COMPILING FORTRAN 90D/HPF FOR DISTRIBUTED MEMORY MIMD COMPUTERS

by

ZEKI BOZKUS

B.S., Middle East Technical University, 1988
M.S., Syracuse University University, 1990

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering in the Graduate School of Syracuse University

June 1995

Approved ______________________
Professor Geoffrey C. Fox

Date ______________________

Abstract

Distributed memory multiprocessors are increasingly being used to provide high performance for advanced calculations with scientific applications. Distributed memory machines offer significant advantages over their shared memory counterparts in terms of cost and scalability, though it is widely accepted that they are difficult to program given the current status of software technology. Currently, distributed memory machines are programmed using a node language and a message passing library. This process is tedious and error prone because the user must perform the task of data distribution and communication for non-local data access.

This thesis describes an advanced compiler that can generate efficient parallel programs when the source programming language naturally represents an application's parallelism. Fortran 90D/HPF described in this thesis is such a language. Using Fortran 90D/HPF, parallelism is represented with parallel constructs, such as array operations, *where* statements, *forall* statements, and intrinsic functions. The language provides directives for data distribution. Fortran 90D/HPF gives the programmer powerful tools to express a problem with natural data parallelism. To validate this hypothesis, a prototype of Fortran 90D/HPF was implemented. The compiler is organized around several major units: language parsing, partitioning data and computation, detecting communication and generating code.

The compiler recognizes the presence of communication patterns in the computations in order to generate appropriate communication calls. Specifically, this involves a number of tests on the relationships among subscripts of various arrays in a statement. The compiler includes a specially designed algorithm to detect communications and to generate appropriate collective communication calls to execute array assignments and forall statements.

The Fortran 90D/HPF compiler performs several types of *communication* and *computation* optimizations to improve the performance of the generated code.

Empirical measurements show that the performance of the output of the Fortran

90D/HPF compiler is comparable to that of corresponding hand-written codes on several systems.

We hope that this thesis assists in the widespread adoption of parallel computing technology and leads to a more attractive and powerful software development environment to support application parallelism that many users need.

Acknowledgments

I thank my advisor Geoffrey Fox for providing intellectual stimulus, encouragement and comments on my work during my graduate program at Syracuse University and for the long hours of fruitful discussion. His advice and suggestions form an integral part of this thesis. It has been a pleasure to have known and worked with Geoffrey Fox.

I would like to thank Alok Choudhary for the significant role he has played in shaping my career and intellectual growth. I am also grateful to him for taking a significant time to help me write various papers. Thanks also to Tomasz Haupt and Sanjay Ranka. Tomasz has been a great source of encouragement and support over the past few years. I thank Sanjay helping me with my first publication.

I would like to thank Alok Choudhary, Tomasz Haupt, Sanjay Ranka Simon Catterall and Kenneth Hawick for serving on my dissertation committee and for their many useful comments and suggestions. I thank Min-You Wu for his knowledge of parallel compiler design and his assistance that has had a great influence on this work. I would like to thank my colleagues Larry Meadows, Seteve Nakamoto, Mark Young and Vincent Schuster at The Portland Group, Inc. (PGI) Their invaluable comments helped me to improve the quality of this presentation. Special thanks to Mark Young for helping me to adapt PGI.

I am grateful to Parasoft for providing the Fortran 90 parser and Express without which the prototype compiler could have been delayed.

I want to thank Tomasz Haupt and Tom Van Raalte for careful reading of a draft.

Hakan Ancin, Mehmet Gulsen, Selim Akyokus, Kivanc Dincer, Ersel Anar and

Kubilay Cardakli have provided me with friendship and support. Thanks to you all. Special thanks to Omur Ozavci for the love and support.

Finally, I wish to acknowledge my father Mehmet and my mother Emine. Their love, encouragement, patience, and support have made possible my accomplishments and successes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern distributed memory computers offer very high levels of flexibility, scalability and performance but leave the programmer or the algorithm designer with the difficult and detailed task of planning computations, i.e, the orchestration of the entire parallel execution. The programmer is forced to manually distribute code and data in addition to explicitly managing communication. These tasks, in addition to being error-prone and time consuming, generally lead to non-portable code. Hence compiler technologies for distributed memory machines have received great attention due to their ease of use and portability.

To overcome the deficiencies of distributed memory machines, our Syracuse University group, along with colleagues at Rice University have designed the Fortran D language [1]. Fortran D is a version of Fortran that is enhanced with a rich set of data decomposition specifications providing a simple machine-independent programming model for most data-parallel computations. Standard Fortran 90 with these extensions is called "Fortran 90D", a Fortran 90 version of the Fortran D language. There is an analogous version of Fortran 77 with compiler directives and other constructs for use of parallel system, called Fortran 77D. Recently, the High Performance Fortran Forum, an informal group of people from academia, industry and national labs, led by Ken Kennedy, developed a language called HPF (High Performance Fortran) [2] based on Fortran D. Companies that have already committed to developing compilers and/or supporting HPF include Intel SSD, The Portland Group Inc., (PGI), DEC, IBM, and others. Hence, Fortran 90D and HPF are very similar. For this reason, we call the Fortran 90D language Fortran 90D/HPF.

1.1 Background

Current commercial parallel supercomputers are clearly the next generation of high performance machines [3, 4]. Although parallel computers have been commercially available for some time, their use has been mostly limited to academic and research institutions. This is mainly due to the lack of software tools available to develop parallel programs. Writing programs for parallel machines using message passing model is a complicated, time-consuming, and error-prone task [5]. Karp and Babb [6, 7] selected a simple program and rewrote it to run on nine commercially available parallel machines. They report being surprised to see how complicated some of these programs became.

Fortran has been a popular language for developing software for industry for the past few decades. Accordingly, there has been significant research in developing parallelizing compilers for Fortran codes. Most notable examples include Parafrase at the University of Illinois [8] and PFC at Rice University [9]. In this approach, the compiler takes a sequential Fortran 77 program as input, applies a set of transformation rules, and produces a parallelized code for the target machine. New transformation rules are added to the compiler as they are learned.

A sequential language, such as Fortran 77, hides a problem parallelism in sequential loops and in other sequential constructs. A program is written without any parallel constructs provided, even if the user is willing to express parallelism explicitly. Furthermore, the user may optimize the program to reduce memory usage and computation time. This makes the potential parallelism more difficult to detect. Therefore, compiling a sequential program into a parallel program is not a natural approach. An alternative approach is to use a source programming language that can naturally represent an application using parallel constructs. Fortran 90 (with some extensions) is such a language. The extensions may include the *forall* statement and compiler directives for data partitioning, such as decomposition, alignment, and distribution.

From our point of view, Fortran 90 (and its dialects including Fortran 90D/HPF) is not regarded as the natural portable language only for SIMD computers [10], but as a natural language for parallelism of a large class of synchronous and some loosely synchrounous problems [11, 12, 13, 14]. In Fortran 90D/HPF, parallelism is represented with parallel constructs, such as array operations, *forall* statements, and intrinsic functions. The *forall* statement is not a standard construct in Fortran 90; however, this construct is included in Fortran 90D/HPF.

A Fortran 90D/HPF parallel compiler exploits only the parallelism expressed in these parallel constructs. Fortran 90D/HPF compiler does not attempt to *parallelize* other constructs, such as *do* loops and *while* loops, since they are naturally sequential. Developing a compiler under this assumption becomes much easier. Also users can reliably understand where parallelism will be exploited.

1.2 Hypothesis

Our hypothesis is that an advanced compiler can generate efficient parallel programs for distributed memory machines if a programming language can naturally represent an application's parallelism. Fortran 90D/HPF is such a language. In Fortran 90D/HPF, parallelism is represented with parallel constructs, such as array operations, *where* statements, *forall* statements, and intrinsic functions. The Fortran 90D/HPF language provides distribution directives to help the compiler distribute data efficiently on distributed memory machines which demand high data locality for good performance. This language gives the programmer a powerful tool to express data parallelism that is natural to a problem. To validate this hypothesis, we have implemented the Fortran 90D/HPF compiler.

1.3 Contributions

The core of this thesis is devoted to demonstrating that Fortran 90D/HPF programs written in a data-parallel programming style can be compiled into efficient parallel programs for distributed memory machines.

This thesis describes the design and implementation of a Fortran 90D/HPF compiler. A systematic methodology to process distribution directives of Fortran 90D/HPF is presented. Furthermore, techniques for data and computation partitioning, communication detection and generation, and run-time support for the compiler are discussed.

The compiler must recognize the presence of communication patterns in the computations in order to generate appropriate communication calls. Specifically, this involves a number of tests on the relationships among subscripts of various arrays in a statement. We designed an algorithm to detect communications and to generate appropriate collective communication calls to execute array assignments and foralls statement on distributed memory machines.

The thesis presents several types of *communication* and *computation* optimizations used to maximize the performance of the generated code: Communication optimizations can be classified as

- Communication hierarchy
- Vectorized communication
- Message aggregation
- Evaluating expression
- Communication parallelization
- Communications union
- Eliminate unnecessary communications

- Reuse of scheduling information.

In addition, some computation optimizations are developed for sequentialization of *forall* statements such as

- Forall dependency
- Forall loop interchange
- Forall mask insertion

Some of these optimizations are validated with examples.

We have indicated our confidence in the performance of the code generated by publishing the absolute execution times of several benchmarks. We believe that Fortran 90D/HPF greatly improves programmer productivity. Fortran 90D/HPF programs are shorter, easier to write, and easier to debug than programs written in Fortran 77 with message passing. We have found that Fortran 90D/HPF makes it much easier to tune nontrivial programs.

We believe that the methodology used to process data distribution, computation partitioning, communication system design and the overall compiler design can be used by implementors of future HPF compilers.

1.4 Overview

This thesis describes the design, implementation, and evaluation of the Fortran 90D/HPF compiler. Here we present an overview of the remainder of the thesis.

Fortran 90D/HPF Language: Chapter 2 presents the Fortran 90D/HPF language, concentrating on its strategy for expressing data parallelism and mapping data to the underlying parallel architecture. The chapter discusses a number of language issues including the crucial language features for partitioning data.

Architecture of the Compiler: Chapter 3 presents the major phases of the Fortran 90D/HPF compiler. The basic structure of the compiler is organized around seven major phases: front-end, semantic analysis of distribution directives, transformation of all parallel constructs into equivalent internal *forall* representations, sequentialization, building array descriptors to pass to the runtime routines, and code generation. This chapter describes some of these phases and also provides examples showing code generation for parallel statements.

Distribution Model: Chapter 4 presents methods to compile distribution directives and illustrates the important design considerations. Specifically, we show how the alignment and distribution directives can be systematically processed to produce efficient code.

Communication Model: Chapter 5 presents how the Fortran 90D/HPF compiler recognizes the presence of communication patterns in computations and generates appropriate communication calls. The chapter also describes computation partitioning, the run-time support system and the storage management methods used by the Fortran 90D/HPF compiler.

Optimization: Chapter 6 presents several optimization techniques to reduce the total cost of communication and computation. The chapter gives an example program to show the effectiveness of optimizations.

Experimental Results: Chapter 7 presents benchmark results to illustrate performance obtained using the Fortran 90D/HPF compiler. The chapter emphasizes the portability and scalability of the Fortran 90D/HPF compiler. It gives the performance results for different distributions and compares with the hand-written Fortran 77 + message passing codes.

1.5 Summary of Related Work

Fortran 77D

One compiler related to Fortran 90D is the Fortran 77D compiler which was developed at Rice University[15, 16]. The Fortran 77D compiler introduces and classifies a number of advanced optimizations needed to achieve acceptable performance. Fortran 77D compiler is guided by the concept of data dependency, unlike Fortran 90D. The compiler performs *vector message pipelining*, a technique that combines message vectorization and message pipelining to hide communication. It detects *pipelined* computations via *cross-processor loops* and exploits pipeline parallelism while balancing communication costs though *coarse-grain pipelining* [17].

HPF

High Performance Fortran Forum, an informal group of people from academia, industry and national labs, led by Ken Kennedy, developed a language called HPF (High Performance Fortran) [2, 18]. HPF language combines the full Fortran 90 language with special user annotations dealing with data distribution It is expected that HPF will be the standard programming language for computationally intensive applications on many types of machines, such as traditional vector processors and newer massively parallel MIMD and SIMD multiprocessors. Companies that have already committed to developing compilers and/or supporting HPF include Intel, TMC, Portland Group(PGI), DEC, IBM, and others.

SUPERB and Vienna Fortran 90

The compilation system SUPERB (University of Vienna) [19] takes a sequential Fortran program and a specification of the desired data distribution. SUPERB then converts the code to an equivalent program to run on a distributed memory machine by inserting the communication required and optimizing communications where possible. The user is able to specify arbitrary block distributions and the compiler performs dependence analysis to guide interactive program transformations.

Recently, SUPERB has been adapted for a new language called Vienna Fortran 90. Vienna Fortran 90 is a language extension of Fortran 90 which enables the user to write programs for distributed memory machines using only global references. It is similar to Fortran 90D/HPF language. However, Vienna Fortran does not provide a data decomposition, but does support alignment and distribution directives.

Kali

The KALI compiler [20, 21] is the first compiler system available to support both regular and irregular computations on MIMD machines, using an inspector/executor strategy to handle indirectly distributed data. An inspector/executor strategy is used for run-time preprocessing of the communication for irregularly distributed arrays. KALI requires that the programmer explicitly partition loop iterations onto the processor grid. It produces code which is independent of the number of available processors.

ASPAR

ASPAR (Automatic and Symbolic PARallelization) [22] consists of a source-to-source parallelizer and a set of interactive graphic tools. It uses symbolic analysis and data dependency analysis methods to determine an explicit data decomposition scheme. ASPAR utilizes collective communication primitives from the EXPRESS run-time system for distributed memory machines. Communication utilizing EXPRESS primitives are then automatically generated. ASPAR performs less compile-time analysis and optimization, instead relying heavily on run-time support system.

Dataparallel C

Dataparallel C [23, 24] is a variant of the C* programming language, designed by Thinking Machine Corporation for its Connection Machines processor array. Data parallel C extends C to provide the programmer with access to a parallel virtual machine. It supports a variety of standard domain decomposition primitives, and it also allows the programmer to specify a custom mapping of data to the distributed memories of the hypercube. This compiler generates code suitable for execution on both the nCUBE 3200 and the Intel iPSC/2.

ARF

ARF (ARguably Fortran) is a compiler for irregular computations [25, 26, 27]. Distributed arrays are declared in ARF source. An ARF user can declare a mapping of an array in an irregular manner. It is capable of handling a wide range of irregular problems in scientific computing. The ARF compiler generates inspector and executor loops with embedded primitives. It provides an interface between application programs and the PARTI run-time support primitives using a set of run-time library routines that support irregular communication on MIMD distributed memory machines. The Fortran 90D/HPF compilation system may also use the PARTI[28] system to support irregular communications.

CM Fortran

The CM Fortran language [29, 30] is implemented as a subset of Fortran 77, extended by Fortran 8x array features to support a data parallel programming style for the Connection Machine (CM) computer system. CM Fortran maps arrays into the CM architecture. The compiler generates code to be executed by a CM system with a DEC VAX front end. For a given routine, both VAX and CM code are generated. In general, array code is executed by CM processors, while scalar code is executed by the VAX. This approach is used by the CM-1 and CM-2 SIMD architectures. The new version of the CM Fortran compiler [31] is developed for the CM-5 MIMD architecture [32]. The new compiler generates two classes of output code: code for scalar control processors and code targeted for the nodes of the CM-5 parallel processing elements.

Cray MPP Fortran

Cray Research Inc. has announced a set of language extensions to Cray Fortran [33] which enable the user to specify the distribution of data and work. The extensions provide intrinsics for data distribution and permit redistribution at subroutine bounds. Furthermore, the CRAY extensions permit the user to structure the executing processors by giving them a shape and weighting the dimension. Several methods for distributing iterations of loops are provided. Many features of shared memory parallel languages have been retained; these include: critical sections, events and locks.

Additional Research

Callahan and Kennedy [34] proposed distributed-memory compilation techniques based on data-dependence driven program transformations. These techniques were implemented in a prototype compiler in the ParaScope programming environment.

Under the ADAPT system [35] developed at the University of Southampton, the parallelization process of Fortran 90 is guided by distribution declarations for arrays, in a similar but, more restricted approach than Fortran D.

The ADAPTOR [36] is a tool that transforms data parallel programs written in Fortran with array extensions and layout directives to explicit message passing.

Li and Chen [37, 38] describe general compiler optimization techniques that reduce communication overhead for Fortran-90 implementation on massively parallel machines. Our compiler uses similiar pattern matching techniques to Li and Chen to detect communication.

Chapter 2

Fortran 90D/HPF Language

2.1 Introduction

Our Syracuse University group, along with colleagues at Rice University have designed the Fortran D language [1]. Fortran D is a version of Fortran that is enhanced with a rich set of data decomposition specifications Standard Fortran 90 with these extensions is called "Fortran 90D", a Fortran 90 version of the Fortran D language. There is an analogous version of Fortran 77 with compiler directives and other constructs for use of parallel system, called Fortran 77D. Rice University implemented Fortran 77D compiler. Our group at Syracuse University implemented Fortran 90D compiler. Recently, the High Performance Fortran Forum, an informal group of people from academia, industry and national labs, led by Ken Kennedy, developed a language called HPF[2] based on Fortran D. HPF uses standard Fortran 90 as base language. Hence, Fortran 90D and HPF becomes very similar. For this reason, we call the Fortran 90D language as Fortran 90D/HPF. The core part of this thesis does not claim to design a language such as Fortran 90D/HPF, but claims to design a compiler for Fortran 90D/HPF language.

This chapter presents the Fortran 90D/HPF language, concentrating on its strategy for expressing data parallelism and mapping data to the underlying parallel architecture. The chapter discusses a number of language issues including the crucial language features for partitioning data. Because data partitioning ultimately determine the shape of the resulting code by defining the computation partitioning and communication, we start by describing the data distribution.

2.2 Data Distribution

Data distribution can be done in two steps which separate machine independent problem parallelism from machine dependent details. The first step is to determine the best alignment among different arrays. To reduce unnecessary data movement, distributed arrays should be aligned with each other in a fashion that is usually determined by the underlying computation structure. The alignment of arrays depends on the program and is often machine-independent. The second step is to determine how arrays should be distributed to the underlying hardware and is therefore machine dependent. The objective of array distribution is to balance the computation load for each processor and to minimize the communication between processors. Array distribution is largely dependent on hardware, such as the number of processors, communication mechanisms, and interconnection topologies.

Fortran 90D/HPF provides users with annotation facilities for data partitioning. The annotation facilities take the form of compiler directives.

2.2.1 Decomposition directives

A decomposition directive declares a problem domain. The directive declares the name, dimensionality, and size of a decomposition.

A *decomposition-directive* is:

DECOMPOSITION *decomposition-spec*-list

A *decomposition-spec* is:

array-*name* (*size*-list)

A *size* is:

scalar-integer-*constant*

The decomposition directive defines arrays as data parallel [39] and is machine independent. Examples of decomposition directives are shown below:

DECOMPOSITION A(N)

DECOMPOSITION B(N,N)

where A is declared as a one-dimensional decomposition of size N, and B is a two-dimensional N by N decomposition.

2.2.2 Alignment directives

An alignment directive aligns one array to another array. Arrays aligned with each other share a common "data parallelism". Alignment directives specify which elements of two arrays are to be allocated together by aligning each axis of a source array with a given target array.

An *alignment-directive* is:

ALIGN *source-spec* WITH *target-spec*

A *source-spec* is:

source-array-*name* (index-*name*-list)

A *target-spec* is:

target-array-*name* (*target-axis-spec*-list)

A *target-axis-spec* is one of:

- index-*name* [$*$ *stride-value*] [+ *offset-value*]
- *index-value*
- *subscript-triplet*

A *stride-value* is:

integer-constant

An *offset-value* is:

integer-constant

An *index-value* is:

[–] *integer-constant*

The number of *index-name*s in a *source-spec* must equal the rank of the source array. The number of *target-axis-spec*s in a *target-spec* must equal the rank of the target array.

The following examples of alignment directives specify different alignment patterns:

1. Alignment offsets:

   ALIGN A(I,J) with X(I-1,J+1)

2. Alignment strides:

   ALIGN B(I,J) with X(I*2,J*2)

3. Embedding:

   ALIGN C(I) with X(I,2)

4. Permutation:

   ALIGN D(I,J) with X(J,I)

5. Collapse:

   ALIGN E(I,*) with Y(I)

6. Replication:

   ALIGN F(I) with X(I,*)

Alignment is usually machine independent. A complete specification of the alignment directive is available in [1].

2.2.3 Distribution directives

A distribution directive provides control over the distribution of an array. Specifications are block distribution, scattered distribution, block-scattered distribution, or no distribution. The relative weight of distribution along each axis indicates the distribution ratio among axes. The distribution ratio is defined as the ratio of the number of partitions along different axes.

