RICE UNIVERSITY

Communication Generation for Data-Parallel Languages

by

Ajay Sethi

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

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor
Department of Computer Science
Rice University

Peter Druschel, Assistant Professor
Department of Computer Science
Rice University

Richard Tapia, Noah Harding Professor
Department of Computational and
Applied Mathematics
Rice University

Linda Torczon, Faculty Fellow
Department of Computer Science
Rice University

Houston, Texas

December, 1996

Communication Generation for Data-Parallel Languages

Ajay Sethi

Abstract

Data-parallel languages allow programmers to use the familiar machine-independent programming style to develop programs for multiprocessor systems. These languages relieve users of the tedious task of inserting interprocessor communication and delegate this crucial and error-prone task to the compilers for the languages. Since remote access in hierarchical multiprocessor systems is orders of magnitude slower than access to a processor's local memory, interprocessor communication introduces significant overheads to the total execution time. The success of data-parallel languages depends heavily on the compiler's ability to reduce the communication overhead.

This dissertation describes novel techniques for communication generation. It covers issues related to communication analysis, placement, and optimization. The techniques have been implemented in the Rice Fortran D95 research compiler – a High Performance Fortran (HPF) compiler – being developed at the Rice University.

A major contribution of the dissertation is the development of a data-flow analysis framework for supporting communication placement and optimization in the presence of machine-dependent resource constraints. Examples of resource constraints include in-core memory size, cache size, and the number of physical registers. Communication placement and optimizations that do not take resource constraints into account can lead to incorrect communication placement and/or performance loss. This work also

describes how the data-dependence information can be combined with data-flow analysis to improve the scope of some of the well-known communication optimizations.

Finally, the dissertation presents communication generation techniques for the *cyclic(k)* distributions supported by HPF. It presents efficient algorithms for computing the local addresses as well as for generating the communication sets. The innovative techniques described in the dissertation exploit the repetitive pattern exhibited by the *cyclic(k)* accesses.

Acknowledgments

Ken Kennedy. For his support and guidance. For letting me learn how to do research. For giving me freedom to explore and experiment. For providing an excellent and stimulating research environment.

Thesis committee members: Peter Druschel, Linda Torczon, and Richard Tapia. For finding time to listen to me despite hectic schedules.

Ivy Jorgensen and Ellen Butler. For their patience and willingness to help. For taking care of the small details that keep the department running smoothly. For trying, and succeeding, in scheduling my defense despite seemingly insurmountable odds.

Fortran D group members. Seema Hiranandani for introducing me to the compiler and for helping me to get started. Vikram Adve, John Mellor-Crummey, and Nenad Nedeljković for countless interesting discussions and for being extremely helpful and reliable collaborators during the implementation of the Fortran D95 compiler. Uli Kremer, Nat McIntosh, Mike Paleczny, Chuck Koelbel, Dejan Mircevski, Debbie Campbell, Mark Anderson, Reinhard von Hanxleden, Lani Granston, and Gil Hansen for patiently answering all my questions.

Friends and family. And the friends who have become as close as my own family members. For everything.

God. For his grace – without which nothing would have been possible. For giving me the opportunity to learn: academically as well as otherwise.

Contents

Illustrations

Chapter 1

Introduction

Communication delay is the main impediment to high performance on modern parallel computers. Since most parallel machines package memory with processors, accesses to remote memory are typically an order of magnitude slower than accesses to local memory. Thus, remote accesses add communication overhead to the total execution time and, as the number of processors increase, communication overhead can dominate the execution time [85]. Reducing communication overhead is a key goal of optimizing parallel programs.

Multiprocessor/parallel systems support high-level parallel languages that allow programmers to use the familiar, machine-independent programming style to develop programs for multiprocessor systems. Parallel languages provide an illusion of global memory to the users and, thus, relieve them of the burden of inserting machine-dependent, tedious, and often error-prone interprocessor communication. This task is handed over to the compilers for these languages. The compilers for multiprocessor systems take a parallel language as the source language and an assembly language or another high-level language such as Fortran together with a machine-dependent interprocessor communication library as the target language.

