'(I,J) )
   ENDDO
ENDDO
Set_Context (iproc2 < 4)
DO I = 1, 2
   Y2'(I,5) = ABS ( Y2'(I,5) )
ENDDO
Set_Context (iproc1 < 4 .AND. iproc2 < 4)
DO I = 3, 5
   Y2'(I,5) = ABS ( Y2'(I,5) )
ENDDO
```

Figure 7.16 Context splitting a multidimensional distribution.

The additional loop overhead generated by context splitting is not a problem. For SIMD machines, recall that all the control flow operations related to the looping constructs are executed on the FE processor. Since the FE processor is usually much faster than the PE processors, and is executing asynchronously from them, it is able to handle the extra loop overhead while still keeping the PE array busy. In essence, we have increased the workload executed on the FE, but this has allowed us to decrease the workload sent to the PE array.

Since context splitting produces several copies of the loop body for each loop level which is split, code growth is exponential in the number of nested subgrid loops which are split. If this growth is a concern, we have two alternatives that can be used to address it. First, the loop body could be encapsulated as an internal subroutine, which is branched to and returned from. Since the subroutine is internal, the interface simply requires that the return address be saved. Alternatively, by limiting context splitting to only the innermost one or two subgrid loop levels, one can keep code growth bounded by a linear amount. Our experiments have shown that this small limitation still retains most of the performance gains achieved when splitting all subgrid loop levels.

Throughout this presentation we have ignored the context setting required for masked assignment instructions, such as that generated by the `WHERE` statement. The semantics of masked instructions imply that each element has its own execution mask. This means that the iteration space cannot be divided into convenient sets each with a constant execution mask. For this reason context splitting cannot be used to hoist the calls to `Set_Context` out of such subgrid loops. Context partitioning is still applicable and can be used to generate a single subgrid loop for groups of congruent statements.

7.5 Experimental Results

To verify the effectiveness of these optimizations, we performed experiments on both SIMD and MIMD machines. In this section we report the results from the SIMD experiments executed on a DECmpp 12000, which is equivalent to a MasPar MP-1. In Section 9.8 we report the results of performing context partitioning on a stencil code executing on an IBM SP-2, a MIMD machine.

For our SIMD experiments, we performed the optimizations by hand on a section of code taken from a Fortran90 version of the ARPS weather prediction code [66].

The code initializes 16 two-dimensional arrays. We chose this section of code since context partitioning would not benefit additionally from data reuse nor would it be penalized for generating excessive register pressure. Thus all performance improvements are directly attributable to the elimination of redundant context changes and the reduction of loop overhead.

We generated five versions of this code segment and timed each on a dedicated DECmpp 12000. The first version was simply the Fortran90 segment as taken from the ARPS program. This was compiled with MP Fortran Version 2.1. The second version was a translation of the Fortran code into MPL. We optimized this version by performing the following optimizations by hand: common subexpression elimination, strength reduction, and loop-invariant code motion. We also used `register` declarations on array pointers. We then took this MPL version and generated three new versions by applying our context optimizations; one version for each of the optimizations, and one version which combined both optimizations. All the MPL routines were compiled by Version 3.0 of the MPL compiler. All five versions used a (CYCLIC,CYCLIC) distribution, the default distribution of the MP Fortran compiler.

The version combining context partitioning and splitting reduced the execution time by 45% when compared to the original MPL code (which itself reduced the execution time by approximately 10% compared to the Fortran code). See Figure 7.17

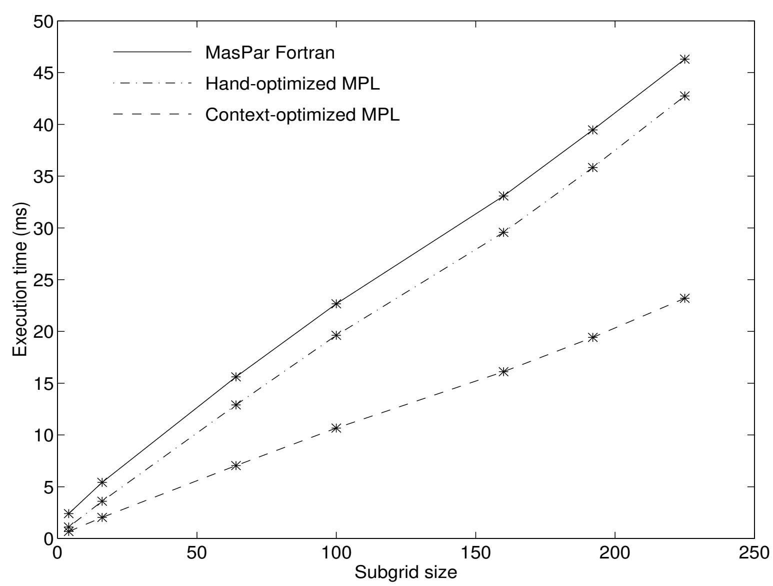


Figure 7.17 Time for ARPS weather code to initialize sixteen 2-D arrays.

for a comparison of execution time versus subgrid size for these three versions of the code. Individually, context partitioning and context splitting reduced the execution time by 35% and 45%, respectively. The reason that the combination of the two optimizations did not out-perform context splitting is that, once splitting eliminated the costly context setting code from the subgrid loops, the loops became memory bound. For subgrid loops that are more computationally intensive, we expect these two optimizations to have an additive effect, although the total effect may be less than the improvement experienced with this code.

To consider something more computationally interesting, we looked at a five-point difference computation:

```
RESULT = (A + CSHIFT(A,1,1) + CSHIFT(A,-1,1)
       + CSHIFT(A,1,2) + CSHIFT(A,-1,2))/5
```

`RESULT` and `A` were both two-dimensional arrays distributed in a (CYCLIC,CYCLIC) manner. We generated three versions of the code: a Fortran90 version, a hand-optimized MPL version, and an MPL version which had context splitting applied (since there was only a single statement, context partitioning was not applicable). We then timed the subgrid loops. In all cases, the communication time to set up the computation was excluded from the measurements. The results are shown in Figure 7.18.

Since the time to compute and set the context is a smaller portion of the total work performed in this subgrid loop, the performance gain is not as impressive as that obtained on the array initialization code. But the 13% reduction in the execution time from the hand-optimized MPL version is still significant. The hand-optimized MPL had already reduced the execution time by 32% when compared to the Fortran version.

As a final point of interest, we took the above five-point difference computation and performed context splitting only on the innermost subgrid loop as discussed in Section 7.4. Code growth was minimal, adding only two statements to the MPL code: a call to `Set_Context` and a replication of the loop body (a single assignment statement). In comparison, the original split version, in which context splitting was applied to both loops in the subgrid loop nest, slightly more than doubled the amount of code. Additionally, the performance difference between the two split versions was minimal. The new split version reduced the execution time of the hand-optimized MPL version by 12%, compared to the 13% reduction of the original split version.

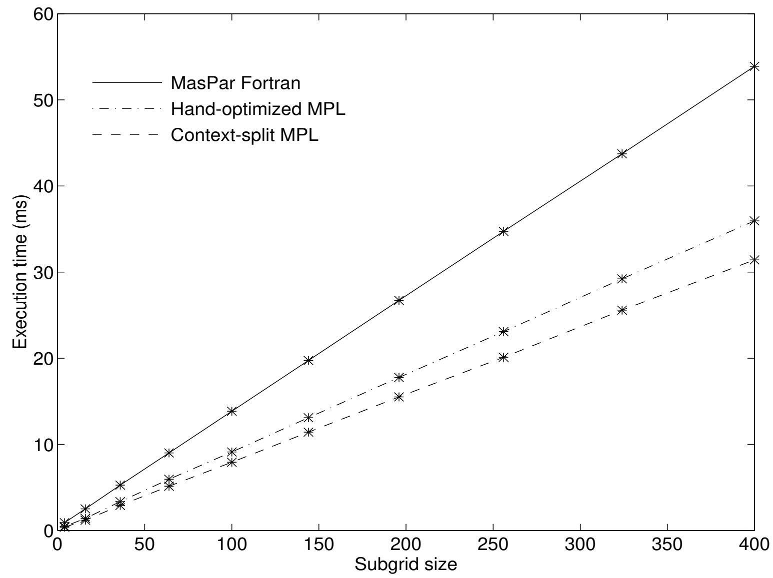


Figure 7.18 Time for 5-point difference computation.

7.6 Related Work

Work at Compass by Albert, *et al.*, describes the generation and optimization of context setting code [7]. They avoid redundant context computations when adjacent statements operate under the same context. They also perform classical optimizations on the context expressions, such as common subexpression elimination. They mention the possibility of reordering computations to minimize context changes, but they do not discuss such transformations.

While giving some optimization hints for the Slicewise CM Fortran compiler, Sabot describes the need for code motion to increase the size of elemental code blocks (blocks of code for which a single subgrid loop can be generated) [142]. He goes on to state that the compiler does not perform this code motion on user code, and thus it is up to the programmer to make them as large as possible. In a later paper describing the internals of the compiler, he describes how it attempts to perform code motion so that subgrid loops may become adjacent and thus fused [140]. However, the code motion performed is limited to only moving scalar code from between subgrid loops, not in moving the loops themselves. Furthermore, due to the limited dependence analysis performed by the compiler, only compiler-generated scalar code is moved. It was this work that motivated us to investigate the context partitioning problem.

Chen and Cowie also recognize the need to fuse parallel loops in their Fortran90 compiler [57]. However, they only fuse adjacent loops and perform no code motion to increase the chances of fusion.

7.7 Summary

We have developed a double-edged sword to combat the cost of context switching in codes for SIMD machines. The first edge of the sword reduces the number of subgrid loops which operate over the same context to a minimum. The second edge reduces the number of context changes per subgrid loop from $O(N)$ to $O(1)$ for unmasked array assignment statements. We have also discussed how they may be used to optimize code for MIMD machines as well.

Chapter 8

Advanced Scalarization of Array Statements

8.1 Introduction

As introduced in Section 4.8, an array statement must be rewritten so that it accesses smaller chunks of data before it can be executed on the target architecture. The size of the chunks must "fit" the hardware of the target machine. For scalar machines, or individual PEs of a parallel machine, that size is a single array element; for vector machines that size is the size of the vector registers. This translation is known as *scalarization* when discussed in terms of a scalar machine, and is called *sectioning* on vector architectures. In this chapter we address compiling to a scalar machine. The material is equally applicable to vector machines.

8.2 Two-Pass Scalarization

Due to the semantics of array statements, their translation into correct serial code is not always trivial, as can be seen in Figure 8.1. The code in Figure 8.1(b), the result of a naive scalarization, is not equivalent to its Fortran90 counterpart since on the second and subsequent iterations of the I loop the reference $A(I-1)$ accesses the new value of the array A assigned on the previous iteration. This violates the "load-before-store" semantics of the Fortran90 array assignment statement.

Fortunately, data dependence information can tell us when the scalarized loop is correct. Allen and Kennedy [14] have shown that a scalarized loop is correct if and