A *distribution-directive* is:

DISTRIBUTE *distribution-spec*-list

A *distribution-spec* is:

array-*name* (*axis-descriptor*-list)

An *axis-descriptor* is one of:

- BLOCK[(*weight*)]
- CYCLIC[(*weight*)]
- BLOCK_CYCLIC(*size* [,*weight*])
- :

A *weight* is:

scalar-integer-*constant*

A *size* is:

scalar-integer-*constant*

The number of *axis-descriptor*s in a *distribution-spec* must equal the rank of the array specified by array-*name* in the *distribution-spec*. Each *distribution-spec* specifies distribution information for the array given by array-*name*. The array is distributed with the attributes specified by the *axis-descriptor*-list of that *distribution-spec*. Each *axis-descriptor* defines the attributes of the corresponding dimension that is to be distributed. The keywords BLOCK, CYCLIC, BLOCK_CYCLIC, and ":" control the distribution style. For each *axis-descriptor* in the list:

- BLOCK indicates that the corresponding dimension is to be block distributed (contiguous).
- CYCLIC indicates that the corresponding dimension is to be scattered distributed (interleaving).
- BLOCK_CYCLIC(*size*) indicates that the corresponding dimension is to be block-scattered-distributed; that is, blocks of size *size* are scattered.
- A star "*" indicates that the corresponding dimension will not be distributed.

The keyword CYCLIC specifies a scattered-distribution that has also been called an interleaving partitioning. It is a powerful distribution method to balance loads for irregular computation structures.

The distributions shown in Figures 1(a) and 1(b) are examples of *block* distributions: each processor contains a contiguous subarray of the specified array. Figure 2(a) illustrates a *cyclic* distribution in which columns of an array are distributed onto four processors so that each processor, starting from a different offset, contains every fourth column. Figure 2(b) shows a distribution that is cyclic in both dimensions onto four processors arranged in a 2 x 2 square. Figure 3 shows the combination distribution with *block* and *cyclic*.

2.3 Data Parallelism in Fortran 90D/HPF

Parallelism can be explicitly expressed in Fortran 90/HPF using several language features: Fortran 90 array assignments, masked array assignments using *where* statements, *where* constructs, *forall* statements, *forall* constructs and intrinsic functions.

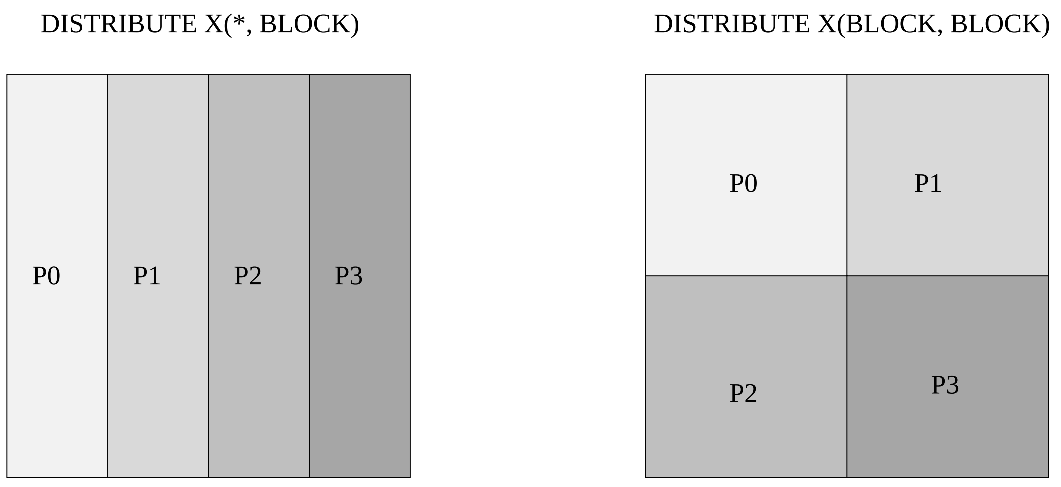


Figure 2.1: 2-D Block distributions

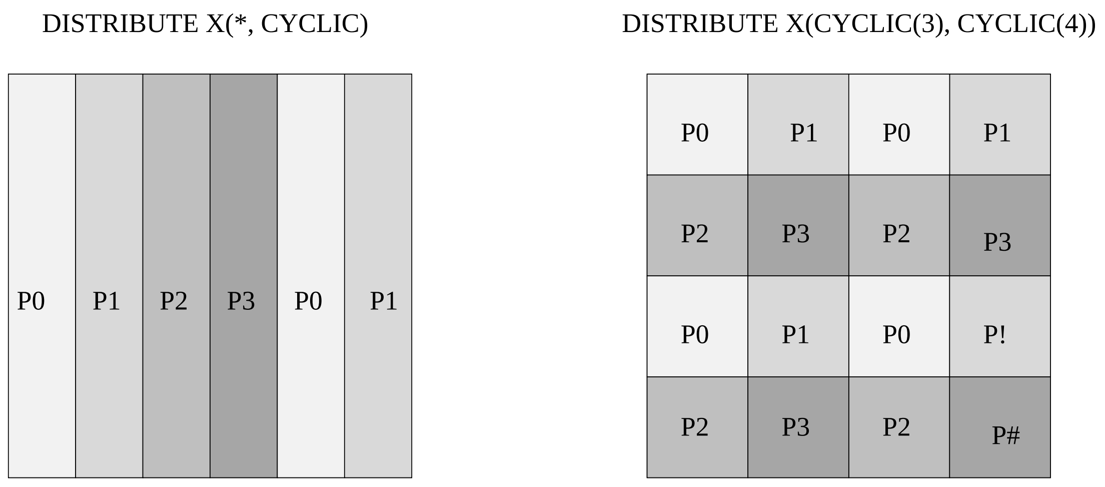


Figure 2.2: 2-D Cyclic distributions

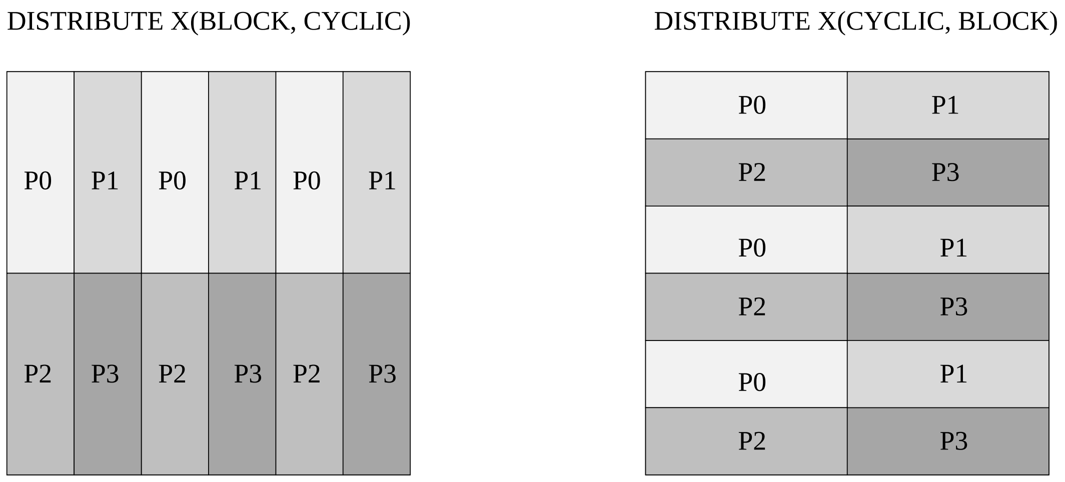


Figure 2.3: 2-D Combination distributions

The *forall* statement [2] is an elemental array assignment statement used to specify an array assignment in terms of array elements or array sections. The element array may be masked with a scalar logical expression. The *forall* statement effectively describes a collection of assignments to be executed elementally. Some examples of *forall* statements are:

```
FORALL( I = 1:N, J=1:N ) H(I,J) = 1.0 / REAL(I + J -1)
FORALL( I = 1:N, J=1:N, A(I,J) .NE. 0.0 ) B(I,J) = 1.0 / A(I,J)
```

The semantics of a FORALL statement are an assignment to each of these elements or sections (one for every possible combination of subscript values for which the mask expression is true) with all right-hand sides being evaluated before any left-hand sides are assigned.

The *forall* statement and construct are new language features expressing data parallelism, that is, providing a convenient syntax for simultaneous assignments to large groups of array elements. The functionality these statements provide is similar to that provided by the array assignments and the *where* constructs in Fortran 90. All Fortran 90 array assignments, including *where*, can be expressed using *forall* statements. However, Fortran 90 places several restrictions on array assignments. In particular, it requires that operands of the right side expressions be conformable with the left hand side array. These restrictions are relaxed in *forall* statements. In addition, a *forall* may call user-defined functions, simulating Fortran 90 elemental function invocation. Functions that may be called in a *forall* loop must not produce any side effects.

The *forall* statement essentially preserves the semantics of Fortran 90 array assignments and the *forall* construct is semantically equivalent to a sequence of *forall* statements. The array elements may be assigned in an arbitrary order, in particular, concurrently. To preserve determinism of the result, it is required that each array element only be assigned once. The execution of the *forall* assignment may require an intra-statement synchronization: the evaluation of the left hand side expression of the *forall* assignment must be completed for all array elements before the actual assignment is made. Then, the processors must be synchronized again, before the next array assignment is processed.

2.4 Intrinsic Functions

An important features of Fortran 90 is its rich set of intrinsic functions and subroutines. These intrinsics allow the coding of data parallel programs at a higher level, and potentially with greater efficiency, than if their functions were programmed by users. They not only provide a concise means of expressing operations on arrays, but also identify parallel computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, reshape, and matrix multiplication.

2.4.1 Array Reduction Functions

The array reduction function SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL perform numerical, logical and counting operations on arrays. They may be applied to a whole array to give a scalar result or they may be applied over a given dimension to yield a result of rank reduced by one.

2.4.2 Array Construction Function

The functions MERGE, SPREAD, PACK and UNPACK construct new arrays from input arrays. MERGE combines two conformable arrays into one array by an element-wise choice based on a logical mask. SPREAD constructs an array from several copies of an actual argument. PACK and UNPACK, respectively, gather and scatter the elements of a one-dimensional array to form another array specified by a logical mask.

2.4.3 Array Manipulation Function

The functions TRANSPOSE, EOSHIFT, and CSHIFT manipulate arrays. TRANSPOSE performs the matrix transpose operations on a two dimensional array. The shift functions leave the shape of an array unaltered but shift the positions of the elements parallel to a specified dimension of the array. These shifts are either circular (CSHIFT), in which case elements shifted off one end reappear at the other end, or end-off (EOSHIFT), in which case specified boundary elements are shifted into vacated positions.

2.4.4 Array Location Functions

The functions MAXLOC and MINLOC return the location (subscripts) of an element of an array that has maximum and minimum values, respectively. By use of an optional logical mask that is conformable with the given array, the reduction may be confined to any subset of the array.

2.5 Discussion

From our point of view, Fortran 90 (and its dialects including Fortran 90D/HPF) is not regarded as the natural portable language for SIMD computers [10], but as a natural language for parallelism of a class of what we have called synchronous and some loosely synchronous problems [11, 12, 14, 13]. In Fortran 90D/HPF, parallelism is represented with parallel constructs, such as array operations, *forall* statements, and intrinsic functions.

A Fortran 90D/HPF program is a Fortran 90 program augmented with a set of data decomposition specifications. If these specifications are ignored the program can be run without change on a sequential machine. Compilers for parallel machines can use the specifications not only to decompose data but also to decompose computations. Moreover, the directives could be generated by an automatic partitioner in future version of compilers. A description of generating optimal alignment can be found in [40]. The distribution directives could be generated with a constraint-based approach [41] or with the guide of a performance estimator [42].

Chapter 3

Architecture of the Compiler

The major phases of the Fortran 90D/HPF compiler are shown in Figure 4. Fortran 90D/HPF takes a syntactically correct Fortran 90D/HPF program and transforms it into Fortran 77 plus runtime code. The basic structure of the compiler is organized around seven major phases: front-end, semantic analysis of distribution directives, transformation of all parallel constructs into equivalent internal forall representations, sequentialization, building array descriptors to pass to the runtime routines, and code generation. This chapter describes some of these phases and also provides examples showing code generation for parallel statements.

3.1 The Front-End

The first step of the compilation is to generate a parse tree. This module parses the input program into an abstract syntax tree, performs semantic analysis to annotate the tree with type information, and builds a symbol table; it also performs error checking. The front-end to parse Fortran 90 was obtained from ParaSoft Corp. It is enhanced such that it accepts all legal forms of Fortran 90D/HPF. This includes the

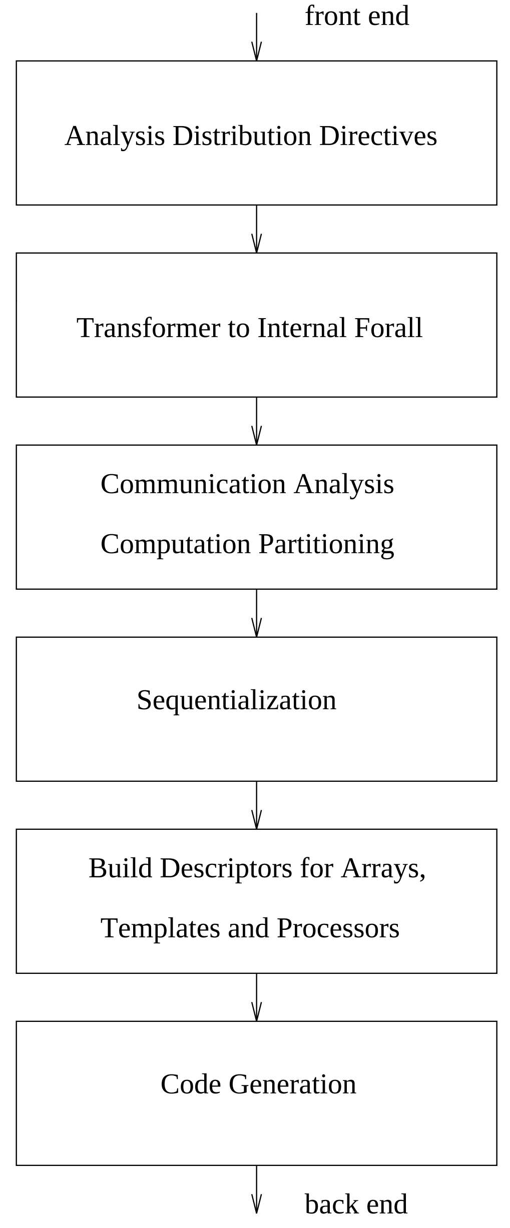


Figure 3.4: The architecture of the Fortran 90D/HPF compiler.

directives for ALIGN, DISTRIBUTE, DECOMPOSITION and the FORALL statement and the FORALL construct. We will not discuss the front-end in detail. We will discuss more about the phases which are directly related to Fortran 90D/HPF language parallelism.

3.2 Analysis of Distribution Directives

The first phase semantically analyzes all Fortran 90D/HPF directives and stores the resulting information on decomposition, distribution, alignment, and processor arrangements in the array descriptor portion of the symbol table. This information is used throughout compilation. For variables that are not explicitly mapped (and compiler-created temporaries), the compiler chooses a default distribution and alignment. Figure 5 shows a typical sequence of Fortran 90D/HPF directives used to align and distribute arrays. The compiler identifies alignment chains and determines that no PROCESSORS directive is present. In the absence of a PROCESSORS directive, the compiler generates code to dynamically determine the number of available processors and uses a default one-dimensional processor arrangement.

```
       integer, dimension(100) :: A, B,C
!F90D$ align A with B
!F90D$ align B with C
!F90D$ distribute C(cyclic)
```

Figure 3.5: An example of Fortran 90D/HPF directives.

The compiler stores the distribution information as in Figure 6. It creates a default template T for array C and collapses all alignments to C into alignments to T. T is distributed by default onto P, the default processor arrangement.

```
          integer, dimension(100) :: A, B,C
!F90D$ processors P(number_of_processors())
!F90D$ decomposition T(100)
!F90D$ align A with T
!F90D$ align B with T
!F90D$ align C with T
!F90D$ distribute T(cyclic) onto P
```

Figure 3.6: Internally completed distribution directives.

3.3 Internal Forall Transformations

The next phase of the compiler transforms parallel constructs: array assignments, *where* statements/constructs, and *forall* statements/constructs into one internal representation which is similar to a *forall* statement. However, the internal representation only allows calls to pure functions (functions without side effects). If there is a call to a transformational intrinsic in the original construct, the transformation phase removes it during conversion to the internal representation to prevent joining of all processors within the parallel construct.

In Fortran 90D/HPF, we choose the *forall* statement as our intermediate language construct. *Array assignment* statements and *where* statements can be translated into equivalent *forall* statements with no loss of information. After transformation, subsequent phases of the compiler, such as optimization, process parallel constructs using only the internal representation. The Fortran 90D/HPF compiler generates the same intermediate and final code regardless of which constructs the programmer chooses.

Figure 7 gives the algorithm for transforming the *array assignment statement* and *where statement* into equivalent *forall statements*.

```
! array assignment statement
      A(l1:u1:s1)=B(l2:u2:s2) + ...
! equivalent forall statement
      FORALL(i=l1:u1:s1) A(i)=B(l2+((i-l1)/s1) * s2) + ...
! where statement
      WHERE(C(l3:u3:s3).NE.0.0)  A(l1:u1:s1)=B(l2:u2:s2) + ...
! equivalent forall statement
      FORALL(i=l1:u1:s1, C(l3+((i-l1)/s1)*s3).NE.0.0)
     &      A(i)=B(l2+((i-l1)/s1)*s2)
```

Figure 3.7: Transforming array assignment and where into foralls.

3.4 Communication Analysis

The communication phase of the compiler selects the communication primitives. It inserts code for allocation of buffers as well as calls to communication primitives. This phase also partitions computations by modifying the bounds of parallel loops and inserting conditional statements which restrict execution of statements to the appropriate processors. This pass also performs numerous communication optimizations. This phase is discussed further in Chapter 6.

3.5 Sequentialization

The computation assigned to each processor element will be executed sequentially. Fortran 90D/HPF has to do sequentialization for the program with *forall* statements after modifing the bounds of *forall* statements.

There are two main problems in sequentialization: how to create a legal sequentialized version of the parallel construct and reduce the space needed for array temporaries. The *fetch-before-store* semantics of an array operation or of a *forall* statement can be stated as follows: the entire right-hand side must be fetched and evaluated before any results can be stored in the left-hand side. Thus, the array operation

```
a = b + a
```

or *forall* statement

```
forall (i=1:N) a(i) = b(i) + a(i)
```

can be correctly interpreted by the following two *do* loops:

```
do i=1,N
  tmp(i) = b(i) + a(i)
end do
do i=1,N
  a(i) = tmp(i)
end do
```

Here, the array temporary *tmp* can be eliminated without changing the semantics of the original *forall* statement as follows:

```
do i=1,N
  a(i) = b(i) + a(i)
end do
```

However, the temporary array cannot be simply eliminated for the following *forall* statement:

```
forall (i=1:N) a(i) = b(i) + a(i-1)
```

Various techniques, such as loop reversal, loop interchange, and loop skewing can be applied to eliminate the array temporary in some cases. Sequentialization algorithms can be found in [43], which include algorithms for finding a correct sequentialization, and the use of loop reversal and loop interchange. But these methods exhibit poor *spatial locality* [44]. The current Fortran 90D/HPF compiler does not perform the loop transformations and complex dependency analysis needed to eliminate temporary usage. However, the compiler checks whether the left hand side (*lhs*) array is used in the right hand side (*rhs*) of the *forall* statement. If not, the compiler does not create a temporary.

3.6 Building Array Descriptors

The next phase of the compiler builds descriptors for each processor arrangement, decomposition, and array so that all information available at compile time is also available at run-time. Processor descriptors include information on the shape and mapping of the processor arrangement. A decomposition data structure describes the shape and distribution of the template. An array descriptor contains all the information necessary to determine the shape of the array, the decomposition to which it is aligned, and how the alignment is specified.

3.7 Code Generation

Finally, the code generator targets an Single Program Multiple Data (SPMD) programming model. The SPMD model provides for a system where each processor executes the same program, but operates on different data. This is implemented by loading the same program image into each processor. Each processor allocates and operates on its own local portion of distributed arrays, according to the distributions, array sizes and number of processors as determined at runtime.

The generated code is structured as alternating phases of local computation and calls to communication primitives. Communication primitives are synchronization points. Most of the time the compiler does not know until run-time which groups of processors may communicate, so it guarantees that communication primitives will be called by all processors.