There are two basic tasks that any compiler for multiprocessor systems must perform. These tasks can be broadly classified under two categories: *parallelism* and *communication*. First, the compiler must extract parallelism from programs and schedule the programs on multiprocessor computer so that they run as fast as possible. Second, the compiler must generate the necessary interprocessor communication. Communication in a program imposes overhead and opposes parallelism.

The current generation of parallel languages often allow programmers to assist the compilers by exposing the inherent parallelism in the programs. However, since communication is machine dependent, the task of communication generation is handled completely by compilers. As mentioned above, the communication overhead is often the single biggest cause for poor performance on multiprocessor systems. Thus, the success of these languages depends heavily on the compiler's ability to reduce the communication overhead.

1.1 Data-parallel computing

There are two common kinds of parallelism. The parallelism can be in terms of data: same operations are performed, except for boundary cases, on many elements of a data structure (such as an array). This form of parallelism is called *data parallelism.* Or, the parallelism can be in terms of the functions that are called: program invokes conceptually different functions or tasks that can be executed in parallel. This kind of parallelism is known as *task* or *functional parallelism*.

This dissertation concentrates on *data parallelism*, which is exhibited by a wide variety of scientific and engineering applications. The current generation of parallel languages provide constructs that allow users to express this parallelism at an abstract level. The advantage of this approach is that the abstractions can be implemented efficiently on many multiprocessor systems. The languages that allow expressing data parallelism are known as *data-parallel languages.* Data-parallel languages allow users to specify how shared data are distributed onto processors. This information is a very high-level specification that often depends on the nature – such as the physical or some other underlying criteria – of the problem being solved. In return for this information, data-parallel languages allow the users to write essentially sequential programs.

The compilers utilize user-specified data distribution information to determine how work is divided among processors and, consequently, how processors need to cooperate during the execution of a program. Work is divided among processors by translating the user's program into a program that runs on each processor such that every processor has the same program structure but works on a different portion of the shared data. The translated program is called processor node code and this style is known as the single-program multiple-data (SPMD) compilation paradigm. Typically, processors work independently on their own data for some duration of time and communicate or synchronize with other processors when there exists some potential data dependence among various node code instances.

1.2 Communication generation

The communication overhead is reduced during communication generation in the compilers for parallel languages. The communication generation process can be divided into three subphases:

1. **Communication analysis:** This phase determines the program references that require communication, represents communication using some suitable representation, computes the communication pattern, and determines set of processors that are involved in interprocessor communication.

2. **Communication placement:** This phase determines the placement of communication primitives to reduce communication overhead. This is achieved by reducing communication frequency, which requires hoisting communication to the least frequently executed program location, and by hiding communication latency. Communication latency is hidden by separating communication "start"

and "end" primitives and overlapping communication with some intervening computation. This reduces (perhaps, even eliminates) the time processors spend waiting for the communicated data to arrive.

3. **Communication optimization:** This phase further reduces the communication overhead by performing various communication optimizations. An example of communication optimization is the elimination of redundant communication.

An important issue in communication generation is the interaction between the three subphases as well as the interaction with other compiler phases such as the code generation phase.

1.3 Overview of the dissertation

While considerable research has been done to develop compiler techniques for data-parallel languages, there are still many inadequacies in the current techniques. This dissertation addresses the compilation issues related to communication generation and presents a number of novel techniques as well as improved versions of some of the existing techniques. The thesis of this dissertation is:

Compilers for data-parallel languages can perform communication generation efficiently with the help of new techniques for communication analysis and with a systematic data-flow communication placement framework that incorporates communication optimizations and allows resource constraints to influence the placement.

To support the thesis, this dissertation makes the following original contributions:

1. Most current compilers performs communication placement by using dependence information and by considering a loop nest in isolation. Dependence-based placement does not support the non-atomic placement of communication

"start" and "end" primitives for latency hiding. Moreover, the absence of data-flow analysis precludes the possibility of redundant communication elimination in presence of conditionals and arbitrary control-flow. Though, over the last two years, a few data-flow based communication placement frameworks that address these issues have been developed, with the exception of the Fortran D compiler [44], no compiler has implemented data-flow based communication placement analysis. This dissertation presents a global data-flow analysis technique that determines *balanced* non-atomic placement of communication primitives and improves placement analysis in various other respects.