```
A(2:N) = A(1:N-1) + B(1:N-1)
```

(a) array statement

```
DO I=2, N
  A(I) = A(I-1) + B(I-1)
END DO
```

(b) naively scalarized code

Figure 8.1 Invalid scalarization example.

only if it does not carry a true dependence. Using this fact, most compilers perform scalarization in the following manner:

1. Perform a naive scalarization of the array statement into a scalar loop nest.

2. Compute the data dependences of the resulting code.

3. While a scalarized loop carries a true dependence, perform code transformations to either eliminate the dependence or change it into an antidependence.

The code transformations that can be applied to handle the loop carried true dependences include loop reversal, loop interchange, prefetching, and as a last resort the generation of array temporaries. The interested reader is referred to Allen and Kennedy [14] for a complete discussion.

The Allen & Kennedy algorithm requires two passes over the code, one to perform the naive scalarization and another to perform code transformations to restore the semantics of the program if the initial scalarization is invalid. Using the dependence information produced by the methods described in Chapter 5, we propose a new algorithm that eliminates the need for the first pass and is able to determine a valid scalarization before attempting any transformations.

8.3 One-Pass Scalarization

Our new scalarization algorithm begins by performing dependence analysis directly on Fortran90 array statements [138]. When attempting to scalarize an array statement, we only need to be concerned with the *scalarization dependences* of that statement on itself. As discussed in Section 5.3.4, such dependences are always antidependences and may contain any of the three direction specifiers in triplet positions. If we perform naive scalarization on a triplet that has a forward ($<$) or loop independent ($=$) antidependence, the resulting loop has an equivalent dependence. However, if we naively scalarize a triplet that carries a backward ($>$) antidependence, the resulting loop carries a forward true dependence indicating an incorrect scalarization. Thus we must be careful to address the antidependences that contain an "$>$" in the position corresponding to a triplet we are scalarizing.

Our algorithm proceeds to scalarize the statement one triplet at a time, paying particular attention to those triplets that carry backward antidependences. There are several methods we can use to handle these dependences; basically the same methods that the two-pass algorithm uses to address loop carried true dependences.

First we can choose the order in which the triplets are scalarized. If we choose a triplet position that contains only "$<$" and "$=$" elements in the scalarization dependences, we can perform a naive scalarization of that triplet and know that it is correct. Afterward we can eliminate from further consideration those dependences which contained an "$<$" in that position, since those dependences are carried by the scalarized loop. This is advantageous when the eliminated dependences contained non-"$=$" elements in other positions.

Second, if all dependences contain either a "$>$" or "$=$" in a given position, the corresponding triplet can be correctly scalarized with a reversed loop. Again, those dependences that were carried at that triplet position can be eliminated. Failing these, we can continue to attempt all the transformations that the two-pass algorithm utilized, including prefetching.

If there are triplets remaining that cannot be scalarized by any of the transformations, we generate a temporary array whose size equals the remaining array section. We then create two adjacent loop nests for the remaining triplets. The first nest performs the desired computation and stores it in the temporary array, and the second copies the results from the temporary array into the destination array.

As an example, consider the statement in Figure 8.2(a) and its corresponding scalarization dependences in Figure 8.2(b). After scanning the dependences, we see that the second triplet can safely be scalarized using the naive method. This eliminates the first two dependences from further consideration since they both contain an "$<$" in the position corresponding to the second triplet. That leaves us with a single dependence of $(>, =)$ and only the first triplet to scalarize. The direction vector quickly tells us that the remaining triplet can safely be scalarized by generating a reversed loop. The resulting code is shown in Figure 8.3.

```
A(2:N-1,2:N-1) = A(1:N-2,3:N)           (>,<)
               + A(3:N,3:N)             (<,<)
               + A(1:N-2,2:N-1)         (>,=)

(a) array statement                 (b) dependence vectors
```

Figure 8.2 One-pass scalarization example.

```
DO J = 2, N-1
  DO I = N-1, 2, -1
    A(I,J) = A(I-1,J+1)
           + A(I+1,J+1)
           + A(I-1,J)
  END DO
END DO
```

Figure 8.3 Generated scalar code.

8.4 Scalarization of Offset Arrays

Recall from Chapter 6 that our compilation model includes the creation of offset arrays to eliminate the intraprocessor copying of data associated with SHIFT intrinsics. The presence of such arrays necessitates that the rest of the compiler be modified as appropriate to handle them. In particular, dependence analysis needs to produce the proper dependence vectors for them, and scalarization must insure the production of correct code.

Dependence analysis can easily be modified to handle offset arrays by including the offset annotation, as described in Section 6.3.2, in the analysis of the array subscripts. In this way, subscript analysis would treat the offset array reference $A^{<-1,+1>}$(1:N,1:N) simply as the array reference A(1-1:N-1,1+1:N+1).

Once dependence analysis has determined the dependences arising from both normal and offset array references, scalarization proceeds as described previously. When generating scalar subscript references for offset arrays, we need to alter the subscript expressions. This is accomplished by incrementing each subscript by the corresponding annotation entry. This was described in Section 6.3.2 and an example was given in Figure 6.5.

As a final note, we would like to once again address the issue of array copies. In Section 6.5.3 we proved a theorem that showed that an array copy statement inserted by our offset array algorithm could never execute more often than the shift operations it optimized. However, we also noted in Section 6.6.1 that the use of offset arrays invalidates the perfect alignment property that is a part of our compilation model (see Section 4.8.1). Due to this, the naive scalarization of array assignment statements that reference offset arrays may no longer be valid and array temporaries may be required.

```
 CALL overlap_cshift(a,-1,1)
 CALL overlap_cshift(a,+1,1)
 CALL overlap_cshift(a,-1,2)
 CALL overlap_cshift(a,+1,2)
 a = cc * a
&  + cn * a<-1,0>
&  + cs * a<+1,0>
&  + cw * a<0,-1>
&  + ce * a<0,+1>
```

Figure 8.4 5-point stencil computation using offset arrays that requires a scalarization temporary.

As an example, consider the code fragment in Figure 8.4. In this example we have a 5-point stencil computation which references four offset arrays. The four offset arrays are based on the same array which is being defined in the statement. Due to the scalarization dependences that exists for this statement $((>,=),(<,=),(=,>),(=,<))$, proper scalarization requires a temporary array. This temporary array first gets assigned the results of the computation and is then copied back into the array A.

This leads us to the following question: Can the combination of copies inserted by the offset array algorithm and copies inserted for scalarization temporaries due to offset arrays cause more data movement than specified in the original program? The following two theorems prove that this cannot happen. The first theorem states that the generation of an offset array can lead to either a copy statement being inserted by the offset array algorithm or a copy statement being generated for a scalarization temporary, but never both. The second theorem states that if a scalarization temporary is needed, it cannot be more deeply nested than the shift statement that generated the offset array.

> **Theorem 8.1** An offset array can never cause a copy statement to be generated by both the offset array algorithm and the scalarization algorithm.

Proof The proof is in two parts. Each part states that the conditions that must exist for one of the algorithms to insert a copy statement end the live range of the offset array.

1. The offset array algorithm inserts an array copy statement when it can no longer propagate the offset array through the SSA graph. This can occur either at an

UPDATE operation or a ϕ-function. In each case, the copy statement marks the end of the offset array's live range – the algorithm was not able to propagate the offset array past that point. Since the offset array was not propagated past that point there can be no subsequent references to it. Thus its live range ends there. Therefore the offset array could not possibly lead to temporaries being generated by the scalarization algorithm.

2. For the scalarization algorithm to generate an array temporary due to an offset array, the offset array must be based on the same array which is being defined on the left-hand side of the assignment statement. See Figure 8.4 for an example. This definition constitutes a modification of the source of the offset array. Since it would be a violation of criterion 1 to propagate the offset array past this point, there can be no subsequent references to the offset array. Thus the live range of the offset array ends at that statement.

Since the live range of the offset array ends at the point of the copy, there are no further possibilities for subsequent copy statements to be inserted by either the offset array algorithm or the scalarization algorithm due to the use of the offset array. □

Theorem 8.2 An array statement which requires a scalarization temporary due to an offset array cannot be more deeply nested than the shift operation that created the offset array.

Proof As stated in the previous proof, if the scalarization algorithm must generate an array temporary for an assignment statement referencing an offset array, then the offset array must be based on the same array which is being defined on the left-hand side of the assignment statement. This definition requires that a ϕ-function exist for the base array in any enclosing loops. Using these facts, the rest of the proof follows the same lines as Lemma 6.2 and is thus omitted. □

Chapter 9

Putting It All Together – Stencil Compilation

9.1 Introduction

In many programs performing dense matrix computations, the main computational portion of the program belongs to a class of kernels known as stencils. For Fortran90D and HPF to gain acceptance as a vehicle for parallel scientific programming, they must achieve high performance on this class of problems. This chapter describes a strategy for optimizing such stencil computations for execution on distributed-memory multiprocessors. The strategy orchestrates many of the optimizations previously presented in this dissertation, and introduces a new optimization that enhances the use of offset arrays. The optimizations presented target the overhead of data movement that occurs between processors, within the local memory of the processors, and between the memory and registers of the processors. We focus on the application of this strategy on distributed-memory architectures, although it is more broadly applicable.

In the next section we introduce stencil computations and discuss their execution cost on distributed-memory machines. In Section 9.3 we give an overview of our compilation strategy, and then address the individual optimizations in detail in Sections 9.4, 9.5, and 9.6. We then trace through an extended example to show how the optimizations work in concert. In Section 9.8 we give some experimental results, and in Section 9.9 we compare this strategy with other known efforts.

9.2 Stencil Computations

In this section we introduce stencil computations and discuss their execution cost on distributed-memory machines.

9.2.1 Stencils

A *stencil* is a stylized matrix computation in which a group of neighboring data elements are combined to calculate a new value. They are typically combined in the form of a sum of products. This type of computation is common in solving partial differential equations, image processing, and geometric modeling. The Fortran90 array assignment statement in Figure 9.1 is commonly referred to as a 5-point stencil. In this statement SRC and DST are arrays, and C1–C5 are either scalars or arrays. Each interior element of the result array DST is computed from the corresponding element of the source array SRC and the neighboring elements of SRC on the North, West, South, and East. A 9-point stencil that computes all grid elements by exploiting the CSHIFT intrinsic might be specified as shown in Figure 9.2.

Stencils are not necessarily symmetrical as in the two previous examples. Consider the stencil computation shown in Figure 9.3 which has a pattern in the shape of a capital "L".

In the previous three examples the stencils were specified as a single array assignment statement, but this need not always be the case. Consider again the 9-point

```
 DST(2:N-1,2:N-1) = C1 * SRC(1:N-2,2:N-1)
&                 + C2 * SRC(2:N-1,1:N-2)
&                 + C3 * SRC(2:N-1,2:N-1)
&                 + C4 * SRC(3:N  ,2:N-1)
&                 + C5 * SRC(2:N-1,3:N )
```

Figure 9.1 5-point stencil computation.