The following sections provide examples showing code generation for parallel statements, such as array assignment, *where* and *forall* statements.

3.7.1 Array Assignment Parallelism

The Fortran 90D/HPF compiler treats Fortran array expressions as parallel expressions. Each node or processor on the parallel system will execute its part of the computation (if the array associated with *lhs* of the expression is distributed). Array constructs are internally converted to an equivalent FORALL statement and then the distributed array is computed with a FORALL statement that is parallelized by localizing array indices. For example, the following Fortran 90 array statement is parallelized and produces the Fortran 77 code shown:

```
REAL X(16), Y(16)
```

```
!F90D$ DISTRIBUTE Y(BLOCK)
!F90D$ DISTRIBUTE X(BLOCK)
      Y=X+1
```

The following code would be generated and run locally on each processor.

```
call set_bound(x_dist,1,1,16,1,llb,lub)
do i1 = llb, lub
   y(i1) = x(i1) + 1
enddo
```

Note that the call set_bound() is a Fortran 90D/HPF runtime library routine. This routine generates the bounds for the index space of the array residing on the local processor. This routine will be discussed more in the Computation Partitioning Section of Chapter 5. For example, on a four processor system, this call would generate different loop bound depending on the processor the call is made on, and the portion of the array stored on that processor, as shown below:

```
! Processor 1                  ! Processor 3
do i1 = 1, 4                    do i1 = 9, 12
   y(i1) = x(i1) + 1               y(i1) = x(i1) + 1
enddo                           enddo

! Processor 2                  ! Processor 4
do i1 = 5, 8                   do i1 = 13, 16
   y(i1) = x(i1) + 1                 y(i1) = x(i1) + 1
enddo                          enddo
```

3.7.2 Where Statement Parallelism

The *where* statement is a Fortran 90 statement that conveys parallelism in a manner similar to array assignment described in the previous section. The compiler adds a conditional statement to mask the elements of the array's index space that are assigned or not assigned a particular value. For example, given that X and Y are distributed arrays, the following WHERE statement produces code similar to the Fortran 77 output shown:

```
WHERE(X/=0) Y=X
```

The following code would be generated:

```
call set_bound(x_dist,1,1,16,1,llb,lub)
do i1 = llb, lub
   if (x(i1) .ne. 0) then
      y(i1) = x(i1)
   endif
enddo
```

The generated code is similar to the node code for an array expression, with the addition of the conditional within the DO loop.

3.7.3 Forall Statement Parallelism

The *forall* statement allows specification of a set of index values and an assignment expression utilizing the index values (or using a masked subset of the index values). The computation involving the index values for the assignment expression may be performed in an any order on a scalar machine, or in parallel on a parallel system. For more details on the definition of FORALL refer to [18]. The following example shows a simple masked FORALL and the Fortran 77 code generated by the Fortran 90D/HPF compiler.

```
FORALL(I=1:15, X(I)>5) X(I)=Y(I)

call set_bound(x_dist,1,1,15,1,llb,lub)
do i1 = llb, lub
   if (x(i) .gt. 5) then
      x(i) = y(i)
   endif
enddo
```

Note that intrinsic functions can be called from the expression part of a FORALL statement.

Chapter 4

Distribution Model

4.1 Introduction

Distributed memory systems solve the memory bottleneck of vector supercomputers by having separate memory for each processor. However, distributed memory systems demand high locality for good performance. Therefore, the distribution of data across processors is of critical importance to the efficiency of a parallel program in a distributed memory system. Fortran 90D/HPF language provides distribution directives to help the compiler distribute data efficiently on distributed memory machines.

There are three ways to generate the directives: 1) users can insert them, 2) programming tools can help users to insert them, or 3) automatic compilers can generate them. In the first approach, users write programs with explicit distribution and alignment directives. A programming tool can generate useful analysis to help users decide partitioning styles, and measure performance to help users improve program partitioning interactively [34, 42, 45]. The directives can also be generated automatically by compilers. Promising work has been done along case 2 and 3 [46, 40, 47, 41, 48]. However, these ideas have not yet been implemented in a practical general system, so we do not consider automatic partitioning in the Fortran 90D/HPF compiler.

The focus of this chapter is to describe the design and implementation of the data partitioning module. It discusses how to distribute data given data distribution directives and illustrates what the important design considerations are. Specifically, we show how the alignment and distribution directives can be systematically processed to produce efficient code.

4.2 Expressive Power of Directives

In this Section, we would like to give an example to show expressive power of Fortran 90D/HPF distribution directives on a real application.

Consider the data partitioning schema for matrix-vector multiplication proposed by Fox *et al.*[49] and shown in Figure 8. The matrix vector multiplication can be described as

$$y = Ax$$

where y and x are vectors of length M, and A is an $M \times M$ matrix. To create the distribution shown in Figure 8, one can use the following directives in a Fortran 90D/HPF program.

```
!F90D$   DECOMPOSITION    TEMPL(M,M)
!F90D$   ALIGN A(I,J)  WITH TEMPL(I,J)
!F90D$   ALIGN X(J)    WITH TEMPL(*,J)
!F90D$   ALIGN Y(I)    WITH TEMPL(I,*)
!F90D$   DISTRIBUTE   TEMPL(BLOCK,BLOCK)
```

If this program is mapped onto a 4x4 physical processor system, the Fortran 90D/HPF compiler generates the distributions shown in Figure 8. Matrix A is distributed in both dimensions. Hence, a single processor owns a subset of matrix rows and columns. X is column-distributed and row-replicated. But Y is row-distributed and column-replicated.

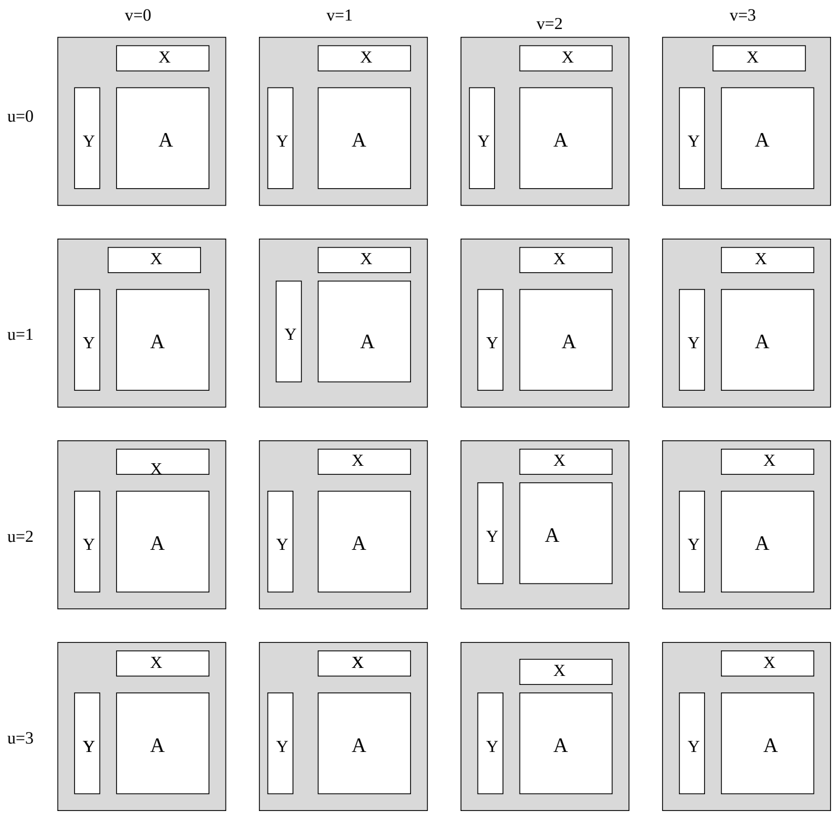


Figure 4.8: Matrix-vector decomposition.

4.3 Design Methodology

The Fortran 90D/HPF compiler maps arrays to physical processors using a three stage mapping as shown in Figure 9. This three stage mapping has also been proposed in HPF[2].

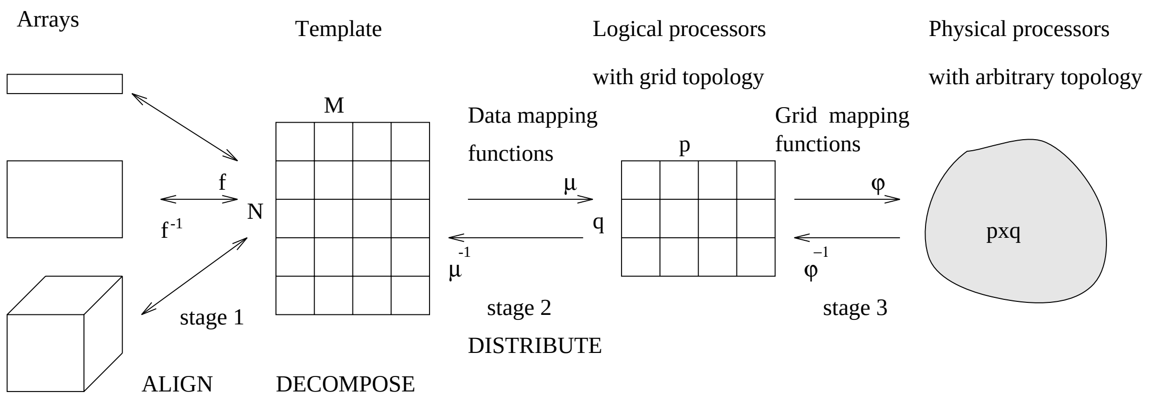


Figure 4.9: Three stage array mapping

Stage 1 : ALIGN directives are processed to compute functions that map the *array index domain* to the *template index domain* and vice versa. Also, the local shape of the arrays it determined.

Stage 2 : Each dimension of the template is mapped onto the logical processor grid based on the distribution directives. Furthermore, mapping functions are computed to generate the relationship between global and local indices.

Stage 3 : The logical processor grid is mapped onto the physical system. This mapping can change from one system to another but the data mapping onto logical processor grid does not need to change. This enhances portability across a large number of architectures.

By performing this three stage mapping, the compiler is decoupled from the specifics of a given machine or configuration.

4.4 Compiling the ALIGN Directive (Stage 1)

Alignment of data arrays to templates is specified by the ALIGN directives. In this section, we describe how the ALIGN directive is processed.

Alignment determines which portions of two or more arrays will be in the same processor for a particular data partitioning. Clearly, if arrays involved in the same computation are aligned so that after distribution their respective sections lie on the same processors, then the number of non-local accesses would be reduced.

Alignment is a relation that specifies a one-to-one correspondence between elements of a pair of array objects. The template is defined by a *DECOMPOSITION* directive with its shape and rank given. Let A be an m-dimensional array and $TEMPL$ be an n-dimensional template. The general form of an alignment directive is:

!F90D$ ALIGN A(i_1**[*], ... ,**i_m**[*]) WITH TEMPL(**$f_1(i_{a_1})$**[*], ... ,**$f_n(i_{a_m})$**[*]).**

The exhibited elements of A are aligned to those of $TEMPL$. The template is eventually distributed on a set of processors. The compiler guarantees that the array elements aligned to the same element of the template will be mapped to the same processor.

The *alignment function* f_k is required to be a linear function $f_k = s_k * i_{a_k} + o_k$ or $f_k = o_k$. The parameters i_{a_k}, s_k, and o_k correspond to the three components of the alignment function: *axis*, *stride*, and *offset*. Misalignment in the axis or stride components causes *irregular communication*, and misalignment in the offset component

```
Algorithm 1 (Compiling Align directives)
Input: Fortran 90D syntax tree with arbitrary alignment functions
Output: Fortran 90D syntax tree with perfect alignment functions
Method: For each aligned array, and for each dimension of that array,
         carry out the following steps
        Step 1. Extend aligned arrays to match template size.
        Step 2. Apply alignment functions to the aligned arrays.
        Step 3. Transform into canonical form.
        Step 4. Compute $f^{-1}(i)$.
```

causes nearest-neighbor communication [46].

Algorithm 1 gives the steps in the algorithm used by Fortran 90D/HPF to process align directives. Algorithm 1 takes Fortran 90D/HPF syntax tree with arbitrary alignment functions, transforms them to perfect alignment functions. That is to transform array indices from the *array index domain* to *template index domain.* The following example illustrates the steps and all the transformations performed to transform by Algorithm 1.

Consider the Fortran 90D/HPF code fragment shown in Figure 10. There are three arrays ODD(N/2), EVEN(N/2) and NUM(N). Elements of the array ODD are aligned with odd elements of TEMPL. Similarly, elements of the array EVEN are aligned with the even elements of TEMPL. NUM is aligned identically with TEMPL which is called perfect alignment. Hence, ODD and EVEN are aligned with odd and even indices of NUM respectively, because they are aligned to the same template.

Step 1. *Extend aligned arrays to match template size.* Note that it is required that the array size is equal to or smaller than the template size in the distributed dimension(s). If an array size is smaller than the template size in the distributed dimension, the compiler extends the array size to match the template size. For

```
PARAMETER(NPROC1=10, N=100)
REAL NUM(N), ODD(N/2), EVEN(N/2)
C$  DECOMPOSITION TEMPL(N)
C$  DISTRIBUTE TEMPL(BLOCK)
C$  ALIGN NUM(I) WITH TEMPL(I)
C$  ALIGN ODD(I) WITH TEMPL(2*I-1)
C$  ALIGN EVEN(I) WITH TEMPL(2*I)
    FORALL(I=1:N:2) NUM(I) = ODD((I+1)/2)
    FORALL(I=2:N:2) NUM(I) = EVEN(I/2)
    LOC=MAXLOC(ODD)
```

Figure 4.10: A Fortran 90D/HPF program fragment.

example, ODD and EVEN arrays are extended to size N to match the template TEMPL's size, which is N. This is a limitation of our compiler.

Step 2. *Apply alignment functions to the aligned arrays.* In this step, all indices of each occurrence of an array, all the statements in the input program are transformed into the *template index domain* using the alignment function f(I). Arrays ODD, EVEN and NUM are associated with the $f_o(I) = 2 * I - 1$, $f_e(I) = 2 * I$, $f_n(I) = I$ functions respectively. Figure 11 illustrates this transformation on the array ODD. For example, the first forall assignment statement in Figure 10:

```
NUM(I)=ODD((I+1)/2)
```

is transformed into

```
NUM(I)=ODD(2*((I+1)/2)-1)  (1)
```

by applying function $f_n(I) = I$ (identical function) and $f_o(I) = 2 * I - 1$ to *lhs* and *rhs* respectively.

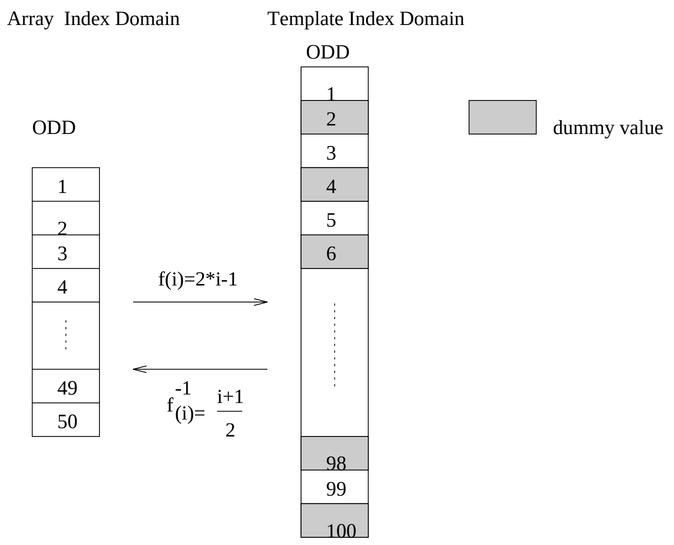


Figure 4.11: Transforming array index domain to template index domain.

Step 3. *Transform into canonical form*[1]. In this step, the compiler simplifies all functions applied in step 3 by performing symbolic manipulation and partial evaluation of constants. For example, the statement (1) becomes:

```
NUM(I)=ODD(I).
```

The above simplification of indices helps the compiler to choose efficient collective communication routines. Our communication detection algorithm [37, 50] is based on symbolically comparing the *lhs* and *rhs* reference patterns and determining if the pattern is associated with one of the predefined collective communication routines. In the above statement the compiler compares the *lhs* and *rhs* indices and determines

[1]A canonical form is a syntactic form in which variables appear in a predefined order and constants are partially evaluated.

that no communication is required because both the array reference patterns are given by I and aligned to the same template. However, if the *rhs* was ODD(I+2), the compiler recognizes the operation as a shift communication.

Step 4. *Compute* $f^{-1}(i)$. For each array, we compute the inverse alignment function $f^{-1}(i)$ corresponding to each $f(i)$. $f^{-1}(i)$ is stored in a *Distributed Array Descriptor* (DAD) [51]. This function is needed when any computation needs to be performed using the original index of an array. For example, the last statement in Figure 10 calls the intrinsic function MAXLOC to find the location of the maximum element in the array ODD. This function must be evaluated using the original array indices. The inverse function for array ODD is $f^{-1}(i) = \frac{i+1}{2}$. MAXLOC returns the location of maximum value in the original *array index domain* by applying the f^{-1} function.

Figure 12 shows the compiler generated Fortran 77+MP code for the Fortran 90D/HPF code given in Figure 10.

We emphasize that the transformation shown in Figure 11 from the *array index domain* to the *template index domain* has two advantages.

1-) This allows the compiler to easily detect regular collective communication patterns among arrays aligned to the same template.

2-) The compiler keeps data distribution functions only for the template and not for all the arrays aligned to the template.

4.5 Data Distribution (Stage 2)

In this section, we describe how the Fortran 90D/HPF compiler distributes the template on the logical processor grid refer back to Figure 9. In this phase, the compiler

```
PARAMETER(NPROC1=10, N=100)
REAL NUM(10), ODD(10), EVEN(10)   ! local shapes
call set_BOUND(lb,ub,st,1,100,2)  ! compute local lb, ub, st
DO I=lb,ub,st
 NUM(I) = ODD(I)                  ! local computations
END DO
call set_BOUND(lb,ub,st,2,100,2)
DO I=lb,ub,st
 NUM(I) = EVEN(I)                 ! local computations
END DO
! put information for ODD into ODD_DAD
call set_DAD(ODD_DAD,....)
! MAXLOC is implemented on f77+MP
LOC=MAXLOC(ODD, ODD_DAD)
```

Figure 4.12: The compiler generated code from Figure 10.

uses information provided by DISTRIBUTE directives.

4.5.1 Distribution functions

A Fortran 90D/HPF program is written in the global name space. Therefore, the arrays and template indices refer to indices in the global name space. Parallelizing the program onto a distributed memory machine requires mapping a global index onto the processor number and local index pair, because on a distributed memory machine, each node has a separate name space. For the above index transformations, we define data-distribution functions , index-conversion functions, as given in Definition 1 below.

Definition 1: *A data-distribution function for each dimension of template μ maps three integers, $\mu(I, P, N) \rightarrow (p, i)$, where I is the global index, $0 \leq I < N$, P is the number of processors, and N is the size of global index. The pair (p, i) represents the processor p, $(0 \leq p < P)$ and i is the local index of p $(0 \leq i < \mu^{\#}(p, P, N))$. $\mu^{\#}(p, P, N)$ gives the cardinality (the number of global indices in processor p). The inverse distribution function $\mu^{-1}(p, i, P, N) \rightarrow I$ transforms the local index i in processor p back into global index I.*

The term *global index* refers to the index of a data item within the global array, global name space, while the term *local index* denotes the index of a data item within a logical processor.

The choice of these distribution functions is one of the most important design choices in the compiler. We use the following criteria:

- calculation of these function at run-time must be efficient.

- distribution functions should yield a good static load balance.

Table 4.1: Data distribution functions

	Block-distribution	Cyclic-distribution
global to proc $I \rightarrow p$	$p = \frac{I*P}{N}$	$p = I \ mod \ P$
global to local $I \rightarrow i$	$i = I - \frac{p*N}{P}$	$i = \lfloor \frac{I}{P} \rfloor$
local to global $(p,i) \rightarrow I$	$I = i + \frac{p*N}{P}$	$I = iP + p$
cardinality	$\frac{N}{P}$	$\lfloor \frac{N+P-1-p}{P} \rfloor$

The **CYCLIC** attribute indicates that global indices of the template in the specified dimension should be assigned to the logical processors in a round-robin fashion. The last column of Table 1 shows the CYCLIC distribution functions. This yields an optimal static load balance since the first $N \ mod \ P$ processors get $\lceil \frac{N}{P} \rceil$ elements; the rest get $\lfloor \frac{N}{P} \rfloor$ elements. In addition, these distribution functions are efficient and simple to compute. Although CYCLIC distribution functions provide a good static load balance, the locality is worse than with block distributions since cyclic distributions scatter data.