2. To simplify the placement analysis, none of the previous communication placement analysis techniques proposed for data-parallel compilers or the code placement techniques used by the traditional (sequential/uniprocessor) compilers provide support to allow machine-dependent resource constraints (such as cache size and buffer size) to influence the placement. The data-flow placement framework presented in this dissertation incorporates resource constraints during the analysis.

3. Current communication optimization techniques have two limitations. First, since the communication placement analysis is performed for individual loop nests, communication optimizations – such as combining overlapping messages – are also applied to communication arising in the same loop nest. Second, communication optimizations are often implemented by recognizing special dependence patterns. In addition, communication optimizations do not take resource constraints into account.

   This dissertation presents improved versions of communication optimizations by incorporating data-flow information. It shows that the scope and precision of communication optimizations can be improved with data-flow analysis. Moreover, it illustrates that the communication optimizations depend not only on the dependence and data-flow information, but also on the resource constraints. It describes how communication optimizations are performed in the presence of not only arbitrary control flow but also resource constraints.

4. Current state-of-the-art HPF compilers [38, 3, 14] support techniques for simple regular distributions with a limited set of subscript patterns and provide either no or inefficient support for the general regular distribution (specifically, the *cyclic(k)* distribution; it is defined in Section 2.1.1). This dissertation describes the first linear-time algorithm developed to compute local address sequence for *cyclic(k)* distribution. It also describes new, linear-time analysis for computing communication sets and innovative techniques to handle the related code generation issues.

It is important to note that only the communication analysis techniques for the *cyclic(k)* distributions are dependent on the source language. Communication placement and optimization techniques are independent of source language. Moreover, all the techniques described in this dissertation are independent of the target language as well as the target architecture. In other words, the communication generation techniques presented in this dissertation are machine-independent analysis techniques.

The techniques presented in this dissertation were implemented and validated in the Fortran D95 compiler, the second generation HPF compiler being developed at the Rice University. Chapter 2 presents an overview of the Fortran D95 compiler along with a brief introduction of HPF.

The rest of the dissertation is organized as follows. Chapter 3 describes communication analysis techniques along with the new communication set representations used by the Fortran D95 compiler. Chapter 4 describes the data-flow analysis framework for determining placement of communication primitives while Chapter 5 describes how machine-dependent resource constraints are incorporated into the placement analysis. Chapter 6 discusses how various communication optimizations are incorporated into the resource-based communication placement framework. Chapter 7 then describes communication analysis and optimization techniques as well as the code generation issues related to *cyclic(k)* distribution. Chapter 8 concludes with a summary of contributions made by the dissertation and describes some issues raised by the research.

Chapter 2

Compiler Overview

As mentioned in Section 1.3, the techniques presented in this dissertation were implemented in the Fortran D95 compiler, the second generation research compiler for HPF. Section 2.2 presents an overview of the compiler. But first, Section 2.1 describes the features of HPF – source language of the Fortran D95 compiler – that are relevant to this dissertation. Section 2.3 compares the Fortran D95 compiler with some other data-parallel compilers.

2.1 Introduction to HPF

HPF is a data-parallel language that has been developed over the last few years after extensive collaboration between the researchers and experts from industry and academia. The following quote summarizes HPF concisely:

"HPF combines the full Fortran 90 language with special user annotations dealing with data distribution. It is expected that HPF will be a 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." — The High Performance Fortran Handbook [69].

HPF is based on Fortran 90 [1], which allows expressing data-parallel array expressions (that is, operations that act on whole arrays). In addition, Fortran 90 eliminates some of the deficiencies of Fortran 77. In particular, it allows the use of dynamically allocatable arrays as well as pointers. It also provides improved facilities for modular programming; new features allow better data abstraction as well as better packaging of procedures. Further, a number of intrinsic functions are added to improve built-in support for mathematical operations.