```
 DST = C1 * CSHIFT(CSHIFT(SRC,-1,1),-1,2)
&    + C2 * CSHIFT(SRC,-1,1)
&    + C3 * CSHIFT(CSHIFT(SRC,-1,1),+1,2)
&    + C4 * CSHIFT(SRC,-1,2)
&    + C5 * SRC
&    + C6 * CSHIFT(SRC,+1,2)
&    + C7 * CSHIFT(CSHIFT(SRC,+1,1),-1,2)
&    + C8 * CSHIFT(SRC,+1,1)
&    + C9 * CSHIFT(CSHIFT(SRC,+1,1),+1,2)
```

Figure 9.2 9-point stencil computation.

```
 DST = C1 * SRC
&    + C2 * CSHIFT(SRC,-1,1)
&    + C3 * CSHIFT(SRC,-2,1)
&    + C4 * CSHIFT(CSHIFT(SRC,-2,1),+1,2)
```

Figure 9.3 An odd-shaped stencil computation.

stencil above. If the programmer attempted to optimize the program by hand, or if the stencil was pre-processed by other optimization phases of the compiler, we might be presented with the code shown in Figure 9.4[9].

When encountering a set of statements such as these, we would like to be able to produce code equivalent to that produced for the single-statement stencil. Thus we have designed our optimizer to handle the most general input form, which has several distinguishing characteristics:

- CSHIFT intrinsics and temporary arrays have been inserted to perform data movement needed for operations on array sections that have different processor mappings.
- Each CSHIFT intrinsic occurs as a singleton operation on the right-hand side of an array assignment statement and is only applied to whole arrays.
- The expression that actually computes the stencil operates on operands that are perfectly aligned, and thus no communication operations are required.

All stencil and stencil-like computations can be translated into this general form by factoring expressions and introducing temporary arrays. In fact, this is the intermediate form used by several distributed-memory compilers [110, 139, 34]. For example, given the 5-point stencil computation presented in Figure 9.1 above, the

```
RIP = CSHIFT(U,SHIFT=+1,DIM=1)
RIN = CSHIFT(U,SHIFT=-1,DIM=1)
T   = U + RIP + RIN
T   = T + CSHIFT(U,SHIFT=-1,DIM=2)
T   = T + CSHIFT(U,SHIFT=+1,DIM=2)
T   = T + CSHIFT(RIP,SHIFT=-1,DIM=2)
T   = T + CSHIFT(RIP,SHIFT=+1,DIM=2)
T   = T + CSHIFT(RIN,SHIFT=-1,DIM=2)
T   = T + CSHIFT(RIN,SHIFT=+1,DIM=2)
```

Figure 9.4 Problem 9 from the Purdue Set.

[9]This example was taken from Problem 9 of the Purdue Set [135] as adapted for Fortran D benchmarking by Thomas Haupt of NPAC.

CM Fortran compiler would translate it into the sequence of statements shown in Figure 9.5[10].

For the rest of this chapter we assume that all stencil computations have been put into this form, and that all arrays are distributed in a BLOCK fashion. And although we concentrate on stencils expressed using the CSHIFT intrinsic, the techniques presented can be generalized to handle the EOSHIFT intrinsic as well.

9.2.2 Stencil Execution Costs

The cost of a stencil computation on a distributed-memory machine can be analyzed by breaking it down into its two major components: the set of CSHIFT operations and the calculation of the sum of products.

When a CSHIFT operation is performed on a distributed array, two major actions take place:

1. Data elements that must be shifted across processing element (*PE*) boundaries are sent to the neighboring PE. This is the interprocessor component of the shift.

2. Data elements that stay within the memory of the PE must be copied to the appropriate locations in the destination array. This is the intraprocessor component of the shift.

```
 ALLOCATE TMP1, TMP2, TMP3, TMP4
 TMP1 = CSHIFT(SRC,SHIFT=-1,DIM=1)
 TMP2 = CSHIFT(SRC,SHIFT=-1,DIM=2)
 TMP3 = CSHIFT(SRC,SHIFT=+1,DIM=1)
 TMP4 = CSHIFT(SRC,SHIFT=+1,DIM=2)
 DST(2:N-1,2:N-1) = C1 * TMP1(2:N-1,2:N-1)
&                 + C2 * TMP2(2:N-1,2:N-1)
&                 + C3 * SRC (2:N-1,2:N-1)
&                 + C4 * TMP3(2:N-1,2:N-1)
&                 + C5 * TMP4(2:N-1,2:N-1)
 DEALLOCATE TMP1, TMP2, TMP3, TMP4
```

Figure 9.5 Intermediate form of 5-point stencil computation.

[10]For this reason most CM Fortran programmers use CSHIFTs explicitly in their stencil computations. Array-syntax stencils produced the same CSHIFT intrinsic calls but then had the additional overhead of the vector masking operations required for handling the array subsections [140].

Since the costs of these two actions were discussed in Chapter 6, we do not address them further here.

The sum of products is calculated within the subgrid loop nest, which is the result of scalarization and SPMD code generation as discussed in Section 4.8. At this point, specific array elements are referenced rather than full arrays or array sections. If the stencil is not computed over the entire matrix, or if the subgrid size is not known at compile time, each PE must compute its subgrid loop bounds. The subgrid loop for the 5-point stencil presented in Figure 9.5 is shown in Figure 9.6 (where $\$MYPID$ returns the processor id number (one based) for the given dimension, G is the extent of the local subgrid, and N is the extent of the original array).

Calculating the execution cost of such a loop nest is usually accomplished by totalling the number of floating point operations in the loop, dividing that number by the rate the target machine can execute those flops, and then multiplying by the total number of iterations. Unfortunately, due to the large number of array references found in such a loop, this metric is insufficient. To better measure the performance of subgrid loops in relation to their memory accesses we use the notion of balance as defined by Callahan, *et al.* [44].

The *machine balance* (β_M) for a particular machine is defined to be the relationship between the rate at which memory can be accessed compared to the rate that floating-point operations can be executed:

$$\beta_M = \frac{max\ words/cycle}{max\ flops/cycle}$$

```
      LB1 = MAX((($MYPID(1)-1)*G)+1,2) - (($MYPID(1)-1)*G)
      UB1 = MIN($MYPID(1)*G,N-1) - (($MYPID(1)-1)*G)
      LB2 = MAX((($MYPID(2)-1)*G)+1,2) - (($MYPID(2)-1)*G)
      UB2 = MIN($MYPID(2)*G,N-1) - (($MYPID(2)-1)*G)
      DO J = LB2, UB2
        DO I = LB1, UB1
          DST(I,J) = C1 * TMP1(I,J)
     &             + C2 * TMP2(I,J)
     &             + C3 * SRC (I,J)
     &             + C4 * TMP3(I,J)
     &             + C5 * TMP4(I,J)
        ENDDO
      ENDDO
```

Figure 9.6 Subgrid loop for 5-point stencil.

The *loop balance* (β_L) for a given loop is defined as:

$$\beta_L = \frac{number\ of\ memory\ references}{number\ of\ flops}$$

If $\beta_L = \beta_M$, then the loop is balanced for the target machine and will run well. If $\beta_L < \beta_M$, then data can be supplied faster than it can be processed, and the loop is said to be *compute bound*. In this case the machine is running at its peak computational rate. If $\beta_L > \beta_M$, then data can be processed faster then it can be supplied, and there exists idle computational cycles. Such a loop is *memory bound*. For many advanced architectures that offer efficient multiply-add operations, the value of β_L for loops generated from stencils is often larger than β_M, resulting in memory-bound loops. The value of β_L is also significantly increased if the stencil is specified with array-valued coefficients rather than with scalar values, thus exacerbating the problem.

9.3 Compilation Strategy

In this section we give an overview of our compilation strategy. We then present the details of this strategy in the subsequent sections.

Assuming that the program containing the stencil has been put into the form presented at the end of Section 9.2.1, we begin by optimizing the CSHIFT operations. We apply two separate optimization phases: the first addresses the intraprocessor data movement and the second handles the interprocessor data movement.

Intraprocessor data movement is completely eliminating when possible. This is accomplished by exploiting our offset array optimization described in Chapter 6. We then follow this with our context partitioning optimization, as described in Chapter 7, to separate the communication operations from the computational operations.

Once the intraprocessor data movement has been eliminated and we have partitioned the statements into groups of congruent operations, we analyze the resulting interprocessor data movement to eliminate redundant and partially redundant movement. We call this optimization *communication unioning*. The resulting program requires only a single communication operation across each edge[11] of the stencil. This optimization produces only four communication operations for the 9-point stencil example presented in Figure 9.2, even though its original specification required

[11] An edge is determined by the dimension and direction specified in a shift operation.

twelve CSHIFT intrinsics. Similarly, a 27-point 3D stencil, originally specified using 54 CSHIFT intrinsics, would require only 6 communication operations.

Finally, after scalarization has produced a subgrid loop nest, we optimize it by applying a set of loop transformations designed to improve the performance of memory-bound programs. These transformations include unroll-and-jam, which addresses memory references, and loop permutation, which addresses cache references. Each of these optimize the program by exploiting reuse of data values.

9.4 Eliminating Intraprocessor Movement

As briefly mentioned in the preceding section, intraprocessor data movement associated with stencil computations is eliminated by employing our offset array optimization [101]. Since Chapter 6 was entirely devoted to describing this optimization, we do not discuss it further here. However, it is important to note that due to the algorithm's optimistic nature, it is able to eliminate the intraprocessor data movement associated with shift operations in many difficult situations. In particular, it can determine when offset arrays can be exploited even when their definition and uses are separated by program control flow. This allows our stencil compilation strategy to eliminate the intraprocessor data movement in situations that other strategies would not even consider.

After offset arrays have been identified and optimized, we apply our context partitioning algorithm [106]. As explained in Section 7.2, this optimization separates a set of statements into groups of congruent array statements, scalar expressions, and communication operations. This assists the compilation of stencils in two ways. First, by grouping congruent array statements together, we ensure that as subgrid loops are generated, via scalarization and loop fusion, as much computation as possible is placed within each loop. Also, the structure of the subgrid loops is very regular. These increase the chances that loop transformations performed later are successful in exploiting data reuse and data locality. Second, by grouping together the communication operations, we simplify the task of reducing the amount of interprocessor data movement, which is our next topic.

9.5 Reducing Interprocessor Movement

After eliminating the intraprocessor data movement via our offset array optimization, we now focus our attention on the interprocessor data movement that occurs during the calls to OVERLAP_SHIFT. Due to the nature of offset arrays, we are presented with many opportunities to eliminate redundant and partially redundant data movement. We call this optimization *communication unioning*, since it combines a set of communication operations to produce a smaller set of operations.

Before proceeding to discuss our strategy, we need to extend the definition of our OVERLAP_SHIFT routine (see Section 6.3.2 for the original definition). We add an optional fourth argument that takes a regular section descriptor (RSD) [12]. The RSD is used to specify those data elements in the overlap areas of other dimensions are to be transferred along with the specified subgrid elements. This extension allows us to include "corner" elements that are a part of multi-offset arrays. The RSD contains a null specifier "$*$" for the dimension being shifted. The default RSD would contain the range $1 : N$ for all other dimensions. Here's an example of an OVERLAP_SHIFT along the second dimension that carries along the data in the overlap area from the top of the column but not the overlap area from the bottom: `OVERLAP_SHIFT(SRC,+1,2,[0:N,*])`.

There are two key observations that allow communication unioning to find and eliminate redundant interprocessor data movement. First, shift operations, including OVERLAP_SHIFT, are commutative:

`CSHIFT(CSHIFT(SRC,+1,1),-1,2)` $\equiv$ `CSHIFT(CSHIFT(SRC,-1,2),+1,1)`

Thus, for arrays that are shifted more than once, we can order the shift operations in any manner we like without affecting the result. And secondly, since all OVERLAP_SHIFTs move data into the overlap areas of the subgrids, a shift of a large amount in a given direction and dimension may subsume all shifts of smaller amounts in the same direction and dimension. More formally, an OVERLAP_SHIFT of amount i in dimension k is redundant if there exists an OVERLAP_SHIFT of amount j in dimension k such that $|j| \geq |i|$ and $sign(j) = sign(i)$. Given these two points, we proceed to eliminate redundant data movement in the following manner. Note that since we have already applied our context partitioning optimization to the program, we can restrict our focus to the individual groups of calls to OVERLAP_SHIFT.

In communication unioning, we first use the commutative property to rewrite all the shifts for multi-offset arrays such that the OVERLAP_SHIFTs for the lower dimensions occur first and are used as input to the OVERLAP_SHIFTs for higher dimensions. We then reorder all the calls to OVERLAP_SHIFT, sorting them by the shifted dimension, lowest to highest. We now scan the OVERLAP_SHIFTs for the lowest dimension and keep only the largest shift amount in each direction. All others can be eliminated as redundant.

Communication unioning then proceeds to process the OVERLAP_SHIFTs for each higher dimension in ascending order by performing the following three actions:

1. First we scan the OVERLAP_SHIFTs for the given dimension to determine the largest shift amount in each direction.

2. Then we look for source arrays that are already offset arrays, indicating a multi-offset array. For these, we use the annotations associated with the source array to create an RSD to be used as the fourth argument in the call to OVERLAP_SHIFT. Mapping the annotations to the RSD is simply a matter of adding the annotations to the corresponding RSD dimension; the annotation is added to the lower bound of the RSD if the shift amount is negative, otherwise it is added to the upper bound. As with shift amounts, larger RSDs subsume smaller RSDs.

3. Finally, we generate a single OVERLAP_SHIFT in each direction, using the largest shift amount and including the RSD as needed – all other OVERLAP_SHIFTs for that dimension can be eliminated.

This eliminates all communication for an offset array, except for a single message in each direction of each dimension. The number of messages for the offset array is thus minimized.

As an example, consider again the 9-point stencil computation that we presented in Figure 9.2. The original stencil specification required twelve CSHIFT intrinsics. After applying the above transformations, only the following four calls are required:

```
CALL OVERLAP_SHIFT(SRC,-1,1)
CALL OVERLAP_SHIFT(SRC,+1,1)
CALL OVERLAP_SHIFT(SRC,-1,2,[0:N+1,*])
CALL OVERLAP_SHIFT(SRC,+1,2,[0:N+1,*])
```

Figures 9.7–9.10 display the data movement that results from these calls. The figures contain a 5×5 subgrid (solid lines) surrounded by its overlap area (dashed lines). Portions of the adjacent subgrids are also shown. Figure 9.7 depicts the data movement specified by the first two calls. The result of that data movement is shown in Figure 9.8, where the overlap areas have been properly filled in. The data movement of the last two calls is shown in Figure 9.9. Notice how the last two calls pick up data from the overlap areas that were filled in by the first two calls, and thus they populate all overlap area elements needed for the subsequent computation, as shown in Figure 9.10.

As another example consider a 27-point 3D stencil, which is similar to a 9-point stencil but adds a third dimension. Such a stencil requires 54 CSHIFT intrinsics. After communication unioning there would only exist six communication operations: one across each face of the stencil.

At this point we have generated the minimal number of messages for an offset array. However, there may be additional opportunities for optimizing the interprocessor data movement. If the program is computing multiple stencils or if the stencil being computed uses multiple source arrays, there may be several offset arrays with similar communication patterns. Since we have already ordered the calls to OVERLAP_SHIFT by dimension, it is a trivial task to use message aggregation to join into a single message all the data that must be moved across a given dimension for all the different offset arrays.

Additionally, since we have ordered the calls to OVERLAP_SHIFT by dimension, there are never any data dependences that exist between two calls that go across the same dimension but in opposite directions. Thus they can be performed in parallel if the target architecture supports bi-directional communication. For example, the two OVERLAP_SHIFTs depicted in Figure 9.7 can be performed simultaneously if supported by the hardware.

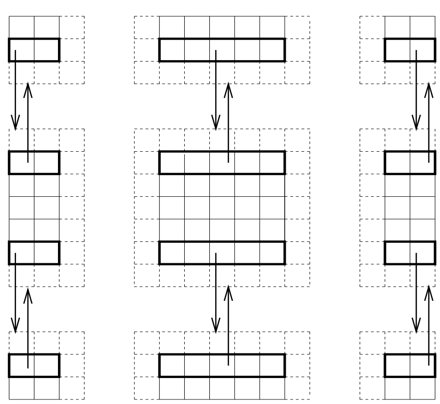


Figure 9.7 First half of 9-point stencil communication

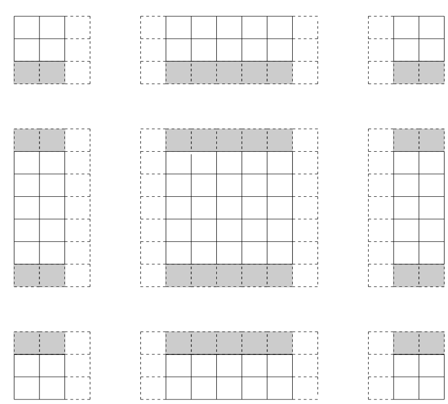


Figure 9.8 Result of communication operations

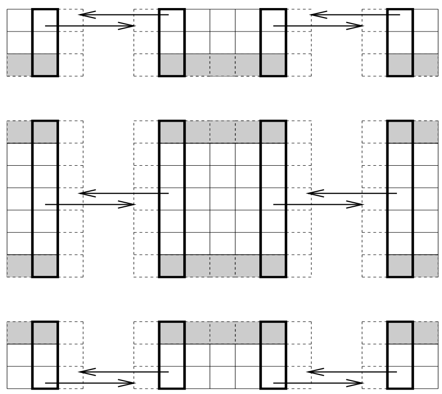


Figure 9.9 Second half of 9-point stencil communication

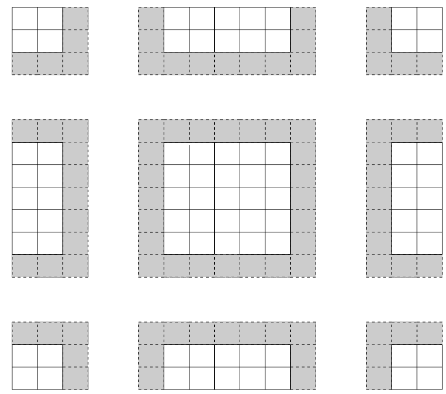


Figure 9.10 Result of communication operations

9.6 Optimizing the Computation

Once the offset array and communication unioning optimizations have been completed, we must optimize the performance of the stencil computation on each node. Our strategy involves the following compiler optimizations to improve data locality:

1. Improve the order of memory accesses through loop permutation [47].

2. Improve loop balance through *unroll-and-jam* and *scalar replacement* [43, 46].

Note that strip-mine-and-interchange can be included here [158]. We have omitted it because of its relative instability and the large amount of cache reuse that already exists in stencil computations [63, 118]. In the rest of this section we give an overview of loop permutation, unroll-and-jam and scalar replacement. More information can be found elsewhere [125].

9.6.1 Loop Permutation

Not all loops exhibit good cache locality, resulting in idle computational cycles while waiting for main memory to return data. For example, in the loop,

```
DO I = 1, N
  DO J = 1, N
    B(I,J) = A(I,J) + A(I+1,J)
  ENDDO
ENDDO
```

references to successive elements of `B` and `A` are a long distance apart in number of memory accesses (this assumes Fortran's column-major storage). Most likely, current cache architectures would not be able to capture the potential cache-line reuse available because of the volume of data accessed between reuse points. With each reference to `B(I,J)` and `A(I+1,J)` causing a cache miss, the loop would spend a majority of its time waiting on main memory. However, if we interchange the `I`- and `J`-loops to get

```
DO J = 1, N
  DO I = 1, N
    B(I,J) = A(I,J)+ A(I+1,J)
  ENDDO
ENDDO
```

the references to successive elements of `B(I,J)` and `A(I+1,J)` immediately follow one another. In this case, we have attained locality of reference for `B(I,J)` and `A(I+1,J)` by moving reuse points closer together. The result is fewer idle cycles waiting on main memory. For a more complete discussion of loop permutation see Wolf and Lam [158], Kennedy and McKinley [102] and Carr, *et al.* [47].

9.6.2 Scalar Replacement

Even with better cache performance through loop permutation, a loop may still not perform as well as possible. If a loop is memory bound, then its balance must be lowered. Balance can be lowered by reducing the number of memory references in a loop, replacing references to arrays with sequences of scalar variables. In the code shown below,

```
DO 10 J = 1, N
  DO 10 I = 1, N
    B(I,J) = A(I,J)+ A(I+1,J)
  ENDDO
ENDDO
```

the value accessed by `A(I,J)` is defined on the previous iteration of the loop by `A(I+1,J)` on all but the first iteration. Using this knowledge, the flow of values between the references can be expressed with scalar temporaries as follows.