4.5.2 Usage of the data distribution functions

The following examples illustrate how the data distribution function can be used for various constructs. For these examples, the array A has the following alignment.

```
C$ DECOMPOSITION TEMPL(N,M)
C$ ALIGN A(I,J) WITH TEMPL(I,J)
C$ DISTRIBUTE TEMPL(CYCLIC,BLOCK)
```

and TEMPL is distributed on a two-dimensional PxQ processor grid.

Example 1 (Masking) Consider the statement:

```
A(5,8)=99.0
```

The owner processor of the array element $A(5,8)$ executes the statement. Since the compiler generates SPMD style code, it masks the rest of the processors:

```
if( 5 mod P .eq. my_id(1) .and. 8*Q/M .eq. my_id(2))
     A(5/P, 8-my_id(2)*M/Q) = 99.0
```

Where my_id(1) and my_id(2) describes the processor's position in the two dimensional logical grid. In this case, the compiler uses the global to processor and global to local functions for cyclic and block distributions. The processors are masked according to the coordinate id numbers since the logical processors are arranged in a grid topology.

Example 2 (Grouping) Consider the statement:

```
A(:,8)=99.0
```

Only, the group of processors owning the 8^{th} column of array A need to execute this statement. The rest of the grid must be masked.

```
do i=my_id(1),N,P
   if(8*Q/M .eq. my_id(2))  A(i/P, 8-my_id(2)*M/Q) = 99.0
end do
```

Note that the iterations (indexed by i above) are distributed cyclicly following the owner computes rule.

Example 3 (Forall) Consider the statement:

```
forall(i=1:N,j=1:M)  A(i,j)=j
```

In the above computations all elements of each column of array A are assigned the corresponding column number (in the global index domain).

```
do i=my_id(1),N,P
do j=1,M/Q
  A(i/P,j)=j+my_id(2)*M/P
end do
end do
```

The compiler distributes the iterations i and j in cyclic and block fashion respectively since array A is distributed in that fashion. Iteration index j is localized. The compiler transforms j back to a global index using local to global index conversion in the *rhs* expression.

Example 4 (Broadcast) Consider the statement:

```
x=A(5,8)
```

where x is a scalar variable (scalars are replicated on all processors). The above statement causes a broadcast communication. The source processor of the broadcast is found using a global-to- processor function similar to that in Example 1.

Example 5 (Gather) Consider the statement:

```
B=A(U,V)
```

where U and V are one-dimensional replicated arrays. B is a two-dimensional array and is distributed in the same way as is array A. This vector-valued assignment causes an unstructured communication (also called *gather*[28] in this case). The owner processors of array B may need some values of array A, depending on the contents of arrays U and V at run-time. The compiler makes each owner processor of array B

calculate which processor has the non-local part of array A using global to processor function. The compiler also generates code that computes the local index the array A using the global to local index conversion function for each source processor. After making each processor calculate the local list and the processor list, the compiler generates a statement to the call gather collective communication.

Example 6 (Scatter) Consider the statement:

```
A(U,V)=B
```

The above statement causes *scatter* communications. Again the compiler generates code such that each owner processor of the array B uses data distribution functions to find the destination of the local array B.

4.6 Grid Mapping Functions (Stage 3)

So far we have presented techniques used in our compiler that map data onto logical processors. In this section we describe the mapping of logical processors onto physical processors.

There are several advantages of decoupling logical processors from physical system configurations. These advantages include *locality*, *portability* and *grouping*.

Locality: Multiple accesses to consecutive memory locations is called *spatial locality*. *Spatial locality* is very important for Distributed Memory Machines. Arrays representing spatial locations are distributed across the parallel computer. For instance, it makes sense to have data distributed in such a way that processors that need to communicate frequently are neighbors in the hardware topology. It has been shown that this is extremely important in the common regular problems in scientific applications such as relaxation [49]. Our template is a d-dimensional mesh. If this template is BLOCK distributed on a d-dimension grid of processors, the neighboring array elements (spatial locality) will be in the neighboring processors. The grid topology is a very good topology for spatial locality. Fortran 90D/HPF makes the logical processor topology grid according to the number of dimensions of the DECOMPOSITION as shown in Figure 13.

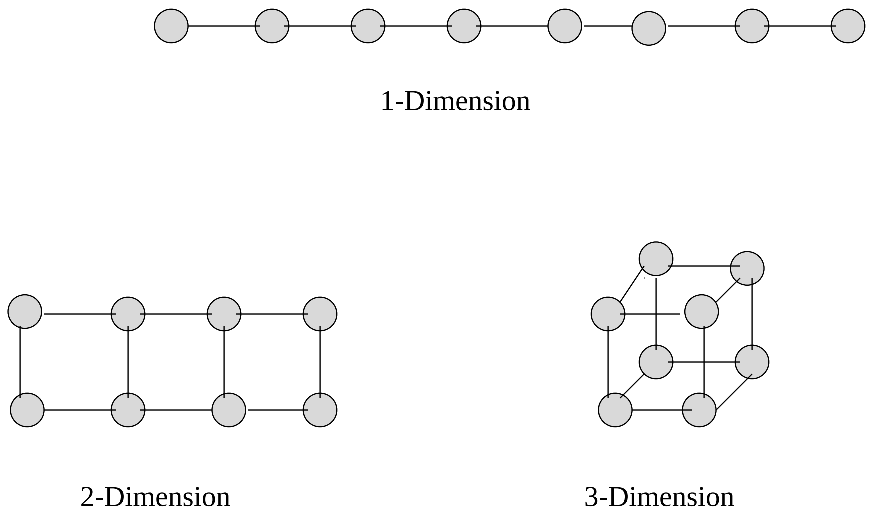


Figure 4.13: Logical processor topologies

Portability: The physical topology of a hardware system may be a grid, a tree, a hypercube or some other layout. The mapping for the best (possible) grid topology changes from one physical topology to another. To enhance portability, we separate the physical and logical topologies. Therefore, porting the compiler from one hardware platform to another involves changing the functions that map the logical grid topology to the target hardware.

Grouping: Operations on a subset of dimensions in arrays are very common in scientific programming, e.g., row and column operations on matrices. Fortran 90D/HPF provides intrinsic functions such as **SPREAD**, **SUM**, **MAXVAL** and **CSHIFT** that let a user specify operations along different dimensions by specifying the **DIM** dimension parameter. These dimensional operations conceptually group elements in the same dimension. The dimensional array operations result in "dimensional array communications". We have designed a set of collective communication routines that operate along one or more dimensions (groups of processors) of the grid. For example, we have developed spread (broadcast along dimension), shift along dimensions and concatenate communications. these primitives are discussed in Chapter 5.

The performance of the resulting code may be adversely affected if the logical grid to physical system mapping is not efficient. Therefore, one of the goals of these mapping functions is to map nearby processors in the logical grid to physically close processors in the machine architecture.

Definition 2: *A logical processor grid consists of d dimensions, $(P_0, P_2, ..., P_i, ..., P_{d-1})$, where P_i, $0 \leq i < d$ is the size of the i^{th} dimension. A processor grid mapping function, φ, maps a processor index in the d-dimensional space, $\varphi(v_0, v_1, ..., v_{d-1}) \rightarrow p$ where $0 \leq v_i < P_i$ (i.e., v_i is the index of the logical processor in the i^{th} dimension), and p is the physical processor number, $(0 \leq p < \prod_{i=0}^{d-1} P_i)$. The inverse mapping function $\varphi^{-1}(p) \rightarrow (v_0, v_1, ..., v_{d-1})$ transform the processor number p back into logical grid number.*

For example, the grid mapping function φ and φ^{-1} for hypercube using Gray Code can be found in [49] and the grid mapping onto a fat tree can be found in [52].

Figure 14 gives some of the grid mapping functions implemented in the Fortran

```
int gridinit(dim, num(*))
int gridproc(coord(*))
int gridcoord(proc, coord(*))
```

Figure 4.14: The prototype of grid-mapping functions

90D/HPF compiler. The first routine, *gridinit*, takes the dimensionality of the grid, *dim*, and the number of physical processors in each dimension as an array, *num* and performs the necessary initializations in order to use the other two grid mapping functions φ and φ^{-1}. The routine *gridcoord* implements the function φ to generate the physical processor number corresponding to the logical processor grid specified in the parameter array "coord(*)". Similarly, the routine *gridproc* implements the function φ^{-1}. Its input parameter "proc" specifies the physical processor id and its output is the corresponding index in the logical grid which is stored in the array "coord(*)". The details of these functions can be found in[49].

The goal of these functions is to enhance portability. The compiler generates all the communication calls based on the logical coordinates of the processors. The communication routines in turn use the above functions to compute the physical processor ids of involved processors. Another important point to note is that by using the logical grid at the compiler level, masking and grouping are performed using logical grid coordinates.

Chapter 5

Communication Model

It is not possible to arrange interconnected data on a distributed memory machine so that all the pieces of data will reside in the processors that need to use them, because one data item may be used in more than one part of a computation by more than one processor. Thus, interprocessor communication may be required. Computations on data structures have a definite mechanism: first, data elements are brought together, then computations are performed. Once the data elements have been brought together, the computations are local. Even on very complex data structures, it is possible to have most of the interacting elements located in the same processor memory. Typically, only a few data items need to be communicated from another processor's memory.

The compiler must recognize the presence of communication patterns in computations in order to generate appropriate communication calls. Specifically, this involves a number of tests on the relationships among subscripts of various array in a statement. We designed an algorithm to detect communications and to generate appropriate collective communication calls to execute array assignments and *forall* statements on distributed memory machines.

This chapter describes the computation partitioning and communication generation phases of the Fortran 90D/HPF compiler. The chapter also describes the run-time support system and storage management used by the Fortran 90D/HPF compiler.

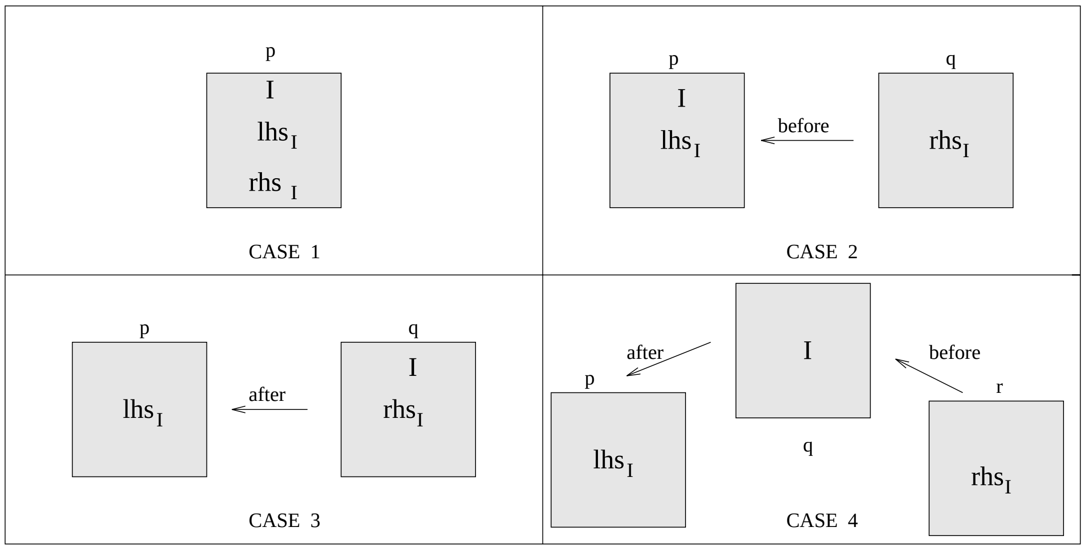
5.1 Computation Partitioning

Once data is distributed, there are several alternatives to assign computations to processing elements for each instance of a *forall* statement. One of the most common methods is to use the *owner computes rule.* Using the owner computes rule, the computation is assigned to the processor owning the *lhs* data element. This rule is easy to implement and performs well in a large number of cases. Most the current implementations of parallelizing compilers uses the owner computes rule [19, 53]. However, it may not be possible to apply the owner computes rule for every case without extensive overhead.

Figure 15 shows the possible data and iteration distributions for the $lhs_I = rhs_I$ assignment caused by iteration instance I. Cases 1 and 2 illustrate the order of communication and computation arising from the owner computes rule. Essentially, all the communications to fetch the off-processor data required to execute an iteration instance are performed before the computation is performed. The generated code will have the following communication and computation order.

```
Communications  ! some global communication primitives
Computation     ! local computation
```

The following examples describe how Fortran 90D/HPF performs computation partitioning.



CASE 1: No communications

CASE 2: Communication before computation to fetch non-local rhs

CASE 3: Communication after computation to store non-local data lhs

CASE 4: Communication before and after computation to fetch and store non-locals

Figure 5.15: Possible computation distribution.

Example 1 (canonical form) Consider the following statement, taken from the Jacobi relaxation program:

```
  forall (i=1:N, j=1:N)
 &   B(i,j) = 0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
```

In the above example, as in a large number of scientific computations, the *forall* statement can be written in the canonical form. In this form, the subscript value in the *lhs* is identical to the *forall* iteration variable. In such cases, the iterations can be easily distributed using the owner computes rule. Furthermore, it is also simpler to detect structured communication by using this form (This will be considered further in Section 5.4.).

Example 2 (non-canonical form) Consider the following statement, taken from an FFT program:

```
  forall (i=1:incrm, j=1:nx/2)
 &   x(i+j*incrm*2+incrm) = x(i+j*incrm*2) - term2(i+j*incrm*2+incrm)
```

The *lhs* array index is not in the canonical form. In this case, the compiler equally distributes the iteration space on the number of processors on which the *lhs* array is distributed. Hence, the total number of iterations will still be the same as the number of *lhs* array elements being assigned. However, this type of *forall* statement results in either Case 3 or Case 4 in Figure 2. The generated code will be in the following order:

```
Communications  ! some global communication primitives to read
Computation     ! local computation
Communication   ! a communication primitive to write
```

For reasonably simple expressions, the compiler transforms such index expressions into the canonical form by performing symbolic expression operations [54]. However, it may not always be possible to perform such transformations for complex expressions.

Example 3 (vector-valued index) Consider the statement

```
forall (i=1:N) A(U(i)) = B(V(i)) + C(i)
```

The iteration i causes an assignment to element $A(U(i))$, where $U(i)$ may only be known at run-time. Therefore, if iterations are statically assigned at compile time to various PEs, iteration i is likely to be assigned to a PE other than the one owning $A(U(i))$. This is also illustrated in cases 3 and 4 of Figure 15. In this case, the compiler distributes the computation i with respect to the owner of $A(i)$.

Having presented the computation partitioning alternatives for various reference patterns of arrays on the *lhs*, we now present a primitive to perform global to local transformations for loop bounds.

```
! computes local lb, ub, st from global ones
set_BOUND(llb,lub,lst,glb,gub,gst,DIST,dim)
```

The *set_BOUND* primitive takes a global computation range with global lower bound, upper bound and stride. It distributes this global range statically among the group of processors specified by the *dim* parameter on the logical processor dimension. The *DIST* parameter gives the distribution attribute such as *block* or *cyclic*. The *set_BOUND* primitive computes and returns the local computation range in local lower bound, local upper bound and local stride for each processor. The algorithm to implement this primitive can be found in [50].

The other functionality of the *set_BOUND* primitive is to mask inactive processors by returning appropriate local bounds. For example, such a case may arise when the global bounds do not specify the entire range of the *lhs* array. If the inputs for this primitive are compile-time constants, the compiler can calculate the local bounds at compile-time.

In summary, our computation and data distributions have two implications.

- The processor that is assigned an iteration is responsible for computing the *rhs* expression of the assignment statement.

- The processor that owns an array element (*lhs* or *rhs*) must communicate the value of that element to the processors performing the computation.

5.2 Why Use Runtime Collective Communication?

Our Fortran 90D/HPF compiler produces calls to collective communication routines [49] instead of generating individual processor send and receive calls inside the compiled code. There are four main reasons for using collective communication to support interprocessor communication in the Fortran 90D/HPF compiler.

1. *Improved performance of Fortran 90D/HPF programs.* To achieve good performance, interprocessor communication time must be minimized. By developing a separate library of interprocessor communication routines, each routine can be optimized. This is particularly important given that the routines will be used by many programs compiled through the compiler.

2. *Increased portability of the Fortran 90D/HPF compiler.* By separating the communication library from the basic compiler design, portability is enhanced because to port the compiler, only the machine specific low-level communication calls in the runtime library need to be changed to support a new system

3. *Improved performance estimation of communication costs.* Our compiler uses the data distribution for the source arrays specified by compiler directives. However, a future compiler may require a capability to perform automatic data distribution and alignment [47, 40, 46]. Such techniques usually require computing trade-offs between exploitable parallelism and the communication costs. The costs of collective communication routines can be determined more precisely, thereby enabling the compiler to generate better distributions automatically.

4. *Reduce the complexity of the compiler.* Interprocessor communication is one of the most error-prone aspects of writing parallel programs. Furthermore, the communication bugs may be intermittent and are difficult to discover, especially when using large numbers of processors. Similar problems may exist if the communication is generated at a low level inside the compiler. Separating the two by developing an independent library of communication routines frees the compiler writer from the complexities of interprocessor communications.

5.3 Communication Primitives

In order to perform a collective communication on array elements, the communication primitive needs the following information

1. send processors list

2. receive processors list
3. local index list of the source array
4. local index list of the destination array

There are two ways of determining the above information. 1) Use a preprocessing loop to compute the above values or, 2) Based on the type of communication, the above information may be implicitly available, and therefore, not require preprocessing. We classify our communication primitives into *unstructured* and *structured* communication.

Our structured communication primitives are based on a logical grid configuration of the processors. Hence, they use grid-based communications such as shift along dimensions, broadcast along dimensions etc. The following summarizes some of the structured communication primitives .

- **transfer:** Single source to single destination message.
- **multicast:** broadcast along a dimension of the logical grid.
- **overlap_shift:** shifting data into overlap areas in one or more grid dimensions. This is particularly useful when the shift amount is known at compile time. This primitive uses that fact to avoid intra processor copying of data and directly stores data in the overlap areas [55].
- **temporary_shift:** This is similar to overlap_shift except that the data is shifted into a temporary array. This is useful when the shift amount is not a compile time constant. This shift may require intra-processor copying of data.

- **concatenation:** This primitive concatenates a distributed array and the resultant array ends up in all the participating processors in this primitive.

The other structured communications in data parallel languages are tree-based communications to perform reduction operations on the specified dimensions of arrays. For example, in Fortran 90D/HPF, the reduction operations on arrays are included as intrinsic functions which can be efficiently hand-coded and supplied as a part of the run-time library for the compiler. Therefore, tree-based communication primitives patterns are not considered in this chapter.

The other advantages of these types of communication primitives are that they can be combined to form composite communication patterns for better performance. (This will be elaborated on in section 5.5.) Further, some structured communication calls can be eliminated using appropriate alignment directives.

Example 1 (Alignment) Consider the following statement:

```
!F90D$   ALIGN A,B with T
     A(1:N-1,1:N-1) = B(2:N,2:N)
```

The above code results in an overlap_shift of array B in two dimensions. However, note that this shift communication may be avoided by the aligning arrays A and B as shown below.

```
!F90D$   ALIGN A(I,J) with T(I,J)
!F90D$   ALIGN B(I,J) with T(I-1,J-1)
```

We have implemented two sets of unstructured communication primitives: One, to support cases where the communicating processors can determine the send and receive lists based only on local information, and hence, only require preprocessing that involves local computations [21], and the other, where to determine the send and receive lists, preprocessing itself requires communication among the processors [27]. The primitives are as follows.

- **precomp_read:** This primitive is used to bring all non-local data to the place it is needed before the computation is performed.
- **postcomp_write:** This primitive is used to store remote data by sending it to the processors that own the data after the computation is performed. Note that these two primitives require only local computation in the preprocessing loop.
- **gather:** This is similar to *precomp_read* except that preprocessing loop itself may require communication.
- **scatter:** This is similar to *postcomp_write* except that preprocessing loop itself may require communication.

5.4 Communication Detection

The compiler must recognize the presence of collective communication patterns in the computations in order to generate the appropriate communication calls. Specifically, this involves a number of tests on the relationship among subscripts of various arrays in a $forall$ statement. These tests should also include information about array alignments and distributions. We use pattern matching techniques similar to those proposed by Marina Chen [37] and also used by Gupta [41]. However, we also need to perform transformation of array indices in order to account for different alignments and distributions. Further, we extend the above tests to include unstructured communication. We also note that we do not expand scalars because scalars are replicated

```
Algorithm 1 (Detecting the communication for the forall statement.)
Input: Forall statement with untagged array and array subscripts
Output: Forall statement with arrays and array subscripts tagged
        with communication primitives.
Method:
1.  for each RHS array do
2.    if (is_aligned_same_template(LHS,RHS)) then
3.      for each subscript $g_i$ of RHS do
4.        find $f_j$ such that $g_i$ and $f_j$ are aligned with the same dimension of a template
5.        if the pair ($f_j$, $g_i$) is in Table 2
            tag the subscript $g_i$ with the corresponding structured communication primitive.
6.      end do
7.    end if
8.    • if an untagged distributed dimension of array reference pattern is in Table 3,
          tag the RHS array with the unstructured primitives
                to read RHS before computation.
9.  end do
10.   • if a distributed dimension of LHS reference pattern is in Table 3
          tag the LHS array with the unstructured primitives
                to write LHS after computation
11.   • if LHS array is not distributed
          tag the distributed RHS array with concatenation primitive.
```

accross processors.