HPF extends Fortran 90 by augmenting it with data distribution directives. The important directives are *align* and *distribute.* These two directives together specify how the shared arrays are distributed among processors. Let's consider the simple HPF program shown in Figure 2.1. The *processors* directive specifies the geometry of the logical processor space: the example program declares a vector of four processors. Note that this is *not* the geometry or the organization of the actual, physical processors; the processors directive only indicates the shape and size of logical processor grid desired by the programmer. The *template* directive indicates the shape of the logical space the user wants to work in. In our example, we have a one dimensional template that is identical to the three arrays. The *align* directive specifies how different arrays need to be aligned with respect to templates and, thus, also with respect to each other. The specification here indicates that the arrays are aligned perfectly to each other. (Chapter 3 presents an example where the arrays are not perfectly aligned.) Finally, the *distribute* directive specifies how the template and, therefore the arrays, are partitioned among the processors. In our example, the template and the three arrays are distributed with *block* distribution; Section 2.1.1 explains the *block* and other regular distributions supported by HPF.

2.1.1 Data distributions

HPF supports three principal types of regular data distribution directives – where regular refers to the uniformity of distribution: *block*, *cyclic*, and *cyclic(k)* (also called block-cyclic). These directives partition arrays among the processors in a parallel computer. A distribute directive is associated with a particular dimension of an array

```
real a(100), b(100), c(100)
processors p(4)
template t(100)
align with t :: a, b, c
distribute t(block) onto p
a(2:98) = b(1:97) + c(1:97) + a(4:100)
```

Figure 2.1 Example program in HPF.

and indicates how the array is partitioned along that dimension. A *block* distribution along an array dimension indicates that the array is partitioned along that dimension into a set of equal length intervals, one for each processor assigned to the array dimension. *Block* distributions are the distribution of choice for the nearest neighbor, stencil-based computations. A *cyclic* distribution along an array dimension indicates that the data is partitioned into unit intervals which are then assigned to processors in a round-robin fashion. *Block* and *cyclic* distributions are just common cases of the more general *cyclic(k)* distribution, which specifies the length of the intervals into which an array dimension is partitioned. *Cyclic(k)* distribution partitions the array dimension into intervals (blocks) of size k and then assigns these intervals to the processors in a cyclic fashion. Figure 2.2 shows an examples for each of the three distributions. *Cyclic* and *cyclic(k)* distributions are useful for writing efficient and load-balanced dense matrix algorithms [25].

2.2 The Fortran D95 compiler

The Fortran D95 compiler, being developed at Rice University, is a research and prototype compiler for HPF. A goal of the Fortran D95 compiler is to achieve performance comparable to or better than that extracted extracted by hand-compiled programs. To achieve this, the compiler concentrates on three areas:

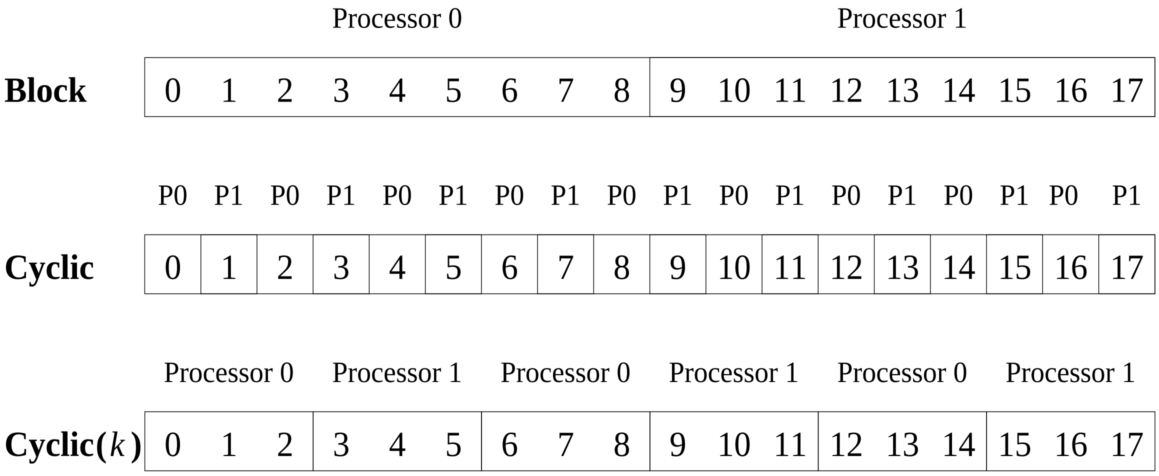


Figure 2.2 Regular distributions supported by HPF.