```
DO 10 J = 1, N
  T1 = A(1,J)
  DO 10 I = 1, N
```

```
      T0 = A(I+1,J)
      B(I,J) = T0 + T1
      T1 = T0
    ENDDO
  ENDDO
```

Since the values held in scalar quantities will probably be in registers, the load of A(I,J) has been removed, resulting in a reduction in the memory cycle requirements of the loop (the register copy, T1 = T0, can be removed by unrolling I) [48]. This transformation is called *scalar replacement* and is described in detail elsewhere [43].

9.6.3 Unroll-And-Jam

Unroll-and-jam is a transformation that can be used in conjunction with scalar replacement to improve the performance of many memory-bound loops [6, 10, 44]. The transformation unrolls an outer loop and then fuses the resulting inner loops back together. Using unroll-and-jam, more computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, the loop

```
  DO 10 J = 1, 2*N
    DO 10 I = 1, N
      B(I,J) = A(I,J)+ A(I,J+1)
    ENDDO
  ENDDO
```

after unroll-and-jam of I by a factor of 1 becomes

```
  DO 10 J = 1, 2*N, 2
    DO 10 I = 1, N
      B(I,J) = A(I,J)+ A(I,J+1)
      B(I,J+1) = A(I,J+1)+ A(I,J+2)
    ENDDO
  ENDDO
```

In the original loop, one floating-point operation and three memory references are left after scalar replacement, giving a balance of 3. After applying unroll-and-jam, two floating-point operations and five memory references exist in the loop, giving a balance of 2.5 (the second reference to A(I,J+1) can be scalar replaced). If the original loop were memory bound, the unroll-and-jammed loop would perform better, since it has a lower balance.

Carr and Kennedy describe an automatic method for applying unroll-and-jam. Their method computes the unroll amount for a loop that best balances the nest with respect to a target architecture while limiting register pressure. For a detailed discussion of this method, see their paper [46].

9.7 An Extended Example

In this section, we trace our compilation strategy through an extended example. This detailed examination shows how our strategy is able to produce code that matches or beats hand-optimized code. It will also demonstrate how we are able to handle stencil computations that cause other methods to fail.

For this exercise, we have chosen to use Problem 9 of the Purdue Set [135], as adapted for Fortran D benchmarking by Thomas Haupt of NPAC [128, 86]. The program kernel is shown in Figure 9.11. The arrays `T`, `U`, `RIP`, and `RIN` are all two-dimensional and have been distributed in a (BLOCK,BLOCK) fashion.

This kernel computes a standard 9-point stencil, identical to that computed by the single-statement stencil shown in Figure 9.2. The reason it has been written in this fashion is to reduce memory requirements. Recall from Section 6.2 that the semantics of CSHIFT state that a copy of the source array is returned with the specified dimension shifted the given number of times. Almost all Fortran90 compilers perform this operation by making a copy of the source array and storing it into a compiler-generated temporary array. Thus for the 9-point stencil shown in Figure 9.2, 12 temporary arrays are created to compute the single statement. This greatly restricts the size of the problem that can be solved on a given machine.

```
RIP = CSHIFT(U,SHIFT=+1,DIM=1)
RIN = CSHIFT(U,SHIFT=-1,DIM=1)
T   = U + RIP + RIN
T   = T + CSHIFT(U,SHIFT=-1,DIM=2)
T   = T + CSHIFT(U,SHIFT=+1,DIM=2)
T   = T + CSHIFT(RIP,SHIFT=-1,DIM=2)
T   = T + CSHIFT(RIP,SHIFT=+1,DIM=2)
T   = T + CSHIFT(RIN,SHIFT=-1,DIM=2)
T   = T + CSHIFT(RIN,SHIFT=+1,DIM=2)
```

Figure 9.11 Problem 9 from the Purdue Set.

In contrast, the 9-point stencil written in Figure 9.11 requires only three copies of array `U` at any one time: the two copies that are in `RIP` and `RIN` plus an additional compiler-generated temporary that can be shared among all the subsequent statements. This reduces the temporary storage requirements by a factor of four! Additionally, the assignments of the CSHIFTs into `RIP` and `RIN` perform a common subexpression elimination, removing four duplicate CSHIFTs from the original specification of the stencil.

Figure 9.12 shows a comparison of execution times for the single-statement CSHIFT stencil in Figure 9.2 and the multi-statement stencil in Figure 9.11. The programs were compiled with IBM's xlhpf compiler and executed on an 8 processor SP-2 for varying problem sizes. As can be seen, the single-statement stencil specification exhausted the available memory for the larger problem sizes, even though each PE had 256Mbytes of real RAM. We attribute the difference in execution time mainly to the elimination of the four duplicate CSHIFTs, but the strain that the large number of temporary arrays imposes on the memory system may also be a factor.

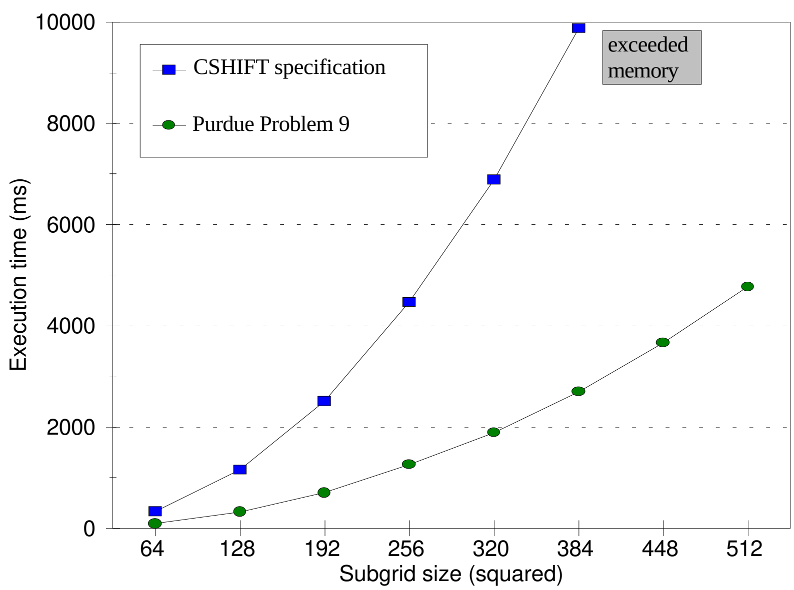


Figure 9.12 Comparison of two 9-point stencil specifications.

9.7.1 Preprocessing

A certain amount of preprocessing is performed by our compilation model as specified in Chapter 4. In particular, the communication generation phase of our compiler separates communication operations from computational operations. This results in temporary arrays being generated for all CSHIFT operations that are a part of larger expressions.

For our example, the six CSHIFTs that are subexpressions in the assignment statements to array `T` are hoisted from the statements and assigned to compiler-generated temporary arrays. Since the live ranges of the temporary arrays do not overlap, a single temporary can be shared among all the statements. Alternatively, each CSHIFT could receive its own temporary array – that would not affect the results of our stencil compilation strategy. The result is shown in Figure 9.13.

9.7.2 Offset Array Optimization

Once all shift operations have been identified and hoisted into their own assignment statements, we apply our offset array optimization. For this example, our algorithm determines that all the shifted arrays can be made into offset arrays. As can be seen in Figure 9.14, all the CSHIFT operations have been changed into OVER-

```
RIP = CSHIFT(U,SHIFT=+1,DIM=1)
RIN = CSHIFT(U,SHIFT=-1,DIM=1)
T   = U + RIP + RIN
ALLOCATE TMP
TMP = CSHIFT(U,SHIFT=-1,DIM=2)
T   = T + TMP
TMP = CSHIFT(U,SHIFT=+1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIP,SHIFT=-1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIP,SHIFT=+1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIN,SHIFT=-1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIN,SHIFT=+1,DIM=2)
T   = T + TMP
DEALLOCATE TMP
```

Figure 9.13 Problem 9 after preprocessing.

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
T = U + U<+1,0> + U<-1,0>
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2)
T = T + U<0,-1>
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2)
T = T + U<0,+1>
CALL OVERLAP_CSHIFT(U<+1,0>,SHIFT=-1,DIM=2)
T = T + U<+1,-1>
CALL OVERLAP_CSHIFT(U<+1,0>,SHIFT=+1,DIM=2)
T = T + U<+1,+1>
CALL OVERLAP_CSHIFT(U<-1,0>,SHIFT=-1,DIM=2)
T = T + U<-1,-1>
CALL OVERLAP_CSHIFT(U<-1,0>,SHIFT=+1,DIM=2)
T = T + U<-1,+1>
```

Figure 9.14 Problem 9 after offset array optimization.

LAP_SHIFT operations, and references to the assigned arrays have been replaced with offset references to the source array U. All intraprocessor data movement has thus been eliminated.

In addition, notice how the temporary arrays, both the compiler-generated TMP array and the user-defined RIP and RIN, are no longer needed to compute the stencil. If there are no other uses of these arrays in the routine, they need not be allocated. This reduction in storage requirements allows for larger problems to be solved on a given machine.

9.7.3 Context Partitioning Optimization

Once we have completed our offset array optimization, we move on to context partitioning. Our algorithm begins by determining the congruence classes present in the section of code. In this example there are only two congruence classes: the array statements, which are all congruent, and the communication statements. The dependence graph is computed next. There are only two types of dependences that exist in the code: true dependences from the OVERLAP_SHIFT operations to the expressions that use the offset arrays, and the true and anti-dependences that exist between the multiple occurrences of the array T. Since all the dependences between the two classes are from statements in the communication class to statements in the congruent array

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2)
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2)
CALL OVERLAP_CSHIFT(U<+1,0>,SHIFT=-1,DIM=2)
CALL OVERLAP_CSHIFT(U<+1,0>,SHIFT=+1,DIM=2)
CALL OVERLAP_CSHIFT(U<-1,0>,SHIFT=-1,DIM=2)
CALL OVERLAP_CSHIFT(U<-1,0>,SHIFT=+1,DIM=2)
T = U + U<+1,0> + U<-1,0>
T = T + U<0,-1>
T = T + U<0,+1>
T = T + U<+1,-1>
T = T + U<+1,+1>
T = T + U<-1,-1>
T = T + U<-1,+1>
```

Figure 9.15 Problem 9 after context partitioning optimization.

class, the context partitioning algorithm is able to partition the statements perfectly into two groups. The result is shown in Figure 9.15.

9.7.4 Communication Unioning Optimization

We now turn our attention to the interprocessor data movement specified in the OVERLAP_SHIFT operations. As directed in Section 9.5, we begin by exploiting their commutative property to rewrite multi-dimensional OVERLAP_SHIFTs so that the lower dimensions are shifted first. No rewriting is necessary for this example since all the dimension 1 shifts occur first, as can be seen in Figure 9.15.

Next we look at the shifts across the first dimension. Since there is only a single shift of distance one in each direction, there is no redundant communication to be eliminated. Moving on to the second dimension we again find only shifts of distance one. However, we do discover four multi-offset arrays. Examining the annotations of the offset arrays, we create RSD's that summarize the overlap areas that are necessary. We generate the two calls to OVERLAP_SHIFT that include the RSD's and then eliminate all other OVERLAP_SHIFT calls for the second dimension. This results in the code shown in Figure 9.16. As can be seen, communication unioning has reduced the amount of communication to a minimum: a single communication operation for each dimension in each direction.

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2,[0:N+1,*])
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2,[0:N+1,*])
```

$\texttt{T = U + U}^{<+1,0>}\texttt{ + U}^{<-1,0>}$
$\texttt{T = T + U}^{<0,-1>}$
$\texttt{T = T + U}^{<0,+1>}$
$\texttt{T = T + U}^{<+1,-1>}$
$\texttt{T = T + U}^{<+1,+1>}$
$\texttt{T = T + U}^{<-1,-1>}$
$\texttt{T = T + U}^{<-1,+1>}$

Figure 9.16 Problem 9 after communication unioning optimization.