We consider the following *forall* statement to illustrate the steps involved in communication detection.

forall(i1=l1:u1:s1, i2= ..., ...)

LHS(f_1,f_2,...,f_n) = **RHS1**(g_1,g_2,...,g_m) + ...

where g_i and f_j, $1 \leq i \leq m$, $1 \leq j \leq n$ are functions of index variables or indirection arrays. The steps involved in determining a communication pattern are summarized in Algorithm 1.

The algorithm uses two different tables. Table 2 detects structured communication primitives based on the relationship between the *lhs* and *rhs* array subscript reference pattern for BLOCK distribution. Table 3 detects unstructured communication primitives. Tables 2 and 3 use variables to represent c: compile time constant, s: scalar, f: invertible function, V: an indirection array.

The algorithm first attempts to detect structured communication if the arrays are aligned to the same template. For each array on the *rhs*, the following processing is performed. For each subscript of the array, it is coupled with the corresponding subscript on the *lhs* array such that both subscripts are aligned with the same dimension of the template. For each such pair, the algorithm attempts to find a structured communication pattern in that dimension according to Table 2. If a structured communication pattern is found then the subscript on the *rhs* is is tagged, indicating the appropriate communication primitive.

Table 5.2: Structured communication detection.

Steps	(lhs,rhs)	Comm. primitives
1	(i, s)	multicast
2	$(i, i + c)$	overlap_shift
3	$(i, i - c)$	overlap_shift
4	$(i, i + s)$	temporary_shift
5	$(i, i - s)$	temporary_shift
6	(d, s)	transfer
7	(i, i)	no_communication

If any distributed dimension of an array on the *rhs* is left untagged then the array is marked with one of the unstructured communication primitives (the third column of Table 3) depending on the reference pattern given in the second column of Table 3.

Table 5.3: Unstructured communication detection.

Steps	Reference pattern	Comm. primitives to read RHS	Comm. primitive to write LHS
1	$f(i)$	precomp_read	postcomp_write
2	$V(i)$	gather	scatter
3	*unknown*	gather	scatter

The algorithm tags the *lhs* array as *postcomp_write* or *scatter* according to the reference pattern given in Table 3 if one or more of the distributed dimension's subscript is in non-canonical form , or is vector-valued or is *unknown*. Note that any pattern that can not be classified according to Tables 2 or 3, is marked as *unknown* (such as when a subscript involves more than one forall index, e.g $I + J$) so that scatter and gather can be used to parallelize any *forall* statement.

Finally, the algorithm marks the distributed RHS array with the *concatenation* primitive if the LHS array is replicated.

5.5 Communication Generation

Having recognized the type of communication in each dimension of an array for structured communication or each array for unstructured communication in a *forall* statement, the compiler needs to perform the appropriate program transformations. We now illustrate these transformations.

5.5.1 Structured Communication

All the examples discussed below have the following mapping directives.

```
        CHPF$ PROCESSORS(P,Q)
        CHPF$ DISTRIBUTE TEMPL(BLOCK,BLOCK)
```

```
CHPF$ ALIGN A(I,J) WITH TEMPL(I,J)
CHPF$ ALIGN B(I,J) WITH TEMPL(I,J)
```

Example 1 (transfer) Consider the statement:

```
FORALL(I=1:N) A(I,8)=B(I,3)
```

The first subscript of B is marked as *no_communication* because A and B are aligned in the first dimension and have identical indices. The second dimension is marked as *transfer*.

```
1.  call set_BOUND(lb,ub,st,1,N,1)
2.  call set_DAD(B_DAD,.....)
3.  call transfer(B, B_DAD, TMP,src=global_to_proc(8),
                  dest=global_to_proc(3))
4.  DO I=lb,ub,st
5.     A(I,global_to_local(8)) = TMP(I)
6.  END DO
```

In the above code, the *set_BOUND* primitive (line 1) computes the local bounds for computation assignment based on the iteration distribution (Section 5.1). In line 2, the primitive *set_DAD* is used to fill the Distributed Array Descriptor (DAD) associated with array B so that it can be passed to the *transfer* communication primitive at run-time. The DAD has sufficient information for the communication primitives to compute all the necessary information including local bounds, distributions, global shape etc. Note that transfer performs one-to-one send-receive communication based on the logical grid. In this example, one column of grid processors communicate with another column of the grid processors as shown in Figure 16 (a).

Example 2 (multicast) Consider the statement:

```
      FORALL(I=1:N,J=1:M) A(I,J)=B(I,3)
```

The second subscript of B marked as *multicast* and the first as *no_communication.*

```
1.    call set_BOUND(lb,ub,st,1,N,1)
2.    call set_BOUND(lb1,ub1,st1,1,M,1)
3.    call set_DAD(B_DAD,.....)
4.    call multicast(B, B_DAD, TMP,
    &      source_proc=global_to_proc(3), dim=2)
5.    DO I=lb,ub,st
6.    DO J=lb1,ub1,st1
7.       A(I,J) = TMP(I)
8.    END DO
```

Line 4 shows a broadcast along dimension 2 of the logical processor grid by the processors owning elements $B(I,3)$ where $1 \leq I \leq N$ (Figure 16 (b).)

Example 3 (multicast_shift) Consider the statement:

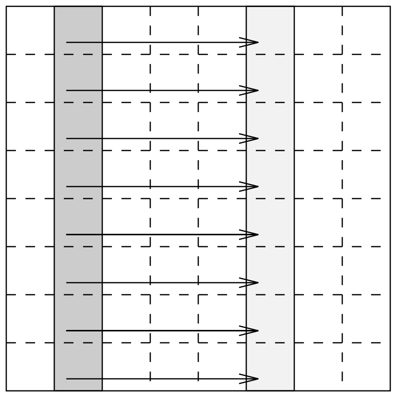
```
      FORALL(I=1:N,J=1:M) A(I,J)=B(3,J+s)
```

The first subscript of array B is marked as *multicast* and the second subscript is marked as *temporary_shift*. The above communication can be implemented as two separate communication steps: multicast along the first dimension of logical grid and *temporary_shift* along the second dimension of the logical grid. Alternatively, the two communication patterns can be composed together to obtain a better communication primitive such as the *multicast_shift* primitive.

```
      call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
      call set_BOUND(lb1,ub1,st1,1,M,1) ! compute local lb, ub, and st
      multicast_shift(B, B_DAD,TMP, source=global_to_proc(3),
    &      shift=s, multicast_dim=1, shift_dim=2)
      DO I=lb,ub,st
```

```
DO J=lb1,ub1,st1
   A(I,J)=TMP(J)
END DO
END DO
```

Combining two primitives eliminates the need for creating temporary storage and eliminates some of intra-processor copying, message-packing, and unpacking.


(a) transfer

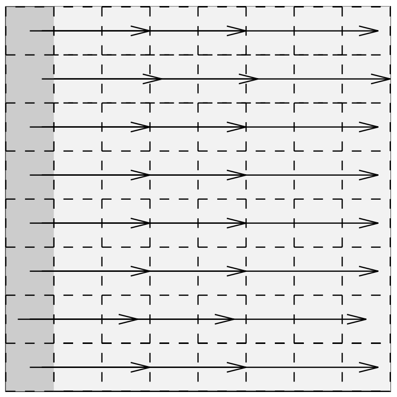

(b) multicast

Figure 5.16: Structured communication on logical grid processors.

5.5.2 Unstructured Communication

In distributed memory MIMD architectures, there is typically a non-trivial communication latency or startup cost. Hence, it is attractive to vectorize messages to reduce the number of startups. For unstructured communication, this optimization can be achieved by performing the entire preprocessing loop before communication so that the schedule routine can combine messages to the maximum extent. The preprocessing loop is also called the "inspector" loop [28, 20].

Example 1 (precomp_read) Consider the statement:

```
FORALL(I=1:N) A(I)=B(2*I+1)
```

The array B is marked as *precomp_read* since the distributed dimension subscript is written as $f(i) = 2 * i + 1$ which is invertible as $g(i) = (i - 1)/2$.

```
 1  count=1
 2  call set_BOUND(lb,ub,st,1,N,1)
 3  DO I=1, N/P
 4    receive_list(count)=global_to_proc(f(i))
 5    send_list(count)= global_to_proc(g(i))
 6    local_list(count) = global_to_local(g(i))
 7    count=count+1
 8  END DO
 9  isch=schedule1(receive_list, send_list,local_list,count)
10  call precomp_read(isch, tmp,B)
11  count=1
12  DO I=1, N/P
13    if((I.ge.lb).and.(I.le.ub).and.(mod(I,st).eq.0))
   &      A(I) = tmp(count)
14    count= count+1
15  END DO
```

The preprocessing loop is given in lines between 1-9. Note that this preprocessing loop executes concurrently in each processor. The loop covers the entire local array bounds since each processor has to calculate the *receive_list* as well as the *send_list* of processors. Each processor also fills the local indices of the array elements which are needed by that processor.

The *schedule1* routine does not need to communicate, it constructs the scheduling data structure *isch*. The schedule *isch* can also be used to carry out identical patterns of data exchanges on several different but identically distributed arrays or array sections. The same schedule can be reused repeatedly to carry out a particular pattern of data exchange on a single distributed array. In these cases, the cost of generating the schedules can be amortized by only executing it once. This analysis can be performed at compile time. Hence, if the compiler recognizes that the same schedule can be reused, it does not generate code for scheduling, it passes a pointer to the already existing schedule. Furthermore, the preprocessing computation can be moved up as much as possible by analyzing *definition-use chains* [56]. Reduction in communication overhead can be significant if the scheduling code can be moved out of one or more nested loops by this analysis. In the above example, *local_list* (line 6) is used to store the index of one-dimensional array. However, in general, *local_list* will store indices from a multi-dimensional Fortran array using the usual column-major subscript calculations to map the indices to a one-dimensional index.

The *precomp_read* primitive performs the actual communication using the schedule. Once the communication is performed, the data is ordered in a one dimensional array, and the computation (lines 12-15) uses this one dimensional array. The *precomp_read* primitive brings an element into *temp* for each local array element since preprocessing loops covers entire local array. The *if* statement masks the assignment to preserve the semantics of the original loop.

Example 2 (gather) Consider the statement

```
FORALL(I=1:N) A(I)=B(V(I))
```

The array B is marked as requiring *gather* communication since the subscript is only known at runtime. The receiving processors can know what non-local data they need from other processors, but a processor may not know what local data it needs to send to other processors. For simplicity, in this example, we assume that the indirection array V is replicated. If V is not replicated, it must also be communicated to compute the receive list on each processor.

```
count=1
call set_BOUND(lb,ub,st,1,N,1)
DO I=lb,ub,st
  receive_list(count)=global_to_proc(V(i))
  local_list(count) = global_to_local(V(i))
  count=count+1
END DO
isch = schedule2(receive_list, local_list, count)
call gather(isch, tmp,B)
count=1
DO I=lb,ub,st
  A(I) = tmp(count)
  count= count+1
END DO
```

Once scheduling is completed, every processor knows exactly which non-local data elements it needs to send to and receive from other processors. Recall that the task of *scheduler2* is to determine exactly which send and receive communications must be carried out by each processor. The scheduler first figures out how many messages each processor will have to send and receive during the data exchange. Each processor computes the number of elements (*receive_list*) and the local index of each element it needs from all other processors. In *schedule2* routine, processors communicate to combine these lists (a fan-in type of communication). At the end of this processing, each processor contains the send and receive list. After this point, each processor transmits a list of required array elements (*local_list*) to the appropriate processors. Each processor now has the information required to set up the send and receive messages that are needed to carry out the scheduled communication. This is done by the gather primitives. Note that *schedule1* does not need to communicate to form scheduling unlike schedule2.

Example 3 (scatter) Consider the statement

```
FORALL(I=1:N) A(U(I))=B(I)
```

The array A is marked as requiring *scatter* primitive since the subscript is only known at runtime. Note that the owner computes rule is not applied here. The processor performing the computation knows the processor and the corresponding local-offset where the resultant element must be written.

```
count=1
call set_BOUND(lb,ub,st,1,N,1)
DO I=lb,ub,st
  send_list(count)=global_to_proc(U(i))
  local_list(count) = global_to_local(U(i))
  count=count+1
 END DO
 isch = schedule3(proc_to, local_to, count)
 call scatter(isch, A, B)
```

Unlike the *gather* primitive, each processor computes a *send_list* containing processor ids and a *local_list* containing the local index where the communicated data must be stored. The *schedule3* routine is similar to *schedule2* of the gather primitive except that schedule3 does not need to send local index in a separate communication step.

The gather and scatter operations are powerful enough to provide the ability to read and write distributed arrays with a vectorized communication facility. These two primitives are available in PARTI (Parallel Automatic Runtime Toolkit at ICASE) [28] which is designed to efficiently support irregular patterns of distributed array accesses. The PARTI and other communication primitives and intrinsic functions form the unstructured run-time support system of our Fortran 90D/HPF compiler.

5.6 Run-time Support System

The Fortran 90D/HPF compiler relies on a powerful run-time support system. The run-time support system consists of functions which can be called from the node programs of a distributed memory machine. Runtime system efficiently support the address translations and data movements that occur when mapping a shared address space program onto a multiple processor architecture.

Intrinsic functions support many of the basic data parallel operations in Fortran 90. The intrinsics not only provide a concise means of expressing operations on arrays, but also identify parallel computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, and matrix multiplication. The intrinsic functions that may induce communication can be divided into five categories as shown in Table 4.

The first category requires data to be transferred using the low overhead structured shift communications operations. The second category of intrinsic functions require computations based on local data followed by use of a reduction tree on the processors involved in the execution of the intrinsic function. The third category uses multiple broadcast trees to spread data. The fourth category is implemented using

Table 5.4: Fortran 90D/HPF Intrinsic Functions

1. Structured communication	2. Reduction	3. Multicasting	4. Unstructured communication	5. Special routines
CSHIFT EOSHIFT	DOTPRODUCT ALL, ANY COUNT MAXVAL, MINVAL PRODUCT SUM MAXLOC, MINLOC	SPREAD	PACK UNPACK RESHAPE TRANSPOSE	MATMUL

unstructured communication patterns. The fifth category is implemented using existing research on parallel matrix algorithms [49]. Some of the intrinsic functions can be further optimized for the underlying hardware architecture. Fortran 90D/HPF compiler includes more than 500 parallel run-time support routines. Fortran 90D/HPF run-time is written with Fortran 77 language. If the run-time is implemented with C language, the number of run-time routines may reduce drastically. The run-time implementation details can be found in [51].

Arrays may be redistributed across subroutine boundaries. A dummy argument which is distributed differently than its actual argument in the calling routine is automatically redistributed upon entry to the subroutine, and is automatically redistributed back to its original distribution at subroutine exit. These operations are performed by the redistribution primitives which transform from *block* to *cyclic* or vice versa.

When a distributed array is passed as an argument to some of the run-time support primitives, it is also necessary to provide information such as its size, its distribution among the nodes of the distributed memory machine, and other information. All this information is stored into a structure which is called the *distributed array descriptor*

(DAD) [51]. DADs pass compile-time information to the run-time system and information between run-time primitives. The run-time primitives query alignment and distribution information from a DAD and act upon that information.

The basic layer of a run-time system should be a portable message passing system like Express [57], PVM [58], MPI [59] or PARMACS [60]. Only this approach guarantees the portability of the HPF compiler accross many different platforms. PARMACS is based on the host-node style programming which is not god for Fortran 90D/HPF compilation. PVM are avaliable for nearly all machines, but functinality and efficency is rather low. MPI support many different communication modes, especially blocking and non-blocking communication.

Our run-time library uses the Express parallel programming environment [57] as a message passing communication primitives. The Express parallel programming environment [57] guarantees a level of portability on various platforms including, Intel iPSC/860, nCUBE/2, networks of workstations etc. We choose the Express because it was the avaliable one at the time of Fortran 90D/HPF implementation.

In summary, parallel intrinsic functions, communication routines, dynamic data redistribution primitives and other primitives and routines are part of the run-time support system.

5.7 Storage Management

Data-parallel scientific codes generally require tremendous amounts of memory. In addition to speed-up, this is one of the reasons to run scientific code on distributed memory parallel machines. Besides the user's data, an HPF compiler must create storage for several reasons:

1. When an array expression is passed to a subroutine, storage must be created to hold the value of the expression.

2. When an array-valued function is used in an expression, storage must be created to hold the return value of the function.

3. When a *forall* statement, if scalarized, would carry a dependency, storage must be created to hold the value of the right hand side.

4. When a transformational function is referenced within a *forall*, storage must be created to hold the result of the transformational function.

5. The communications strategy requires creation of storage for nonlocal array references.

The simplest approach to storage management may allocate full-sized arrays on each processor for all the above cases but this strategy could waste tremendous amount of memory. The compiler should use sophisticated storage management techniques to reduce memory use. Fortran 90D/HPF uses two different storage allocation techniques, *overlap areas* and *temporary arrays.*

Overlap areas are expansions of local array sections to accommodate neighboring nonlocal elements. Overlaps are useful for regular computation because they allow the generation of clear and readable code. However, for certain computations storage may be wasted because all array elements between the local section and the one accessed must also be part of the overlap. Storage is also wasted because overlaps are assigned to individual arrays, and cannot be reused or released until arrays have completed their lifetimes. Our compiler uses the overlap storage for *overlap_shift* communication for small shift values.

Temporary arrays are another form of storage. A temporary array is usually aligned and distributed in the same manner as one of the user variables; that is the HPF program could be written in such a way that none of these temporaries would be needed. The algorithm used by the compiler to determine distribution of temporaries takes the statement in which the temporary is used into account. Temporaries are allocated before the statement in which they are used, and deallocated immediately after that statement. For example, an array assignment like:

```
REAL  A(N), B(N), C(N), D(N)
        A = SUM(B, DIM=1) + MATMUL(C,D)
```

would result in the following:

```
allocate (tmp$b)
allocate (tmp$r)
call sum(tmp$b, b, 1)
call matmul(tmp$r, c, d)
a = tmp$b + tmp$r
deallocate(tmp$b)
deallocate(tmp$r)
```

For this class of temporaries, distribution of a temporary is determined depending on how the temporary is used. If a temporary is used as the argument to an intrinsic, the compiler tries to determine its distribution based on the other intrinsic arguments. Otherwise, it tries to assign a distribution based on the value assigned to the temporary. Otherwise, the temporary is replicated.

The above algorithm is very simple and is certainly not optimal. However, it is not clear what algorithm would perform better. Numerous factors, including array alignment, array distribution, array subsection usage, and argument usage need to be taken into account in determining temporary distribution. For example, consider the following case:

```
A(1:m:3) = SUM(B(1:n:2,:) + C(:,1:n:2), dim=2)
```

The section of A is passed directly to the SUM intrinsic to receive the result. A temporary is needed to compute the argument to SUM. The distribution of that temporary has two possibly conflicting goals: minimize communication in the $B + C$ expression, or minimize communication in the SUM computation and assignment to A.

Chapter 6

Optimizations

Performance for Fortran 90D/HPF programs and their communications on any particular parallel system is influenced by several factors including the amount of communications required by a program for computation and for overhead and the system's latency and bandwidth where communication is required. Another factor that influences performance is the number and power of optimizations performed to improve or eliminate communications.

Our compiler performs several optimizations to reduce the total cost of communication. There are several optimizations that can be applied to communications that are not present in the prototype compiler that are intended for future releases.

This chapter introduces a number of optimization techniques used by the Fortran 90D/HPF compilation system. The chapter applies some of these optimizations onto Gaussian Elimination code written in Fortran 90D/HPF to show the effectiveness of optimizations.

6.1 Single Node Parallelism

In terms of *computation* optimization, it is expected that the scalar node compiler performs a number of classic scalar optimizations within basic blocks. These optimizations include common subexpression elimination, copy propagation (of constants, variables, and expressions), constant folding, useless assignment elimination, and a number of algebraic identities and strength reduction transformations. However, to use parallelism within the single node (e.g. using attached vector units), our compiler propagates the information to the node compiler using node directives. Since, in the original data parallel constructs such as the *forall* statement, there is no data dependency between different loop iterations, vectorization can be performed easily by the node compiler.