1. Sophisticated computation partitioning,
2. Aggressive communication analysis, and
3. Program transformations.

We use a sophisticated analysis of the program to select the best computation partitioning for each program statement. To do this, we analyze each program statement and select the work partitioning scheme that minimizes interprocessor communication. Next, we perform aggressive analysis and program transformations to optimize communication.

Another goal of the Fortran D95 compiler is to be compile HPF for a variety of architecture classes. Towards this end, we are currently targeting both the distributed-memory systems (using Message Passing Interface [26], MPI, library for message passing routines) and hardware DSM systems such as Convex Exemplar [21] as well as the software DSM systems such as TreadMarks [55].

Figure 2.3 shows the organization of the Fortran D95 compiler. This section outlines different phases of the Fortran D95 compiler in order to present the context

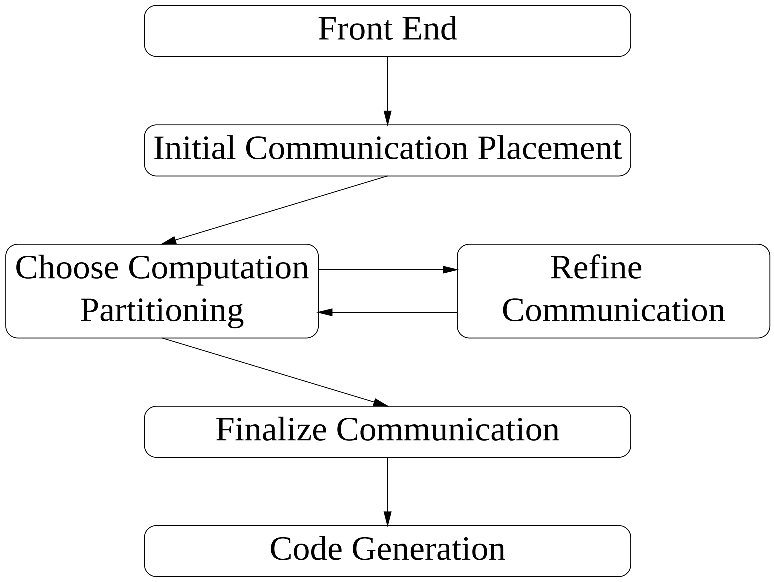


Figure 2.3 Organization of the Fortran D95 compiler.

under which the communication generation techniques have been implemented. The example program shown in Figure 2.1 is used to illustrate different phases of the compiler.

Communication generation is performed during the *refine communication* and *finalize communication* phases of the compiler (Figure 2.3).

Front end

The compiler takes the input program and converts it into an intermediate form; this requires constructing abstract syntax tree (AST), constructing control-flow graph (CFG), converting the program into static single assignment (SSA) form, constructing interval-flow graph, and performing value numbering.

In addition, the Fortran 90 style array expressions in the input program are converted into Fortran 77 loops. This translation, known as scalarization, preserves the semantics of the input program by inserting temporary array variables, if required. Figure 2.4 shows our example program after scalarization; the translated program contains compiler generated temporary array a$tmp, which is of the same size and distribution as the array a.

In the next four phases, the compiler divides the work (that is, the sequential program) among processors and generates interprocessor communication.

Initial communication placement

This phase determines initial placement for all communication primitives. The initial placement is used to provide an estimate of communication costs to the *choose computation partitioning* and *refine communication* phases. Since at the time when *initial communication placement* phase is called, the compiler does not know which processor is responsible for computing what values, we assume that potentially *every* reference can cause non-local accesses. For example, for the example program in Figure 2.4, the compiler assumes that each of the four references can cause off-processor accesses. It determines the outer-most loop level for the placement of corresponding communication primitives. The initial communiation placement for the example program is shown in Figure 2.5.

```
real a(100), b(100), c(100), a$tmp(100)
processors p(4)
template t(100)
align with t :: a, b, c, a$tmp
distribute t(block) onto p

do i = 2, 98
   a(i) = b(i-1) + c(i-1) + a$tmp(i+2)
enddo
```

Figure 2.4 Example program after scalarization.