9.7.5 Scalarization and Memory Optimizations

Our next task is to scalarize the code and generate an optimized SPMD node program. Using the data dependences calculated for context partitioning, our advance scalarization algorithm (presented in Chapter 8) efficiently scalarizes the code in a single pass over the program source. Since there is no possibility of generating a loop-carried dependence, the scalarized loops are easily fused into a single loop nest. This code is shown in Figure 9.17. To keep the example simple and help the reader follow the flow of the optimizations, we have not refined the loop bounds as would be done to generate a node program that only accesses the subgrids local to each PE.

At this point the main contributions of this dissertation have been completed. We now hand this code over to our node compiler that performs the optimizations outline in Section 9.6. The code is ripe with opportunities for improvements, of which we discuss a few.

We begin by performing loop interchange, if necessary, to make the `i`-loop the inner-most loop. This results in the arrays being accessed in memory order, improving the cache behavior of the program. We use scalar replacement on the references to `T(i,j)` so that the array is only accessed when actually storing the result. The scalar that is used is likely to be kept in a register. See Figure 9.18.

To further improve the loop balance, we now apply the unroll-and-jam transformation. For this example we have chosen an unroll amount of four. The resulting code is shown in Figure 9.19. There are now many duplicate array references that appear on the right-hand side of the assignment statements. In fact, of the 36 references to the array `U`, only 18 of them are unique. This effectively cuts the loop balance (β_L)

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2,[0:N+1,*])
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2,[0:N+1,*])
DO i=1,N
  DO j=1,N
    T(i,j) = U(i,j) + U(i+1,j) + U(i-1,j)
    T(i,j) = T(i,j) + U(i,j-1)
    T(i,j) = T(i,j) + U(i,j+1)
    T(i,j) = T(i,j) + U(i+1,j-1)
    T(i,j) = T(i,j) + U(i+1,j+1)
    T(i,j) = T(i,j) + U(i-1,j-1)
    T(i,j) = T(i,j) + U(i-1,j+1)
  ENDDO
ENDDO
```

Figure 9.17 Problem 9 after scalarization.

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2,[0:N+1,*])
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2,[0:N+1,*])
DO j=1,N
  DO i=1,N
    R0 = U(i,j) + U(i+1,j) + U(i-1,j)
    R0 = R0 + U(i,j-1)
    R0 = R0 + U(i,j+1)
    R0 = R0 + U(i+1,j-1)
    R0 = R0 + U(i+1,j+1)
    R0 = R0 + U(i-1,j-1)
    R0 = R0 + U(i-1,j+1)
    T(i,j) = R0
  ENDDO
ENDDO
```

Figure 9.18 Problem 9 after memory optimizations.

in half. This loop runs significantly faster than the original on machines that have a relatively small machine balance (β_M). For architectures with instruction-level parallelism, the assignments to the different scalars `R0--R3`, which will likely be assigned to registers, can be executed in parallel.

9.8 Experimental Results

To measure the performance boost supplied by each step of our stencil compilation strategy, we ran a set of tests on an 8-processor IBM SP-2. We started by generating a naive translation of the Problem 9 test case into Fortran77+MPI. This is considered our *"original"* version. We then successively applied the transformation as outlined in the preceding subsection and measured the execution time. The results are shown in Figure 9.20.

Before analyzing the results in Figure 9.20, it is worthwhile to compare it to the results shown in Figure 9.12 for the Problem 9 code. Our "original" MPI version of the code has already resulted in an order of magnitude improvement over the code produced by IBM's xlhpf compiler: 0.475 seconds versus 4.77 seconds for the largest problem size.

Starting with our Fortran77+MPI test case, we first applied our offset array optimization to the code, as shown in Figure 9.14. This resulted in a 45% reduction in execution time, equivalent to a speedup of 1.80. Context partitioning was applied next, as shown in Figure 9.15. This optimization allowed scalarization to merge all the computation into a single loop nest, resulting in a 31% reduction in execution time over the previous version. At this point we have reduced the execution time of the original program by 62%, a speedup of 2.64.

We then applied our communication unioning optimization, as shown in Figure 9.16. This results in only four communication operations being performed, and reduces the execution time by 41% when compared to the context-optimized version. Applying the memory optimizations described in Section 9.6 reduced the execution time another 14%. This final version has reduced the execution time of the original program by 81%, equivalent to a speedup of 5.19!

Lest someone think that we have chosen IBM's xlhpf compiler as a straw man, we have collected some additional performance numbers. We generated a third version of a 9-point stencil computation, this one using array syntax similar to the 5-point stencil shown in Figure 9.1. This 9-point stencil computation only computes the