6.2 Communication Hierarchy

Communication is divided into a hierarchy of types, with the lowest types in the hierarchy being more expensive. The hierarchy is as follows:

1) No communication. The left and right hand side arrays reside on the same processor.

2) Overlap shift. A BLOCK-distributed array is involved in a shift communication pattern. The local array section is enlarged by the shift amount, and the boundary elements are transferred from neighboring processors.

3) Collective communications. Two arrays, aligned to the same template, are involved in a multicast, transfer, or variable shift pattern.

4) Pre_read and pre_write. An array is indexed with an arbitrary subscript, but

the evaluation of the subscript does not involve any communication. For example, this primitive can efficiently handle the transpose or diagonal accesses that may arise in a *forall* statement.

5) gather and scatter. An array is indexed with a subscript that involves communication.

6) Scalarization. No efficient communication pattern was found, so every processor performs the entire loop, broadcasting the data that it owns one element at a time, and storing the results that it owns.

6.3 Vectorized Communication

One of the important considerations for message passing on distributed memory machines is the setup time required for sending a message. Typically, this cost is equivalent to the sending cost of hundreds of bytes. Vectorization combines messages for the same source and destination into a single message to reduce this overhead [17, 61] Since in Fortran 90D/HPF we are only parallelizing array assignments and *forall* loops, there is no data dependency between different loop iterations. Thus, all the required communication can be performed before or after the execution of a loop on each of the processors involved as shown in Figure 17.

6.4 Overlap Shift Communications

The overlap shift communications optimization recognizes computations with arrays that contain an overlap pattern as shown in table 6-1. When the array involved in an overlap shift computation is allocated the overlap area [55] is also allocated and remains available until a computation requiring an overlap shift area. Immediately

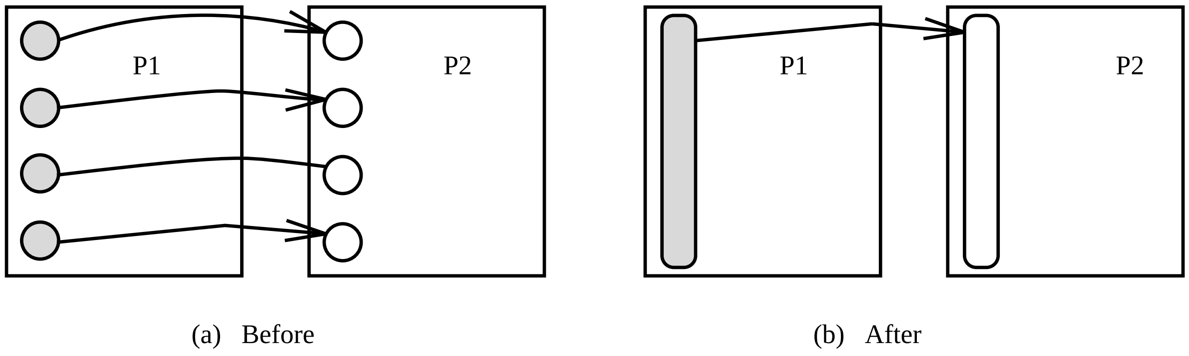


Figure 6.17: Message Vectorization

before the computation, the overlap shift area is filled with the current value(s) of the overlap data (this requires communications). By allocating an overlap shift area, the compiler localizes a portion of a computation prior to the computation that would otherwise require communication during the computations. Figure 18 graphically shows the overlap shift optimization for code similar to the following.

```
      PROGRAM TEST_OVERLAP
      INTEGER I, A(8), B(8)
!F90D$ DECOMPOSITION T(8)
!F90D$ ALIGN A(J) WITH T(J)
!F90D$ ALIGN B(J) WITH T(J)
!F90D$ DISTRIBUTE T(BLOCK)
      FORALL(I=1:7) A(I)=B(I+1)
```

In the first stage of the overlap shift communication, the compiler determines that a computation involving the array B requires an overlap shift area in the positive direction (Fortran 90D/HPF also permits negative overlaps shift areas). A portion of B is then allocated with the extra overlap location(s).

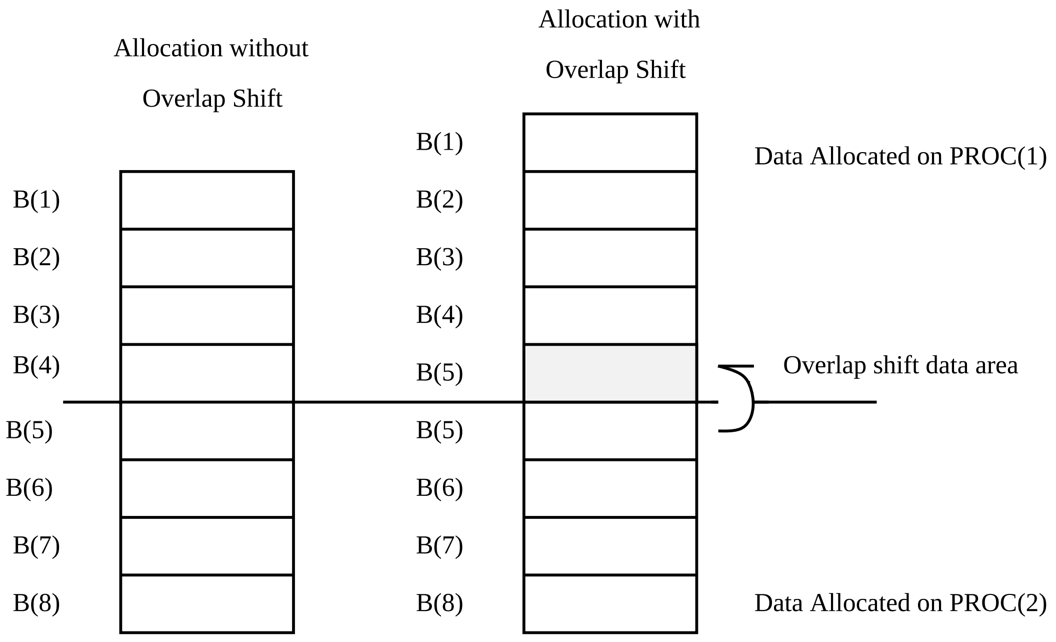


Figure 6.18: Sample Overlap Shift Optimization

6.5 Message Aggregation

The communication library routines try to aggregate messages [37, 17, 61] (corresponding to several array sections) into a single larger message, possibly at the expense of extra copying into a continues buffer. A communication routine first calculates the largest possible array section from this processor to the rest. These may indicate several continuous block of data. Then, it tries to sort the continuous data by destination. Then it aggregates non-continuous array sections (messages) into a continuous message buffers. Messages with an identical destination processor can then be collected into a single communication operation as shown at Figure 19. The gain from message aggregation is similar to vectorization in that multiple communication operations can be eliminated at the cost of increasing the message length.

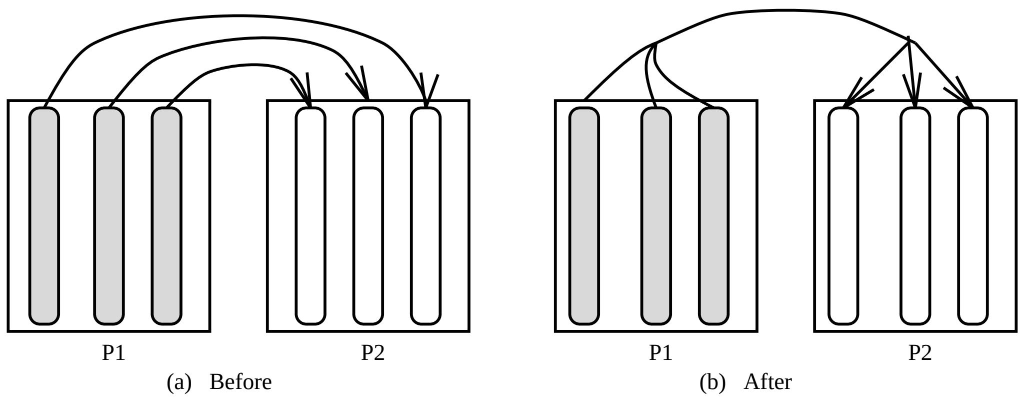


Figure 6.19: Message Aggregation

6.6 Evaluating Expression

Another optimization Fortran 90D/HPF performs involves how expressions are evaluated. For example, consider the following program fragment:

```
      REAL A(N), B(N), C(N), D(N), E(N)
!HPF$ distribute E(BLOCK)
!HPF$ distribute (CYCLIC) :: A, B, C, D
      E = (A + B) * (C + D)
```

In a compiler that blindly applies the owner computes rule, A, B, C and D will be redistributed into temporaries that match the distribution of E, then the computation will be performed locally. This may cause four different communications with four different temporaries.

A more intelligent approach might perform the sum of A and B locally into a cyclically distributed temporary, then perform the sum of C and D locally into another cyclically distributed temporary. Then multiply those two temporaries locally into a cyclically distributed a temporary, finally, redistribute the result into E. This approach may cause one communication with three temporaries (shown in Figure 20(b)).

To apply the above optimization, the compiler has to evaluate the expression according the partial order induced by the expression tree (shown Figure 20(a)). However, Li and Chen [40] show that the problem of determining an optimal static alignment between the dimension of distinct arrays arrays is NP-complete. Chatterjee *et al.* [62] and Bouchitte *et al.* [63] propose some heuristic algorithms to evaluate the expression tree with minimal cost.

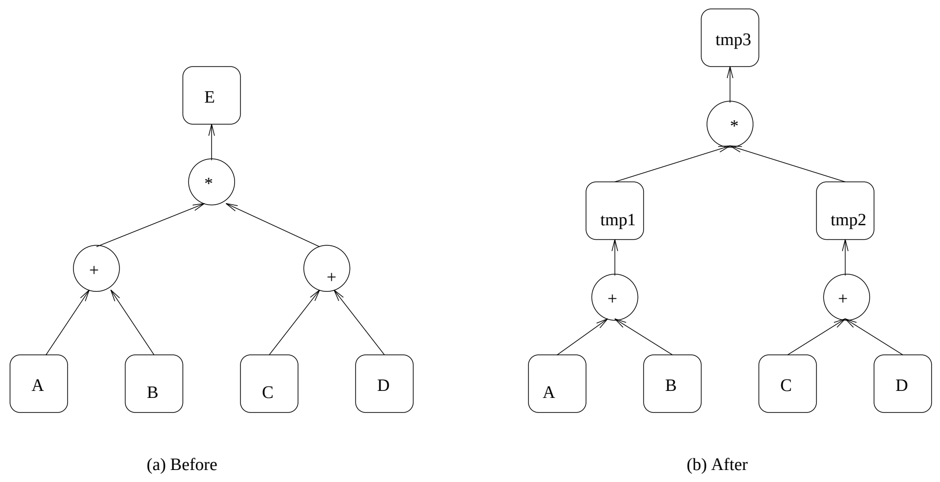


Figure 6.20: Expression Evaluation Trees

6.7 Communication Parallelization

When a block of source data is replicated, any or all of the processors owning it can take part in the communication. Alternatively, one of the source processors is chosen to send to all processors owning the destination block. Ideally, the sends should be spread out over as many source processors as possible to utilize as full available communication bandwidth (shown in Figure 21). This idea is observed by Mark Young [64].

The basic idea is to somehow divide the set of destination processors among the set of source processors. Each source would do a multicast to its assigned subset of destinations. The source and destination sets are computable from information in the template and processor data structures.

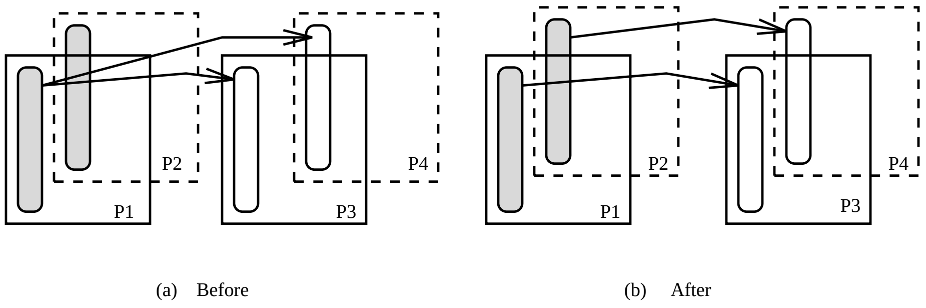


Figure 6.21: Parallel Communication

6.8 Communications Union

In many cases, communication required for two different operands can be replaced by their union. Clearly, the advantage of communication union is that it reduces the number of communication statements and thus the number of messages. For example, the following code may require two *overlapping_shifts*. However, with simple analysis, the compiler can eliminate the shift of size 2.

```
FORALL(I=1:N) A(I)=B(I+2)+B(I+3)
```

The communication union optimization can be applied in a statement as well as inter statement. The compiler needs data-flow analysis infra-structure to perform inter statements communication union.

6.9 Eliminate Unnecessary Communications

In many cases, some portion of distributed arrays are used more than once with the same pattern. The compiler can detect that two references to an array section that has the same pattern if between those references the array is unaltered. The compiler may eliminate extra communications and use the first communication's temporary instead of the second communication. Again this optimization can be done in a statement and on as inter statements basis. Section 6.15 shows a Gaussian Elimination sample program using the eliminate unnecessary communications optimization.

6.10 Reuse of scheduling information

Unstructured communication primitives are required by computations which require the use of a preprocessor. As discussed in Section 5.5.2, the schedules can be reused with appropriate analysis [21, 26] The communication routines perform sends and receives according the scheduling data structure called *isch*.

The schedule *isch* can also be used to carry out identical patterns of data exchanges on several different but identically distributed arrays or array sections. The same schedule can be reused repeatedly to carry out a particular pattern of data exchange on a single distributed array. In these cases, the cost of generating the schedules can be amortized by only executing the schedule generation routine once. This analysis can be performed at compile time. Hence, if the compiler recognizes that the same schedule can be reused, it does not generate code for scheduling but it passes a pointer to an already existing and saved schedule. Furthermore, the preprocessing computation can be moved up as much as possible by analyzing *definition-use chains*

[56]. Reduction in communication overhead can be significant if the scheduling code can be moved out of one or more nested loops.

6.11 Code movement

The compiler try to move up some communication routines by analyzing *definition-use chains* [56] as much as possible . This may lead to moving of the scheduling code out of one or more nested loops which may significantly reduce the amount of overhead. The code movement may also vectorize some of communication. For example, the following loop can not be written as an array construct or a *forall* statement because the loop contains the user defined function *FOO*.

```
DO i=1, N-3
   B(i) = A(i+3) + FOO(C(:,i))
ENDDO
```

The compiler may communicate the array element inside the loop. However, if it applies the optimization, the code becomes:

```
tmp(1:N-3) = A(4:N)
DO i=1, N-3
   B(i) = tmp(i) + FOO(C(:,i))
ENDDO
```

The communication from A to B is taken outside of loop. And it is vectorized.

6.12 Forall Dependency

A *forall* dependency implies that the compiler must generate a temporary to preserve the *forall* semantic. Temporaries are needed to store dependent expressions (mask and the *rhs* of assignment statements). This means two temporaries for the worst case.

The compiler checks to see if the *lhs* array does not appear in any of the expressions, or if it appears and it has the same subscript every where. In such a case, the compiler assumes that the *forall* is not dependent. This is very simple but effective dependency test.

Even if there is a dependency, the Fortran 90D/HPF compiler performs a *forall* dependency analysis after communication generation. Because communication phase may create temporaries, for communication. This may eliminate dependency unintentionally.

If *lhs* and *rhs* array has the same subscripts, that means no dependency and no communication. If subscripts are different, there will be communication, communication will create temporary so it eliminates dependency. This means that most of the time no temporary will be created.

6.13 Forall Loop Interchange

Loop interchange is a transformation that exchanges two levels of a nested loop. Loop interchange rearranges the execution order of the statement instances associated with a loop. Loop interchange is one of the most powerful restructuring transformations [65]. It may be used to enhance *vectorization*, *parallelization* and *memory access* of DO-loops. Since *forall* is already vectorized and parallelized, our compiler uses loop interchange to improve memory accesses. The transformation is valid since *forall* semantics do not specify execution order.

Fortran language stores array in usual column-major order. The Fortran 90D/HPF compiler orders the *forall* triplets according to the column major. For example,

```
          forall(i=1:N,j=1:N) a(i,j) =  b(i, j)
```

This should be written

```
      do j=..
         do i=..
            a(i,j) = b(i, j)
         enddo
      enddo
```

The rule is that the first index of *lhs* will be written the last. This can be used to reduce bank conflicts, to enhance efficiency, and to decrease the number of page faults in a virtual memory system by improving the locality of programs.

6.14 Forall Mask Insertion

Recent parallel computers such as Intel Paragon and Thinking Machine CM-5 have vector units for each processor. However, most vector units does not perform well if the loop has a branch instruction in the loop. Such loops may not be vectorized. A *forall* mask may cause the generated loop to be un-vectorizable. Fortran 90D/HPF compiler tries to insert the mask with only depended indices not all indices. For example, Gaussian Elimination code has a forall:

```
   forall (i = 1:N, j = k:NN, indx(i) .EQ. -1)
  &        a(i,j) = a(i,j) - fac(i)*row(j)
```

Here, mask does not depend on the j index, it only depends on i, so we transform as follows:

```
do i=..
if(index(i).eq.-1) then
   do j=..
      a(i,j)=....
   enddo
endif
enddo
```

The inner loop becomes a vectorizable loop.

6.15 An Example Program for Optimization

We use Gaussian Elimination (GE) with partial pivoting as an example for translating a Fortran90D/HPF program into a Fortran+MP program to show effectiveness of communication elimination optimization. The Fortran90D/HPF code is shown in Figure 22. Arrays a and row are partitioned by compiler directives. The second dimension of a is block-partitioned, while the first dimension is not partitioned. Array row is block-partitioned. This program illustrates now the programmer of working in Fortran 90D/HPF may easily parallelize a program. Data parallelism is concisely represented by array operations, while the sequential computation is expressed by *do* loops. More importantly, explicit communication primitives do not need to be added since the program is written for a single address space using Fortran 90D/HPF.

Figure 23 shows how the Fortran 90D/HPF compiler translates the GE code into Fortran 77+MP form. It is easy to see that the generated code is structured as alternating phases of local computation and global communication. Local computations consist of operations by each processor on the data in its own memory. Global communication includes any transfer of data among processors. The compiler partitions the distributed arrays into small sizes and the parallel constructs are sequentialized

```
      integer, dimension(N) :: indx
      integer, dimension(1) :: iTmp
      real, dimension(N,NN) :: a
      real, dimension(N) :: fac
      real, dimension(NN) :: row
      real :: maxNum
C$  PROCESSORS  PROC(P)
C$  DECOMPOSITION TEMPLATE(NN)
C$  DISTRIBUTE TEMPLATE(BLOCK)
C$  ALIGN row(J) WITH TEMPLATE(J)
C$  ALIGN a(*,J) WITH TEMPLATE(J)

      indx = -1
      do k = 0, N-1
        iTmp = MAXLOC(ABS(a(:,k)), MASK = indx .EQ. -1)
        indxRow = iTmp(1)
        maxNum = a(indxRow,k)
        indx(indxRow) = k
        fac = a(:,k) / maxNum

        row = a(indxRow,:)
        forall (i = 0:N-1, j = k:NN-1, indx(i) .EQ. -1)
   &      a(i,j) = a(i,j) - fac(i) * row(j)
      end do
```

Figure 6.22: Fortran 90D/HPF code for GE.

into *do* loops.

The compiler generates the appropriate communication primitives depending on the reference patterns of distributed arrays. For example, the statement:

```
temp = ABS(a(:,k))
```

is transformed into a broadcast primitive since the array *a* is distributed in the second dimension. All runtime routines are classified according to data types. For example, *_R* specifies the data type of the communication as real and *_V* specifies that it is vector communication. The primitive set_DAD is used to fill the Distributed Array Descriptor (DAD) associated with array *a* so that it can be passed to the *broadcast* communication primitive at run-time. The DAD has sufficient information for the communication primitives to compute all the necessary information including local lower and upper bounds, distributions, local and global shape etc. In this way the communication routines also has an option to combine messages for the same source and destination into a single message to reduce communication overhead. This is the typical characteristic of our compiler since we are only parallelizing array assignments and *forall* statements in Fortran 90D/HPF, there is no data dependency between different iterations. Thus, all the required communication can be performed before or after the execution of the loop on each of the processors involved.

The program code shows how the intrinsic function *MAXLOC* is translated into the library routine *MaxLoc_R_M*. The suffix *_R* specifies the data type and *_M* specifies that *MAXLOC* intrinsic has an optional mask array. Once again the array information passed to the run-time system with the associated *_DAD* data structure.