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2,[0:N+1,*])
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2,[0:N+1,*])
DO j=1,N,4
  DO i=1,N
    R0 = U(i,j) + U(i+1,j) + U(i-1,j)
    R0 = R0 + U(i,j-1)
    R0 = R0 + U(i,j+1)
    R0 = R0 + U(i+1,j-1)
    R0 = R0 + U(i+1,j+1)
    R0 = R0 + U(i-1,j-1)
    R0 = R0 + U(i-1,j+1)
    T(i,j) = R0
    R1 = U(i,j+1) + U(i+1,j+1) + U(i-1,j+1)
    R1 = R1 + U(i,j)
    R1 = R1 + U(i,j+2)
    R1 = R1 + U(i+1,j)
    R1 = R1 + U(i+1,j+2)
    R1 = R1 + U(i-1,j)
    R1 = R1 + U(i-1,j+2)
    T(i,j) = R1
    R2 = U(i,j+2) + U(i+1,j+2) + U(i-1,j+2)
    R2 = R2 + U(i,j+1)
    R2 = R2 + U(i,j+3)
    R2 = R2 + U(i+1,j+1)
    R2 = R2 + U(i+1,j+3)
    R2 = R2 + U(i-1,j+1)
    R2 = R2 + U(i-1,j+3)
    T(i,j) = R2
    R3 = U(i,j+3) + U(i+1,j+3) + U(i-1,j+3)
    R3 = R3 + U(i,j+2)
    R3 = R3 + U(i,j+4)
    R3 = R3 + U(i+1,j+2)
    R3 = R3 + U(i+1,j+4)
    R3 = R3 + U(i-1,j+2)
    R3 = R3 + U(i-1,j+4)
    T(i,j) = R3
  ENDDO
ENDDO
```

Figure 9.19 Problem 9 after unroll-and-jam.

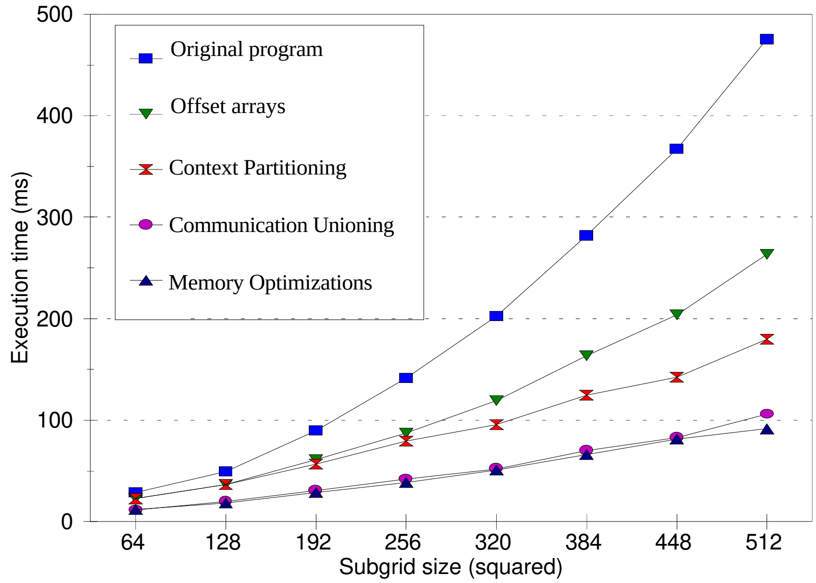


Figure 9.20 Step-wise results from stencil compilation strategy on Problem 9 when executed on an SP-2.

interior elements of the matrix; that is, elements 2:N-1 in each dimension. A graph comparing its execution time to the other two 9-point stencil specifications is given in Figure 9.21. The IBM xlhpf compiler was used in all cases.

It is obvious that the xlhpf compiler is able to minimize the interprocessor communication, prevent the generation of intraprocessor copying, and optimize the computation for the array-syntax stencil. In fact, this specification of the kernel produced performance numbers that tracked our best performance numbers for all problem sizes except the largest, where we had a 10% advantage.

This demonstrates the problem with compiler optimizations based upon pattern matching. When the compiler is able to match a pattern in the user's program, it can produce excellent code. But if the user's program deviates slightly, the compiler cannot match a pattern and the code it produces is mediocre at best. It is important to note that the stencil compilation strategy that we have presented handles all three specifications of the 9-point stencil equally well. That is because our algorithm is based upon the analysis and optimization of the base constructs upon which stencils are built. Our algorithm is designed to handle the lowest common denominator – a form into which our compiler can transform all stencil computations.

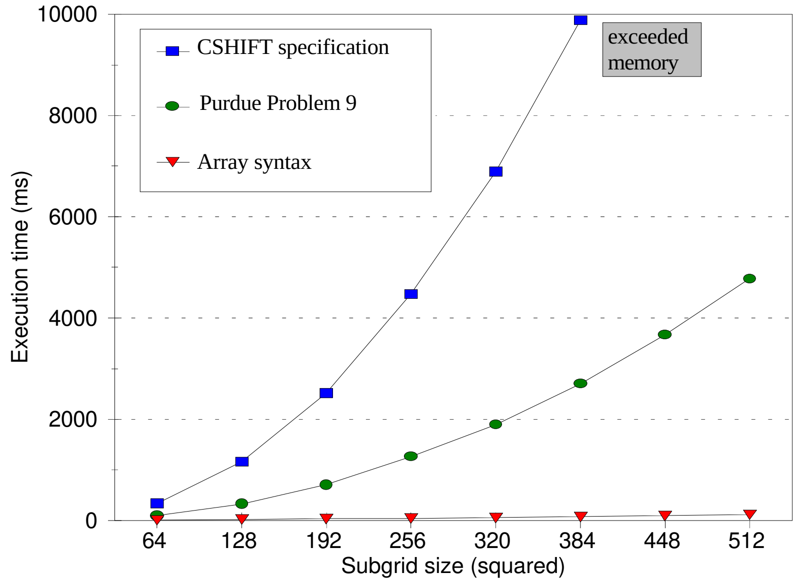


Figure 9.21 Comparison of three 9-point stencil specifications.

9.9 Related Work

One of the first major efforts to specifically address the compilation of stencil computations for a distributed-memory machine was the stencil compiler for the CM-2, also known as the convolution compiler [36, 37, 39]. They eliminated the intraprocessor data movement and optimized the interprocessor data movement by exploiting the CM-2's polyshift communication [72]. The final computation was performed by hand-optimized library microcode that took advantage of several loop transformations and a specialized register allocation scheme.

Our general compilation methodology produces code equivalent to that produced by this specialized compiler. We both eliminate intraprocessor data movement and minimize interprocessor data movement. We also show how our method can exploit bi-directional communication, similar to the parallel communication performed by the polyshift routine. Finally, our use of the unroll-and-jam loop optimizations accomplish the same data reuse as the stencil compiler's "*multi-stencil swath*". It is likely that our "swath" will be smaller than those used by the stencil compiler. This is because we consider machine balance, whereas they consider only the number of available registers. The smaller swath does not suffer a loss of efficiency if we are able to attain the machine balance.

The CM-2 stencil compiler had many limitations however. It could only handle single-statement stencils. The stencil had to be specified using the CSHIFT intrinsic; no array-syntax stencils would be accepted. Since the compiler relied upon pattern matching, the stencil had to be in a very specific form: a sum of terms, each of which is a coefficient multiplying a shift expression. No variations were possible. And finally, the programmer had to recognize the stencil computation, extract it from the program and place it in its own subroutine to be compiled by the stencil compiler.

Our compilation scheme handles a strict superset of patterns handled by the CM-2 stencil compiler. In their own words, they *"avoid the general problem by restricting the domain of applicability."* [39] We have placed no such restrictions upon our work. Our strategy optimizes single-statement stencils, multi-statement stencils, CSHIFT intrinsic stencils, and array-syntax stencils all equally well. And since our optimizations were designed to be incorporated into an F90D/HPF compiler, they benefit those computations that only slightly resemble stencils.

There are also some other commercially available compilers that can handle certain stylized, single-statement stencils. The MasPar Fortran compiler avoids the intraprocessor data movement for single-statement stencils written using array notation. This is accomplished by scalarizing the Fortran90 expression (avoiding the generation of CSHIFTs) and then using dependence analysis to find loop-carried dependences that indicate interprocessor data movement. Only the interprocessor data is moved, and no local copying is required. However, the compiler still performs all the data movement for single-statement stencils written using SHIFT intrinsics. This strategy is shared by many Fortran90/HPF compilers that really only want to handle scalarized code, as discussed in Section 2.4.1. As with the CM-2 stencil compiler, our methodology is a strict superset of this strategy.

Gupta, *et al.* [81], in describing IBM's xlhpf compiler, state that they are able to reduce the number of messages for multi-dimensional shifts by exploiting methods similar to ours. However, they do not describe their algorithm for accomplishing this, and it is unknown whether they would be able to eliminate the redundant communication that arises from shifts over the same dimension and direction but of different distances.

The Portland Group's pghpf compiler, as described by Bozkus, *et al.* [31, 32], performs stencil recognition and optimizes the computation by using OVERLAP_SHIFT communication. They also perform a subset of our communication unioning optimization. However, they are limited to single-statement expressions in both cases.

In general, there have been several different methods for handling specific subclasses of stencil computations. In this paper, we have presented a strategy that encompasses all of them and more.

9.10 Summary

In this chapter, we have presented a general compilation scheme for compiling HPF and Fortran90D stencil computations for distributed-memory architectures. The strategy optimizes such computations by orchestrating a unique set of optimizations. These optimizations eliminate unnecessary intraprocessor data movement resulting from CSHIFT intrinsics, eliminate redundant interprocessor data movement, and optimize memory accesses via loop-level transformations. The optimizations are general enough to be included in a general-purpose Fortran90D/HPF compiler, and they benefit many computations, not just those that fit a stencil pattern. The strength of these optimizations is that they optimize all stencil computation no matter how they were originally specified. It does not matter if array syntax was used or explicit shift intrinsics, or if the stencil was computed by a single statement or across multiple statements. It is the most robust stencil compilation system of which we are aware. Even though we have concentrated on distributed-memory machines in this chapter, the techniques presented are equally applicable to optimizing stencil computations on shared-memory and scalar machines (with the exception of reducing interprocessor movement).

Chapter 10

Conclusions

Currently there are a large number of HPF compiler projects in both industry and academia. Their success has shown that technology exists to correctly translate Fortran90 programs, which have been annotated with data distribution directives, into efficient executables for distributed-memory machines, both MIMD and SIMD. The advanced compilation techniques described in this dissertation demonstrate the additional performance gains attainable when a compiler performs analysis and optimization of programs at the array level. In this chapter we summarize the research presented in this dissertation, give our perspective on compiling the Fortran90 array constructs of HPF, and close by considering areas for future research.

10.1 Compiling HPF

As was discussed early on in this dissertation, the compilation of the array-level constructs of HPF share many of the same characteristics as the compilation of the scalar constructs of HPF. In both cases arrays need to be distributed across the processors of the machine so that parallelism can be effectively exploited. Then the computations specified in the program must be partitioned so that each processor has some amount of work to do. Once data and computation distributions have been finalized, the compiler must insert communication operations to move data so that all operands of an expression reside on the processor which performs the computation. And finally subgrid loops must be generated which iterate over the local subgrids allocated to the individual processors.

However, when it comes to taking advantage of the implicit parallelism present in Fortran90 array constructs, HPF compilers fall into one of two vastly different categories. Each of the two categories has its advantages and its disadvantages. The first category of compilers we have classified as scalarizing compilers. These compilers translate the Fortran90 array constructs into equivalent sequential Fortran77 constructs. The resulting program is then passed to a scalar HPF compiler for optimization and code generation. Such compilers are capable of parallelizing sequential constructs of the program. The other category of compilers we have classified as array operation compilers. Such compilers are characterized by their ability to directly translate the data parallelism found in array constructs for execution on the distributed-memory machine. These compilers usually do an excellent job of handling the explicit parallel constructs found in the program; however they typically do not have the ability to parallelize sequential constructs.

To bridge the chasm between these two categories of HPF compilers, we proposed a new compilation model. The model is a hybrid of the compilation models used by the two categories, and as such we believe that it exploits the advantages of each while minimizing their weaknesses. It accomplishes that by interleaving Fortran90 compilation issues with Fortran77 compilation issues.

We then designed a set of optimizations that are performed at the whole-array level. This is in contrast to most Fortran compilers that only perform optimizations at or below the level of loops or array elements. These array-level optimizations are capable of optimizing dense matrix stencil computations more completely than all other known efforts.

10.2 Contributions

This dissertation make contributions in two main areas: the analysis of Fortran90 array constructs, and the transformation & optimization of Fortran90D/HPF programs at the array level.

10.2.1 Array Analysis Techniques

In this dissertation we presented algorithms for extending data flow analysis, dependence analysis, and SSA form to directly handle Fortran90 array constructs. These extensions provide HPF compilers with analysis capabilities not previously available, allowing them to make decisions and perform transformations at the Fortran90 array level before scalarizing the program into Fortran77 code. In particular, we have shown how to extend dependence analysis to directly handle Fortran90 array-section references, and we have introduced a new class of dependences and showed how they can be used to perform advanced program transformations. An important characteristic of the testing procedures presented is that they were designed to fit smoothly into the framework of existing optimizing compilers.

10.2.2 Optimization Techniques

This dissertation developed a number of novel array-level optimizations for boosting the performance of Fortran90D/HPF programs on distributed-memory machines. The optimizations presented are all based on advanced analysis of Fortran90 array constructs, eliminating the reliance upon pattern matching as is done in many Fortran90 compilers. These optimizations include:

- The offset array optimization which can eliminate the intraprocessor data motion associated with Fortran90 shift operations. This optimization is based on an efficient framework which is global in scope and is capable of optimizing shift operations even when the definitions and uses are separated by control flow.

- Communication unioning optimization which is able to eliminate redundant and partially redundant interprocessor data movement associated with Fortran90 shift operations. Given a group of shift operations, this optimization reduces the number of interprocessor messages to a minimum: one message in each direction for each dimension. This optimization accomplishes its goal by understanding the underlying semantics of the shift operations, as opposed to pattern matching techniques used by other compilers.

- Two different context optimizations which can reduce the cost of context switching code on SIMD machines. The first, context partitioning, reorders the code so that as subgrid loops are generated as many statements as possible that require the same context are placed in the same loop nest. The second, context splitting, splits the iteration space of the subgrid loops into sets that have invariant contexts – this allows us to hoist the context setting code out of the subgrid loops. We have also shown how context partitioning can optimize the code produced for MIMD machines by enhancing the amount of loop fusion possible while preventing loops from being over fused.

- An advanced scalarization algorithm which can scalarize array statements in a single pass over the source code. This algorithm has the same ability to minimize the size of temporary arrays as the standard two-pass scalarization algorithm, but is more efficient.

We have also demonstrated how these optimizations can work in unison to create a powerful stencil compiler. The stencil compiler incorporates the optimizations to target the overhead of data movement that occurs between processors, within the local memory of the processors, and between the memory and registers of the processors. The strength of these optimizations is that they optimize all stencil computation no matter how they were originally specified. It does not matter if array syntax was used or explicit shift intrinsics, or if the stencil was computed by a single statement or across multiple statements. These optimizations make this stencil compiler more robust than all previous efforts in this area.

10.3 Future Work

We conclude by taking a look at areas for continued research in the area of high-level optimizations for Fortran90.

10.3.1 Scalarization and Fusion

In this thesis we presented a methodology for performing dependence analysis directly on array expressions. We classified a new genre of dependences and showed how they could be used to perform scalarization in a single pass over the source code. It is interesting to note that these same scalarization dependences contain all the information needed to determine the validity of loop interchange and loop fusion. It would seem possible, and highly desirable, to develop a single algorithm that would perform all three optimizations at once.

10.3.2 Array Temporaries

When compiling array languages such as Fortran90, a compiler must often generate array temporaries. Such temporaries are required to maintain the semantics of the program. However, the compiler has flexibility in determining where, when, and how many temporaries are generated. There are many conflicting concerns that revolve around the issue of temporary arrays. For example, memory usage verses execution speed. It is sometime possible to reduce total storage requirements by sharing array temporaries; however the sharing of these temporaries will usually prevent the fusion of scalarized loop nests. Such issues must be resolved if we hope to develop Fortran90 compilers that produce code that is both small and efficient.

10.3.3 WHERE Optimization

One Fortran90 array construct not explored in this dissertation is the WHERE statement. In the future we plan to explore the application of our context partitioning optimization and advance scalarization algorithm on code containing WHERE statements and WHERE blocks.

10.3.4 Irregular and Spare Computations

The optimizations presented in this dissertation do an excellent job of optimizing dense matrix stencil computations. Unfortunately, they have little applicability to programs performing irregular or sparse computations. In the future we would like to see if we can extend our success of performing high-level analysis and optimizations into this realm.

Bibliography

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[2] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, NY, 1992.

[3] V. Adve, J. Mellor-Crummey, and A. Sethi. HPF analysis and code generation using integer sets. Technical Report CS-TR97-275, Dept. of Computer Science, Rice University, April 1997.

[4] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[5] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[6] A. Aiken and A. Nicolau. Loop quantization: An analysis and algorithm. Technical Report 87-821, Dept. of Computer Science, Cornell University, March 1987.

[7] 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.

[8] E. Albert, J. Lukas, and G. Steele, Jr. Data parallel computers and the FORALL statement. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[9] 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.

[10] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[11] F. Allen and J. Cocke. A proram data flow analysis procedure. *Communications of the ACM*, 19:137–147, 1976.

[12] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[13] 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.

[14] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[15] B. Alpern, M. Wegman, and K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[16] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[17] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[18] ANSI X3J3/S8.115. Fortran 90, June 1990.

[19] Applied Parallel Research, Sacramento, CA. *Forge High Performance Fortran xhpf User's Guide*, version 2.1 edition, 1995.

[20] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[21] 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.

[22] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[23] U. Banerjee. *Speedup of ordinary programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report No. 79-989.

[24] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[25] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[26] T. Blank. The MasPar MP-1 architecture. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[27] Z. Bozkus. Compiling the FORALL statement on MIMD parallel computers. Technical Report SCCS-389, Northeast Parallel Architectures Center, Syracuse University, July 1992.

[28] Z. Bozkus. *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers.* PhD thesis, Syracuse University, June 1995.

[29] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, Portland, OR, November 1993.

[30] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[31] Z. Bozkus, L. Meadows, D. Miles, S. Nakamoto, V. Schuster, and M. Young. Techniques for compiling and executing HPF programs on shared-memory and distributed-memory parallel systems. In *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, December 1994.

[32] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, 1997.

[33] T. Brandes. Automatic translation of data parallel programs to message passing programs. In *Proceedings of AP'93 International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, Saarbrücken, Germany, March 1993.

[34] T. Brandes. Compiling data parallel programs to message passing programs for massively parallel MIMD systems. In *Working Conference on Massively Parallel Programming Models*, Berlin, 1993.

[35] T. Brandes. Adaptor: A compilation system for data parallel fortran programs. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches*

to Code Generation, Data Distribution, and Performance Prediction. Vieweg, Wiesbaden, 1994.

[36] R. G. Brickner, W. George, S. L. Johnsson, and A. Ruttenberg. A stencil compiler for the Connection Machine models CM-2/200. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[37] R. G. Brickner, K. Holian, B. Thiagarajan, and S. L. Johnsson. A stencil compiler for the Connection Machine model CM-5. Technical Report CRPC-TR94457, Center for Research on Parallel Computation, Rice University, June 1994.

[38] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[39] 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 Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[40] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[41] D. Callahan. Dependence testing in PFC: Weak separability. Supercomputer Software Newsletter 2, Dept. of Computer Science, Rice University, August 1986.

[42] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[43] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[44] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

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

[46] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[47] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[48] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:45–57, January 1981.

[49] S. Chakrabarti, M. Gupta, and J-D. Choi. Global communication analysis and optimization. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[50] 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.

[51] 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.

[52] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[53] S. Chatterjee, G. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[54] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[55] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-11, Xerox Corporation, December 1992.

[56] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Charleston, SC, January 1993.

[57] M. Chen and J. Cowie. Prototyping Fortran-90 compilers for massively parallel machines. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[58] 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.

[59] 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.

[60] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 55–66, Orlando, FL, January 1991.

[61] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C.-W. 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.

[62] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C.-W. Tseng. Unified compilation of Fortran 77D and 90D. *ACM Letters on Programming Languages and Systems*, 2(1–4):95–114, March–December 1993.

[63] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[64] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[65] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[66] K. Droegemeier, M. Xue, P. Reid, J. Bradley, and R. Lindsay. Development of the CAPS advanced regional prediction system (ARPS): An adaptive, massively parallel, multi-scale prediction model. In *Proceedings of the 9th Conference on Numerical Weather Prediction*, American Meteorological Society, October 1991.

[67] R. Esser and R. Knecht. Intel Paragon XP/S – architecture and software environment. Technical Report KFA-ZAM-IB-9305, KFA Research Centre, Juelich, April 1993.

[68] 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.

[69] R. Fatoohi. Performance analysis of four SIMD machines. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[70] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[71] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[72] W. George, R. Brickner, and S. L. Johnsson. Polyshift communications software for the Connection Machine systems CM-2 and CM-200. *Scientific Programming*, 3(1):83, Spring 1994.

[73] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[74] 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.

[75] V. Getov, T. Brandes, B. Chapman, T. Hey, and D. Pritchard. A comparison of HPF-like systems: Early prototypes. In *Proceedings of the Workshop on Performance Evaluation and Benchmarking of Parallel Systems*, Coventry, U.K., 1994.

[76] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[77] K. Gopinath and J. L. Hennessy. Copy elimination in functional languages. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[78] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[79] 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.

[80] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[81] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[82] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[83] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[84] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[85] J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal of Digital Equipment Corp.*, 7(3):5–23, Fall 1995.

[86] T. Haupt, S. Reddy, and G. Vengurlekar. Low level HPF compiler benchmark suite. Technical Report SCCS-735, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, August 1995.

[87] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[88] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[89] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[90] W. Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

[91] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[92] S. Hiranandani, K. Kennedy, and C.-W. 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.

[93] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[94] S. Hiranandani, K. Kennedy, and C.-W. 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.

[95] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[96] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, NY, 1984.

[97] Intel Corporation, Supercomputing Systems Division, Beaverton, OR. *Paragon XP/S Product Overview*, 1991.

[98] S. L. Johnsson. Language and compiler issues in scalable high performance scientific libraries. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[99] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[100] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 5–54. Prentice-Hall, 1981.

[101] K. Kennedy, J. Mellor-Crummey, and G. Roth. Optimizing Fortran 90 shift operations on distributed-memory multicomputers. In *Languages and Compilers for Parallel Computing, Eighth International Workshop*, Columbus, OH, August 1995. Springer-Verlag.

[102] 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.

[103] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993.

[104] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[105] K. Kennedy, N. Nedeljković, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[106] K. Kennedy and G. Roth. Context optimization for SIMD execution. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[107] K. Knobe, J. Lukas, and W. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[108] K. Knobe, J. Lukas, and G. Steele, Jr. Massively parallel data optimization. In *Frontiers '88: The 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, VA, October 1988.

[109] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[110] K. Knobe, J. Lukas, and M. Weiss. Optimization techniques for SIMD Fortran compilers. *Concurrency: Practice and Experience*, 5(7):527–552, October 1993.

[111] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[112] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[113] 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.

[114] 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.

[115] U. Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, October 1995.

[116] Kuck & Associates, Inc. *KAP User's Guide*. Champaign, IL 61820, 1988.

[117] 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.

[118] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[119] 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.

[120] 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.

[121] Z. Li, P. Yew, and C. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[122] J. Lukas and K. Knobe. Data optimization and its effect on communication costs in MIMD Fortran code. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[123] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran Reference Manual*, software version 1.1 edition, August 1991.

[124] MasPar Computer Corporation, Sunnyvale, CA. *MasPar System Overview*, March 1991.

[125] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[126] L. Meadows, D. Miles, C. Walinsky, M. Young, and R. Touzeau. The Intel Paragon HPF compiler. In *Proceedings of the 1995 Intel Supercomputer Users Group*, Albuquerque, NM, June 1995.

[127] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford Science Publications, 1990.

[128] A. Mohamed, G. Fox, G. v. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Applications benchmark set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, June 1992.

[129] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[130] J. Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[131] D. Palermo, E. Su, J. Chandy, and P. Banerjee. Communication optimizations used in the Paradigm compiler for distributed-memory multicomputers. In *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, IL, August 1994.

[132] J. Palmer and G. Steele, Jr. Connection Machine model CM-5 system overview. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[133] 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.

[134] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[135] J. R. Rice and J. Jing. Problems to test parallel and vector languages. Technical Report CSD-TR-1016, Dept. of Computer Science, Purdue University, 1990.

[136] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[137] B. Rosen, M. Wegman, and K. Zadeck. Global value numbers and redundant compuations. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[138] G. Roth and K. Kennedy. Dependence analysis of Fortran90 array syntax. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, Sunnyvale, CA, August 1996.

[139] 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.

[140] 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.

[141] G. Sabot. Optimizing CM Fortran compiler for Connection Machine computers. *Journal of Parallel and Distributed Computing*, 23(1):224–238, November 1994.

[142] G. Sabot, (with D. Gingold, and J. Marantz). CM Fortran optimization notes: Slicewise model. Technical Report TMC-184, Thinking Machines Corporation, March 1991.

[143] P. Schnorf, M. Ganapathi, and J. Hennessy. Compile-time copy elimination. *Software—Practice and Experience*, 23(11):1175–1200, November 1993.

[144] J. T. Schwartz. Optimization of very high level languages – I. Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, 1975.

[145] Stanford SUIF Compiler Group. SUIF: A parallelizing & optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[146] E. De Sturler and V. Strumpen. Scientific programming with High Performance Fortran: A case study using the xHPF compiler. *Scientific Programming*, 6(1):127–152, 1997.

[147] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[148] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-2 Technical Summary*, April 1987.

[149] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 1.0 edition, February 1991.

[150] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-200 Technical Summary*, June 1991.

[151] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM5 Technical Summary*, October 1991.

[152] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[153] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[154] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, January 1984.

[155] M. Wegman and K. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, New Orleans, LA, January 1985.

[156] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[157] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[158] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[159] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[160] M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.

[161] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[162] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

[163] M. J. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

[164] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[165] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[166] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, New York, NY, 1991.