Fortran 90D/HPF may perform several optimizations to reduce the total cost of communication. The compiler can generate better code by observing the following

```
B= NN/P
integer indx(N)
real aLoc(N,B)
real fac(N)
real rowLoc(B)
real maxNum
integer source(1)

call grid_1(P)
do i = 1, N
  indx(i) = -1
end do
do k = 1, N
    do i = 1, N
      mask(i) = indx(i) .EQ. -1
    end do
    source(1)=(k-1)/B
    call set_DAD_2(aLoc_DAD, 1, N, N, k-my_id()*B, k-my_id()*B, B)
    call set_DAD_1(temp1_DAD, 1, N, N)
    call broadcast_R_V (temp1, temp1_DAD, aLoc, aLoc_DAD, source)
    call set_DAD_1(temp1_DAD, 1, N, N)
    call set_DAD_1(mask_DAD, 1, N, N)
    indxRow = MaxLoc_1_R_M(temp1, temp1_DAD, N, mask, mask_DAD)
    source(1)=(k-1)/B
    call broadcast_R_S(maxNum, aLoc (indxRow, k-my_id()*B), source)
    indx(indxRow) = k
    call set_DAD_2(a_DAD, 1, N, N, k-my_id()*B, k-my_id()*B, B)
    call set_DAD_1(temp2_DAD, 1, N, N)
    call broadcast_R_V (temp2, 1, temp2_DAD, aLoc, 2, aLoc_DAD, source)
    do i = 1, N
       fac (i) = temp2 (i) / maxNum
    end do
    do i = 1, 10
       rowLoc (i) = aLoc (indxRow, i)
    end do
```

Figure 6.23: Fortran 90D/HPF compiler generated Fortran77+MP code for GE.

```
          call set_BOUND(k, NN, llb, lub, 1, B)
          do j = llb, lub
             do i = 1, N
                if (indx (i) .EQ. (-1)) THEN
                   aLoc (i, j) = aLoc (i, j) - fac (i) * rowLoc (j)
                end if
             end do
          end do
       end do
```

Figure 23: Fortran 90D/HPF compiler generated Fortran77+MP code for GE (cont.)

from Figure 22:

```
15.         Tmp = ABS(a(:,k))
17.         maxNum = a(indxRow,k)
19.         fac = a(:,k) / maxNum
```

The distributed array section $a(:,k)$ is used at lines 15,17 and 19. The array a is not changed between lines 15-19. Each statement causes a broadcast operation. Because the compiler performs statement level code generation for the above three statements. However, the compiler can eliminate two of three communication calls by performing the above dependency analysis. The compiler needs only generate one broadcast for line 15 which communicates a column of array a. The Lines 17 and 19 can use the same data. The optimized code is shown in Figure 24. We generated this program by hand since the optimizations have not yet been implemented in the Fortran 90D/HPF compiler. Currently our compiler performs statement level optimizations. It does not perform basic-block level optimizations.

To validate the performance of optimization on GE which is a part of the FortranD/HPF benchmark test suite [66], we tested three codes on the iPSC/860 and plotted the results. 1-) The code is given Figure 23. This is shown as the dotted line in Figure 25, and represent the compiler generated code. 2-) The code in Figure 24. This appears as the dashed line in Figure 25 and represents the hand optimized code on the compiler generated code as discussed above. 3-) The hand-written GE with Fortran 77+MP. This appears as the solid line in Figure 25. The code is written outside of the compiler group at NPAC to be unbiased.

The programs were compiled using Parasoft Express Fortran compiler which calls Portland Group if77 release 4.0 compiler with all optimization turned on (-O4). We can observe that the performance of the compiler generated code is within 10% of the hand-written code. This is due to the fact that the compiler generated code produces extra communication calls that can be eliminated using optimizations. The hand optimized code gives performance close to the hand-written code. From this experiment we conclude that Fortran 90D/HPF compiler, incorporated with node-level optimizations, can compete with hand-crafted code on some significant algorithms, such as GE.

```
 do k = 1, N
     do i = 1, N
       mask(i) = indx(i) .EQ. -1
     end do
     source(1)=(k-1)/B
     call set_DAD_2(aLoc_DAD, 1, N, N, k-my_id()*B, k-my_id()*B, B)
     call set_DAD_1(temp1_DAD, 1, N, N)
     call broadcast_R_V (temp1, temp1_DAD, aLoc, aLoc_DAD, source)
     call set_DAD_1(temp1_DAD, 1, N, N)
     call set_DAD_1(mask_DAD, 1, N, N)
     indxRow = MaxLoc_1_R_M(temp1, temp1_DAD, N, mask, mask_DAD)
     maxNum=temp1(indxRow)
     indx(indxRow) = k
     do i = 1, N
        fac (i) = temp1 (i) / maxNum
     end do
     do i = 1, 10
        rowLoc (i) = aLoc (indxRow, i)
     end do
     call set_BOUND(k, NN, llb, lub, 1, B)
     do j = llb, lub
        do i = 1, N
           if (indx (i) .EQ. (-1)) THEN
              aLoc (i, j) = aLoc (i, j) - fac (i) * rowLoc (j)
           end if
        end do
     end do
end do
```

Figure 6.24: GE with communication elimination optimization.

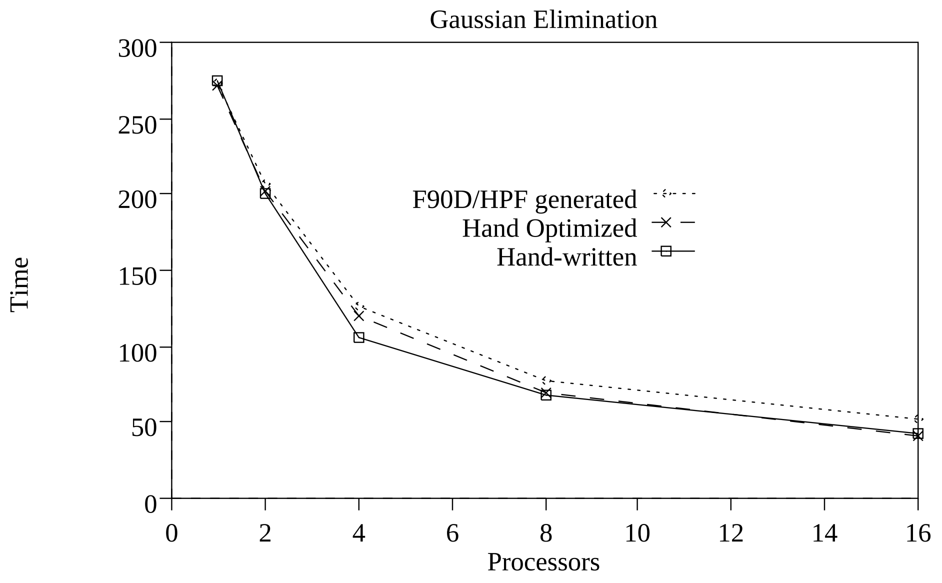


Figure 6.25: Performance of three version of GE. Matrix size is 1023x1024 (time in seconds).

Chapter 7

Experimental Results

This chapter presents benchmark results to illustrate performance obtained using the Fortran 90D/HPF compiler. The chapter shows the portability and scalability of the Fortran 90D/HPF compiler. It gives the performance results for different distributions. The chapter also compares the Fortran 90D/HPF generated code with the hand-written Fortran 77 + message passing codes to show that overall performance of Fortran 90D/HPF compiler and its run-time system is comparable to hand-written codes.

7.1 Test System

Our experiments were performed on a 16-node Intel iPSC/860 at Northeast Parallel Architecture Center at Syracuse University and on a 16-node Intel Paragon at the Portland Group, Inc. Most of the measurements reported were done by using our Fortran 90D/HPF compiler. Our compiler generates Fortran 77 plus message passing calls. The generated programs were compiled using the Parasoft Express Fortran compiler, which calls Portland Group if77 release 4.0 compiler with all optimizations turned on (-O4). The Fortran 90D/HPF runtime primitives use Parasoft's Express

Message Passing system, version 3.0. Timings were taken using *extime()* an Express function having an accuracy of one microsecond.

7.2 Portability

One of the principal requirements for users of distributed memory MIMD systems is some “guarantee” of portability for their code. The Express parallel programming environment [57] guarantees a level of portability on various platforms including, Intel iPSC/860, nCUBE/2, networks of workstations etc. We should emphasize that we have implemented a collective communication library which is currently built on the top of Express message passing primitives. Hence, in order to change to any other message passing system such as PVM [58] or MPI [59] (which also run on several platforms), we only need to replace the calls to the communication primitives in our communication library (not the compiler). However, it should be noted that a penalty must be paid to achieve portability because portable routines are normally built on top of the system routines. Therefore, the performance also depends on how efficient communication primitives are on top of which the communication library is built.

As a test application we use Gaussian Elimination, which is a part of the Fortran D/HPF benchmark test suite [66]. Figure 26 shows the execution times obtained to run the same compiler generated code on a 16-node Intel/860 and nCUBE/2 for various problem sizes.

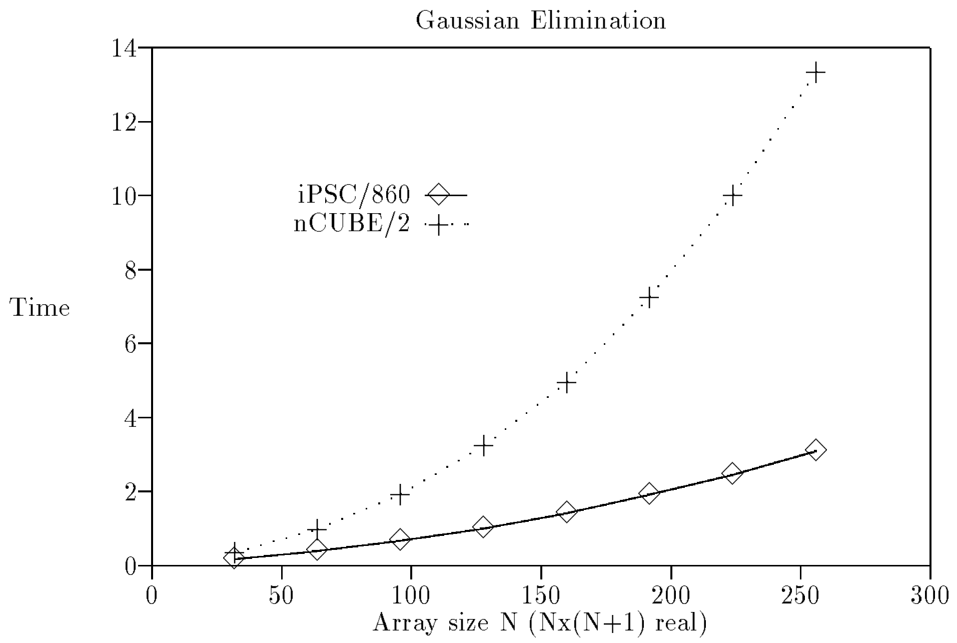


Figure 7.26: Gaussian Elimination on a 16-node Intel iPSC/860 and nCUBE/2 (time in seconds).

7.3 Scalibility

Distributed memory computers are characterized by their scalable architectures. These distributed-memory systems are expandable and can achieve a proportional performance increase without changing the basic architecture. In order to take full advantage of scalable hardware, the software must also be scalable to exploit the increased computing capability. This section presents benchmark results to illustrate that Fortran 90D/HPF generates scalable codes to exploit the scalable distributed memory machine.

All of these benchmarks were run on a 15-processor Intel Paragon. The processors run at 50 MHz, and each node has 32 MBytes of physical memory. The programs were compiled using the Fortran 90D/HPF compiler with all optimization turned on, including the i860 vectorizer which exploits single node parallelism.

The shallow water (*shallow*) benchmark is a small program (300 lines) abstracting a 2-dimensional flow system. The data is distributed in block fashion in one dimension, (*, block). The generated code consists of many computations of order N^2 with communication mostly consisting of overlap-shifts of order N. Figure 27 shows the performance of shallow. The super-linear speed-up on the large data set dramatically exhibits the ability of the Fortran 90D/HPF compiler to make large problems more tractable simply through efficient use of the larger available core memory on a multi-processor system.

The partial differential equation benchmark (*pde1*) is a small program (360 lines) from the Genesis Parallel Benchmark Suite that implements a 3D Poisson Solver using red-black relaxation through five iterations. Figure 29 shows the performance of pde1. The data is distributed block fashion in one dimension, (*,*,block). Good scalability is exhibited. The communication mostly consists of overlap-shifts due to the stencil computations of pde1.

The hydflo benchmark is a small hydrodynamics program (2000 lines). Figure 28 shows the performance of hydflo. The data is distributed block fashion in one dimension, (*,*,block). Good scalability is exhibited. The communication mostly consists of copy-section and collective-communication.

As shown by the data, benchmark programs written in Fortran 90D/HPF can achieve reasonable efficiency given a problem of reasonable size. The figures show reasonably good scalability when increasing numbers of processors are used.

7.4 Scalability of Intrinsics

It is desirable that a library be scalable. This section shows that Fortran 90D/HPF library provides a corresponding performance improvement as the number of processors increases.

Nproc	**ALL** (1K x 1K)	**ANY** (1K x 1K)	**MAXVAL** (1K x 1K)	**PRODUCT** (256K)	**TRANSPOSE** (512 x 512)
1	580.6	606.2	658.8	90.1	299.0
2	291.0	303.7	330.4	50.0	575.0
4	146.2	152.6	166.1	25.1	395.0
8	73.84	77.1	84.1	13.1	213.0
16	37.9	39.4	43.4	7.2	121.0
32	19.9	20.7	23.2	4.2	69.0

Table 7.5: Performance of Intrinsic Functions (time in milliseconds).

Table 5 presents a sample of performance numbers for a subset of the intrinsic functions on iPSC/860. A detailed performance study is presented in [51]. The times in the table include both the computation and communication times for each

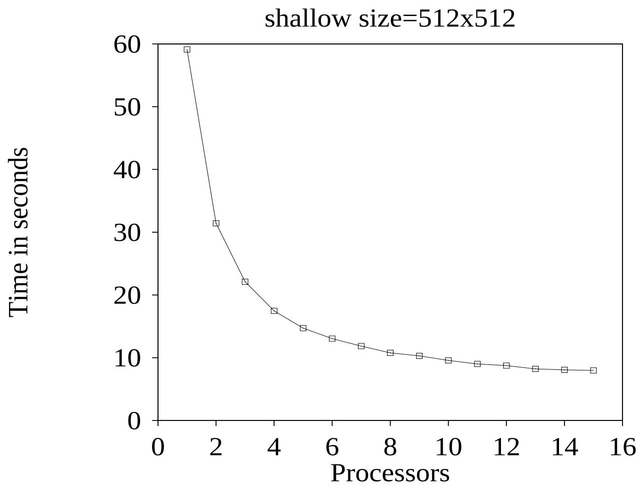


Figure 7.27: Shallow Performance

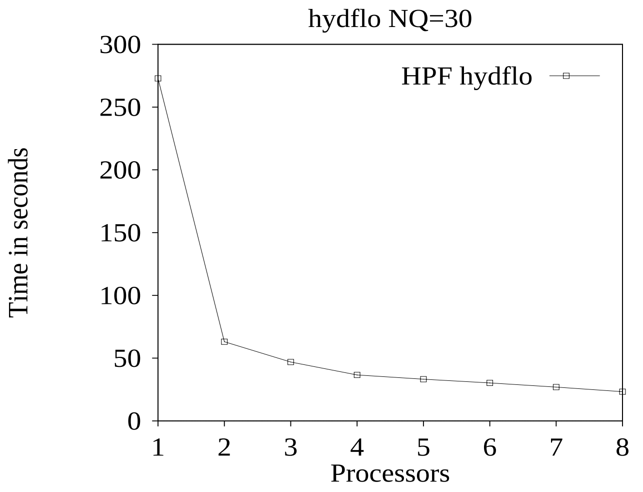
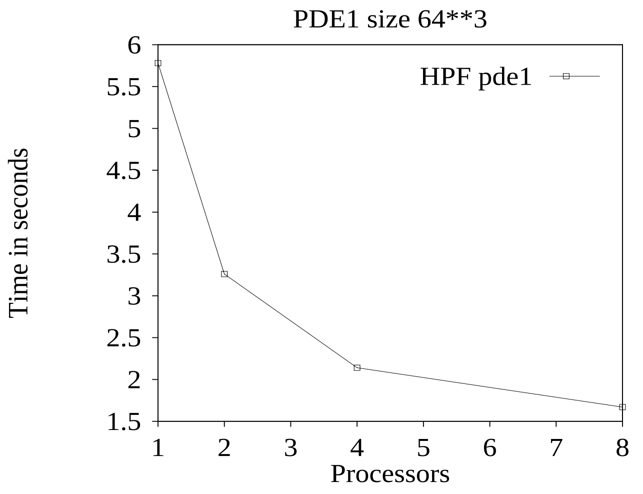


Figure 7.28: Hydflo Performance



function. For most of the functions we were able to obtain almost linear speedups. In the case of the TRANSPOSE function, going from one processor to two or four actually results in an increase in the time due to the communication requirements. However, for larger size multiprocessors the times decrease as expected.

7.5 An Experiment with Distributions

A significant advantage of coding in Fortran 90D/HPF is the ability to specify different distribution directives and measure performance differences without extensive recoding. Block distribution strategy for allocating elements to a processor is ideal for computations that reference adjacent elements along an axis, as is the case in many relaxation methods [67]. The number of references to non-local variables for a given number of local variables is minimized when the volume to surface ratio is maximized. However, block distribution may result in poor load balance. Some experimentation along these lines was performed on the Gauss benchmark (*gauss*), which is a program designed to measure the performance of a Gaussian elimination algorithm.

Figure 7.5 gives the main factorization loop of *gauss* which converts matrix a to upper triangular form. This Gaussian elimination algorithm is sub-optimal due to a mask in the inner loop which prevents vectorization.

Figure 31 (a) shows the updated values of matrix a in the shaded region after the factorization loop. Since the compiler uses the owner computes rule to assign computations, only owners of data in the shaded region will participate in the computation. The remaining processors are masked out of the computation. Figures 31 (b) and (c) show the computation distribution on four processors in block and cyclic fashions respectively. X axes shows that how the data is distributed on four processor grids.

```
  do k = 1, N
    ...
    forall (i = 1:N, j = k:N, indx(i) .EQ. -1)
&     a(i,j) = a(i,j) - fac(i)*row(j)
  endo
```

Figure 7.30: Main factorization loop in gauss.

In this particular benchmark, cyclic distribution results in better load balancing than block distribution.

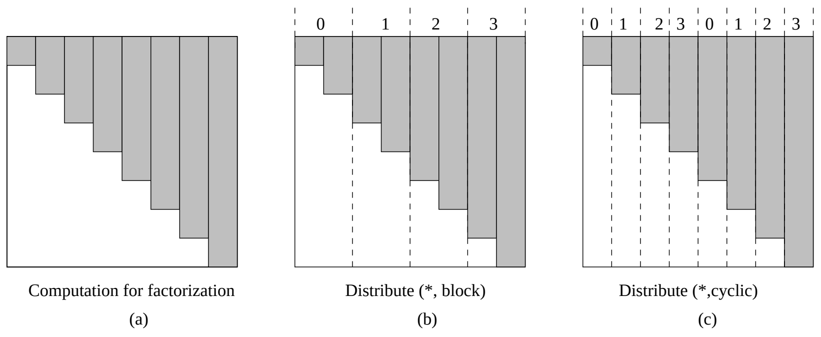


Figure 7.31: Load balancing for gauss

Figure 32 presents the performance using cyclic as well as block distributions on Intel Paragon. As expected, the cyclic distribution exhibits better performance because of load balancing. The communication requirements for these distributions are identical. Both use multicast.

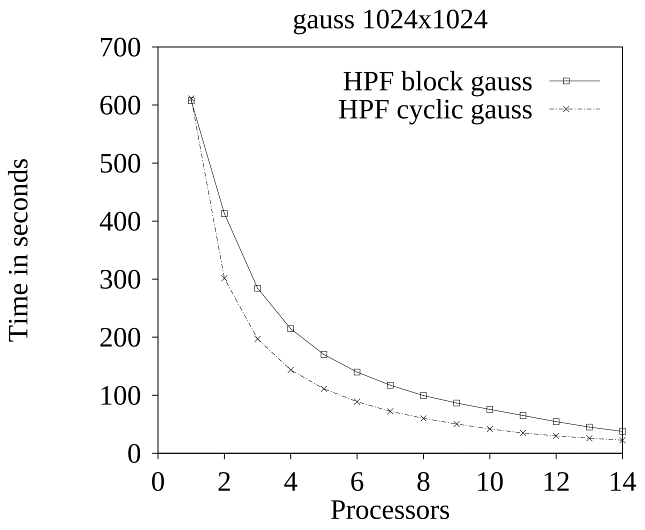


Figure 7.32: Gauss Performance

7.6 Hand-written Comparison

To illustrate the performance of our compiler, we present benchmark results from four programs and the first 10 Livermore loop kernels. *Gauss* solves a system of linear equations with partial pivoting. *Nbody* program simulates the universe using the algorithm in [49]. *Option* program predicts the stock option pricing using stochastic volatility European model [68]. *Pi* program calculates the value of π, using numerical integration. The Livermore kernels are 24 loops abstracted from actual production codes that have been widely used to evaluate the performance of various computer systems. Data for all programs were block distributed and were written outside of the compiler group at NPAC by experienced message passing programmers.

Tables 6 and 7 show the performance of compiler generated codes ($F90D/HPF$)

Table 7.6: Fortran 90D/HPF versus the hand-written code for several applications. (Intel iPSC/860, time in seconds).

Program	Problem Size	Number of PEs				
		1	2	4	8	16
Gauss Hand	1023x1024	623.16	446.60	235.37	134.89	79.48
Gauss F90D	1023x1024	618.79	451.93	261.87	147.25	87.44
Nbody Hand	1024x1024	6.82	1.74	1.29.	0.76	0.42
Nbody F90D	1024x1024	13.82	5.95	2.40	1.31	0.86
Option Hand	8192	4.20	3.14	1.60	0.83	0.43
Option F90D	8192	4.30	3.19	1.64	0.84	0.44
Pi Hand	65536	0.398	0.200	0.101	0.053	0.030
Pi F90D	65536	0.411	0.207	0.104	0.054	0.032

and hand-written f77+ MP code. The tables contain data from running these programs with a varying number of processors on Intel iPSC/860.

We observe that the performance of the compiler generated codes are usually within a factor of 2 of the hand-written codes. This is due to the fact that an experienced programmer can incorporate more optimizations than our compiler currently does. For example, a programmer can combine or eliminate some of the communication or some of the intra-processor temporary copying. The compiler uses a more generic packing routine, whereas a programmer working by handcan combine communication for the same source and destination for different arrays. Another observation is that our run-time system shift routine is slower than the programmer's shift routines.

Table 7.7: Fortran 90D/HPF versus the hand-written code for the first 10 Livermore loop kernels. Data size is 16K real. (a 16 node Intel iPSC/860, time is in milliseconds).

Loop number	Type of Application	F90D/HPF	Hand	Ratio
1.	Hydrodynamics	2.545	2.550	0.9980
2.	Incomplete Cholesky	11.783	10.440	1.1286
3.	Inner product	3.253	3.249	1.0012
4.	Banded linear equations	5.139	3.212	1.600
5.	Tridiagonal elimination	30928.6	30897.7	1.001
6.	Linear recurrence relations	1849.1	1886.5	0.9801
7.	Equation of state	11.346	3.704	3.0632
8.	A.D.I	38.656	20.038	1.9291
9.	Numerical Integration	2.255	2.441	0.9238
10.	Numerical Differentiation	9.814	4.589	2.1386

Chapter 8

Conclusions

HPF is rapidly gaining acceptance as an easy-to-use and portable programming language for high performance scientific applications. The wide variety of current parallel system architecture, however, presents a significant challenge to HPF compiler implementors. Here, we have explored some of the issues and outlined Fortran 90D/HPF compilation and execution techniques. Fortran 90D/HPF compiler demonstrates that with language and run-time support, advanced compilation technology can produce efficient programs for distributed memory machine. In this chapter, we summarize the research embodied in this thesis. We conclude by considering areas for future work.

8.1 Compiling Fortran 90D/HPF

The Fortran 90D/HPF compiler is organized around several major units: parsing the language, partitioning data and computation, detecting communication and generating code.

The compiler transforms data distribution specifications found in the Fortran

90D/HPF source (*decomposition*, *distribute*, *align*) into predefined mathematical distribution functions that determine the partitioning of data on the distributed memory system. We developed an algorithm to compile align directives to minimize communications. The compiler maps the data on abstract processor grids, then maps the processor grid efficiently on the underlying hardware topology to reduce the importance of underlying topology.

The compiler must recognize the presence of communication patterns in the computations in order to generate appropriate communication calls. Specifically, this involves a number of tests on the relationships among subscripts of various arrays in a statement. We designed an algorithm to detect communications and to generate appropriate collective communication calls to execute array assignment and *forall* statements on distributed memory machines.

The Fortran 90D/HPF compiler relies on a powerful runtime support system. The compiler replaces some of the explicit parallelism with calls to the parallel runtime system. The runtime support system consists of functions which can be called from the node programs of a distributed memory machine. We developed an easy-to-use interface to the runtime system.

Our compiler performs several types of *communication* and *computation* optimization to maximize the performance of the generated code. Communication optimization can be classified as *Communication Hierarchy*, *Vectorized Communication*, *Message Aggregation*, *Evaluating Expression*, *Communication Parallelization*, *Communications Union*, *Eliminate Unnecessary Communications* and *Reuse of scheduling* *information.* In addition, some computation optimizations are developed for sequentialization of *forall* statements such as *Dependency*, *Loop Interchange*, *Mask Insertion.* Some of these optimizations are validated with an example.

Empirical measurements show that the performance of the output of the Fortran 90D/HPF compiler for real world application programs is comparable to that of corresponding hand-written codes on the Intel iPSC/860 and Paragon.

We have indicated our confidence in the performance of the code generated by the compiler by publishing the absolute execution times of our benchmarks. We believe that our Fortran 90D/HPF compiler greatly improves programmer productivity. Fortran 90D/HPF programs are shorter, easier to write, and easier to debug than programs written in Fortran 77 with message passing. We have found that Fortran 90D/HPF makes it much easier to tune nontrivial programs.

8.2 Future Work

8.2.1 Fortran 90D/HPF on Low Latency Systems

Much of the work on Fortran 90D/HPF has focused on overcoming the limitations of high latency message based systems. Many existing Massively Parallel Processors (MPP systems) have relatively high latency, especially when compared to conventional shared memory multi-processors. Even machines that support a shared memory programming model have latencies that can be up to 500 clock cycles. Clustered workstations, especially those using off the shelf interconnect technology, may have latencies one or two orders of magnitude higher than this.

For high-latency systems it is important to minimize accesses to remote data, whether it is stored on another processor or in a remote section of shared memory. Techniques such as message vectorization, collective communication, and inspector/executor loops (as found in PARTI) can be used on distributed memory machines. Techniques such as prefetching and data vectorization can hide latencies on shared memory systems.

An important challenge for Fortran 90D/HPF is to exploit lower latency systems. For example, the Meiko CS/2 supports a remote memory access paradigm. Any processor can read or write any other processor's memory without intervention by the second processor. The latency for a remote read or write is less than 500 clock cycles. True shared memory machines like those available from Sun, SGI, and Digital reduce these latencies even further.

Traditionally, shared memory machines are programmed using multi-threaded models. One thread is created for each processor; all but one of these threads waits on a shared memory location, or semaphore, until a parallel region is entered. When a parallel region is entered each thread is handed a unit of work, often a loop iteration or block of loop iterations, it executes until finishing its work unit, then obtains another work unit or returns to the idle state.

The multi-threaded model can be highly efficient in terms of work distribution and processor utilization; however, the model ignores a very important point: locality. Even a simple shared memory multiprocessor does not provide uniform access to memory. Cache hierarchies and bus contention conspire to slow memory access times. On distributed shared memory machines the problem is the same, but perhaps an order of magnitude worse. It is here that Fortran 90D/HPF is useful.

Simply put, Fortran 90D/HPF allows the programmer to describe data locality;

on modern multi-processors data locality is the key to performance. One of the most useful features of HPF is the relaxation of Fortran's storage and sequence association rules. For example, consider a simple loop:

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

On a two processor shared memory system, one strategy for this loop would be to assign even iterations to one processor and odd iterations to the second processor. Let's assume that elements of A and B are 4-bytes long and that cache lines are 32-bytes wide. Then, each processor could access a cache line owned by the other processor as many as 4 times, resulting in many unnecessary cache line transfers and overloading the memory system. This phenomenon is known as false sharing [69]. The loop can also be parallelized by allocating blocks of iterations to the processor. This eliminates some of the performance difficulties, but still results in potential false sharing at block boundaries.

On the other hand, Fortran 90D/HPF allows the compiler to break the arrays A and B into blocks. The compiler can align these blocks on cache line boundaries. This eliminates the false sharing problem without changing the semantics of the program and still allows use of the shared memory system's fast memory access.

Our strategy for Fortran 90D/HPF compilation on low-latency systems is as follows:

- Partition data to eliminate false sharing (on shared memory systems).
- Continue to use fast collective communications routines when the compiler can

recognize their use. These routines will be customized to take full advantage of fast interconnect or shared memory systems.

- Take advantage of low latency to directly read and write distributed elements when unstructured communications patterns exist.

8.2.2 InterProcedural Analysis

Interprocedural Analysis (IPA) applies various techniques across procedure boundaries and can be used on Fortran 90D/HPF programs to enhance optimizations, expose parallelism, minimize communication and check program correctness [70].

The most immediate area where IPA is needed is in detecting the compliance of variables and COMMON blocks with the relaxed storage and sequence association rules presented in the HPF Language Specification. Compliance needs to be checked at link time or whenever all the relevant program units are visible. Obviously, additional information describing the common blocks and variables needs to be made available. This can be done through a database mechanism or by embedding the information within the object file. The method used may depend on whether the mode of operation is sourcetosource or source-to-machine language.

However, the need for IPA goes far beyond correctness checking. HPF can be a successful parallel programming paradigm only when it improves the application's performance. The transformation from a sequential program to a Single Program Multiple Data (SPMD) program occurs in three major areas:

1) Loops and array references are transformed in their index range. Other types of loop transformations may occur such as interchange, splitting and merging.

2) Insertion of message passing calls (communication and synchronization) are

added to handle static and dynamic mapping.

3) Miscellaneous code (including conditional statements) is added in order to regulate processor control over certain portions of the program. Conditional statements are added to check for the need for remapping of data.

It is possible to use IPA to reduce the amount of overhead in this sequential to SPMD transformation.

Constant Propagation across procedure boundaries is probably the easiest and yet one of the most useful IPA methods. It can aid in reducing all three of the above areas. Additionally, it can aid in the latter stages of single node optimizations such as vectorization and software pipelining.

Propagation of mapping information across procedure calls can reduce overhead involved in areas 1 and 2 in multiple ways. One way is to determine program correctness when distribution directives are used for variables that assert their mapping. Probably the largest gain from IPA of mapping information is in the elimination of unnecessary distribution checking or redistribution of data across calls. This can be a significant advantage since the alternative is to redistribute upon entry and upon exit to a procedure.

Function cloning, when used judiciously, can save significant amounts of overhead. For example, mapping information may reveal that an argument to a call has only two mappings possible. If so, it may be beneficial to clone that function with each having a different assumption about the mapping of the incoming mapped argument.

Function inlining is the traditional form of interprocedural analysis and is sometimes referred to as an interprocedural transformation. This technique has enjoyed some success although results have been inconsistent. Function inlining has the potential to expose parallelism, reduce calling overhead, increase optimization opportunities and eliminate redistribution and communications. However, unnecessary function inlining can often have negative side effects as well including code explosion.

Bibliography

[1] G. C. Fox, S. Hiranadani, K. Kenndy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical report, Rice and Syracuse University, 1992.

[2] High Performance Fortran Forum. High performance fortran language specification. *Technical Report, Version 1.0, Rice University.*, May 1993.

[3] G.C. Fox. Parallel computing comes of age: Supercomputer level parallel computations at Caltech. *Concurrency: Practice and Experience*, pages 63–103, September 1989.

[4] StevenW. Otto, Adam Kolawa, and Anthony Hey. Performance of the MarkII Caltech/JPL hypercube. Technical Report C3P-188, California Institute of Technology, August 1985.

[5] J.Boyle, R.Butler, T.Disz, B.Glickfeld, E.Lusk, R.Overbeek, J.Patterson, and R.Stevens. Portable Programs for Parallel Processors. *Holt, Rinehart and Winston, Inc.*, 1987.

[6] A.H. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[7] A.H. Karp and R.G. Babb II. A comparison of 12 parallel Fortran dialects. *IEEE Software*, pages 52–67, September 1988.

[8] C.D.Polychronopoulos et al. Parafrase-2 : An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Proc. Int'l Conf. on Parallel Processing*, pages 39–48, August 1989.

[9] J.R. Allen and K.Kennedy. PFC: A program to convert Fortran to parallel form. *Supercomputers: Design and Applications*, pages 186–205, 1984.

[10] L.W. Tucker and G.G. Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, pages 26–38, August 1988.

[11] G.C. Fox. What have we learnt from using real parallel machines to solve real problems? In *The Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, pages 897–955, January 1988.

[12] G.C. Fox. Achievements and prospects for parallel computing. Technical Report SCCS-29, Syracuse University, 1990.

[13] G.C. Fox. FortranD as a portable software system for parallel computers. Technical Report SCCS-91, Syracuse University, 1991.

[14] G.C. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-78b, Syracuse University, 1991.

[15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimization for Fortran D on MIMD distributed-memory machines. *Proc. Supercomputing'91*, Nov 1991.

[16] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiler support for machine-indepentet Parallel Programming in Fortran D. *Compiler and Runtime Software for Scalable Multiprocessors*, 1991.

[17] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[18] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Handbook*. MIT Press, 1994.

[19] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi Automatic SIMD/MIMD Parallelization. *Parallel Computing*, January 1988.

[20] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting Shared Data Structures on Distributed Memory Architectures. *PPoPP*, March 1990.

[21] C. Koelbel and P. Mehrotra. Supporting Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[22] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An Automatic and symbolic parallelization system for distributed memory parallel computers. the 5th Distributed Memory Computing Conference, April 1990.

[23] M. Quinn, P. Hatcher, and K. Jourdenais. Compiling C* Programs for a Hypercube Multicomputer. *Parallel Computing Laboratory, University of New Hampshire*, PCL-87-12, December 1987.

[24] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and R. Anderson. A Production-Quality C* Compiler for Hypercube Multicomputers. *Third ACM SIGPLAN symposium on PPOPP*, 26:73–82, July 1991.

[25] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, December 1991.

[26] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, December 1991.

[27] J. Saltz, J. Wu, H. Berryman, and S. Hiranandani. Distributed Memory Compiler Design for Sparse Problems. *Interim Report ICASE, NASA Langley Research Center*, 1991.

[28] J. Saltz R. Das and H. Berryman. A Manual For PARTI Runtime Primitives. *NASA,ICASE Interim Report 17*, May 1991.

[29] K. Knobe, J. D. Lukas, and G. L. Steele. Compiling Fortran 8x Array Features for the Connection Machine Computer Systemication on SIMD machines. the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems, 1988.

[30] The Thinking Machine Corporation. *CM Fortran User's Guide version 0.7-f*, July 1990.

[31] G. Sabot. A Compiler for a Massively Parallel Distributed Memory MIMD Computer. *The Fourth Symposium on the Frointiers of Massively Parallel Computation*, 1992.

[32] Thinking Machines Corporation. In *The Connection Machine CM-5 Technical Summary*, October 1991.

[33] D. M. Pase, T. MacDonald, and Andrew Meltzer. MPP Fortran Programming Model. Technical report, Cray Research, Inc., 1992.

[34] M. Y. Wu and D. D. Gajski. A programming aid for message-passing systems. *Parallel Processing for Scientific Computing*, pages 328–332, 1989.

[35] J.H Merlin. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'. Technical Report SNARC 92-02, Southampton Novel Architecture Research Centre, 1992.

[36] T. Brandes. ADAPTOR Language Reference Manual. Technical Report ADAPTOR-3, German National Research Center for Computer Science, 1992.

[37] J. Li and M. Chen. Compiling Communication -Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 361–376, July 1991.

[38] M. Chen and J.J Wu. Optimizing FORTRAN-90 Programs for Data Motion on Massivelly Parallel Systems. Technical Report YALEU/DCS/TR-882, Yale University, Dep. of Comp. Sci., 1992.

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

[40] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, pages 213–221, Oct 1991.

[41] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE: Transaction on Parallel and Distributed Systems*, pages 179–193, March 1992.

[42] V. Balasundaram, G. C. Fox, K. Kennedy, and U. Kremer. An Interactive Environment for Data Partitioning and Distribution. In *Fifth Distributed Memory Compu. Conf.*, Apr 1990.

[43] R. Allen. Dependency analysis for Subscripted Variables and its Application to Program Transformation. Technical Report PhD thesis, Rice University, 1983.

[44] J. Ng, V. Sarkar, and J.F. Shaw. Optimized execution of Fortran 90 array constructs on supercomputer architectures. In *Supercomputing'91*, 1991.

[45] B. Chapman, H. Herbeck, and H. Zima. Automatic Support for Data Distribution. *IEEE: Transaction on Computers*, pages 51–57, 1991.

[46] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S.H Tseng. Automatic Array Alignment in Data-Parallel Programs. *Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1993.

[47] K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, pages 102–118, Feb 1990.

[48] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE: Transaction on Parallel and Distributed Systems*, pages 472–482, October 1991.

[49] G. C. Fox, M.A. Johnson, G.A. Lyzenga, S. W. Otto, J.K. Salmon, and D. W. Walker. *Solving Problems on Concurent Processors*, volume 1-2. Prentice Hall, May 1988.

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

[51] I. Ahmad, R. Bordawekar, Z. Bozkus, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Fortran 90D Intrinsic Functions on Distributed Memory Machines: Implementation and Scalability. Technical Report SCCS-256, Northeast Parallel Architectures Center, March 1992.

[52] Z. Bozkus, S. Ranka, and G. C. Fox. Benchmarking the CM-5 multicomputer . *The Fourth Symposium on the Frointiers of Massively Parallel Computation*, 1992.

[53] D. Callahan and K. Kennedy. Compiling programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, pages 171–207, 1988.

[54] M. Wu and G. Fox et al. Compiling Fortran 90 programs for distributed memory MIMD paralelel computers. Technical Report SCCS-88, Northeast Parallel Architectures Center, May 1991.

[55] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, September 1990.

[56] A.V. Aho, R. Sethi, and J.D Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, March 1988.

[57] ParaSoft Corp. *Express Fortran refernce guide Version 3.0*, 1990.

[58] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam. A Users Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[59] J. Dongara, R. Hempel, A. Hey, , and D. Walker. Message Passing Interface. Technical Report TM-12231, ORNL, 1992.

[60] R. Hempel, H.C. Hoppe, U. Keller, and W. Krotz. Parmacs v6.0 specification, interface descriptionon. *Pallas GMBH, Bruhl.*, Nov. 1993.

[61] D. Palermo, E. Su, J. Chandy, and P. Banarjee. Communication Optimizations Used in the Paradigm Compiler For Distributed-Memory Multicomputers. International Conference on Parallel Processing, 1994.

[62] S. Chatterjee, J.R. Gilbert, and R. Schreiber. Optimal Evaluation of Array Expression on Massively Parallel Machines. Boulder, CO, October 1992. The Second Workshop on Languages, Compilers, and Runtime Environments For Distributed Memory Multicomputers.

[63] V. Bouchitte, P. Boulet, A. Darte, and Y. Robert. Evaluating Array Expressions on Massivelly Parallel Machines with Communication/Computation Overlap. Technical Report 94-10, Ecole Normale Superieure de Lyon, 1994.

[64] M. Young. Personal Communication. Technical report, The Portland Group, Inc., 1994.

[65] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1990.

[66] A. G. Mohamed, G. C. Fox, G. V. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Application Benchmark Set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, Northeast Parallel Architectures Center, May 1992.

[67] S. L. Johnsson. Performance Modeling of Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, pages 300–312, August 1991.

[68] K. Mills, M. Vinson, and G. Cheng. A Large Scale Comparison of Option Pricing Models with Historical Market Data. *The Fourth Symposium on the Frointiers of Massively Parallel Computation*, 1992.

[69] G.C.Fox. Domain decomposition in distributed and shared memory environments I. Technical Report C3P-392, California Institute of Technology, 1987.

[70] R. Ponnusamy, J. Saltz, and A. Choudhary. Compilation Techniques for Data Partitioning and Communication Schedule Reuse. Supercomputing '93, IEEE Computer Society Press, 1993.

Vita

Biographic Data

Name:	Zeki Bozkus
Date and Place of Birth:	June 15, 1964 Kahramanmaraş, Turkey
College:	Middle East Technical University, Ankara B.S. in Computer Engineering (1988)
Graduate Work:	Syracuse University, Syracuse, New York M.S. in Computer and Information Science (1990)
	Syracuse University, Syracuse, New York Ph.D. in Computer and Information Science (1995)
Work Experience:	Computer Programmer Military Electronics Industries, Ankara, Turkey 1987 to 1988 Research Assistant Syracuse University, Syracuse, New York 1991 to 1993 Software Engineer The Portland Group, Inc., Wilsonwille, Oregon 1993 to present