Since this is just a preliminary communication placement, the compiler does not perform any communication optimizations.

Choose computation partitioning

This phase is responsible for dividing work among processors. It determines the set of processors that execute every statement in the program. This phase *can* select different set of processors to execute different statements in a loop body. Unlike other HPF compilers [50, 38, 3, 14] that use either only *owner computes* rule or same computation partitioning for all the statements in a loop body, we evaluate the cost of various computation partitioning options and select the best option for each statement.

The traditional computation paritioning rule, called *owner computes* rule, makes the owner of the *lhs* reference in an assignment statement responsible for fetching the non-local data and performing the computation. However, usually, various choices are available for computation partitioning and, for certain programs, performing the computation on the owner of *lhs* reference might not always be the optimal decision. As an example, let's consider our example program in Figure 2.6. There are three choices for executing the assignment

```
Non-local read b[ ... ]
Non-local read c[ ... ]
Non-local read a$tmp[ ... ]
do i = 2, 98
   a(i) = b(i-1) + c(i-1) + a$tmp(i+2)
enddo
Non-local write a[ ... ]
```

Figure 2.5 Example program with initial communication placement.

```
Non-local read b[ ... ]
Non-local read c[ ... ]
Non-local read a$tmp[ ... ]
do i = 2, 98
  a(i) = b(i-1) + c(i-1) + a$tmp(i+2)
enddo
Non-local write a[ ... ]
```

Choice	Cost
a(i)	3
b(i-1)/c(i-1)	2
a$tmp(i+2)	3

Figure 2.6 Example program with computation partitioning choices.

statement: it can be executed either by the processors that own a(i) (that is, ON_HOME(a(i))), or ON_HOME(b(i-1)), or ON_HOME(a$tmp(i+2)). Note that since the arrays b and c are identically aligned and distributed, the computation partitioning choice ON_HOME(c(i-1)) is identical to the choice ON_HOME(b(i-1)).

After enumerating all computation partitioning choices, the compiler invokes *refine communication* phase to evaluate the relative costs of various choices.

Refine communication

The *refine communication* phase helps the *choose computation partitioning* phase to evaluate communication and other costs for various computation partitioning choices. For a given computation partitioning choice, this phase projects the corresponding communication instances from the communication map generated by the *initial communication placement* phase. The associated communication cost is then computed by considering the placement level of the communication primitives. Finally, the costs for different computation partitioning choices are compared and the most efficient partitioning for each statement in the program is selected. The selected computation partitioning represents the work that must be performed by each processor and is used to generate SPMD node codes.

By determining the communication placement for all possible communication instances in the *initial communication placement* phase, the Fortran D95 compiler avoids computing the communication requirement for different computation partitioning choices. For a selected computation partitioning choice, the *refine communication* phase simply projects out appropriate communication instances from the initial communication placement map.

For our example program, shown in Figure 2.6, the communication cost associated with the three computation partitioning choices is computed as follows. First, consider the choice ON_HOME(a(i)), which corresponds to the traditional owner computes rule. If the owner of a(i) were to perform the computation, one boundary element each for arrays b and c needs to be fetched from the left processor. In addition, two elements of array a$tmp need to fetched from the right processor. As determined by the *initial communication placement* phase, the communication primitives for all three non-local references are placed outside the loop. For the sake of simplicity, let's denote the associated communcation cost as 3. Now, if the computation is performed ON_HOME(b(i-1)) or ON_HOME(c(i-1)), the compiler will need to fetch non-local boundary elements only for a$tmp(i+2) because b(i-1) and c(i-1) are always local to each other due to their identical distribution. In addition, however, since the updated value for a(i) will be computed by a processor that does not own a(i) (for the boundary iteration), the processor will need to updated the value of a(i) on the owner processor by sending back the computed value to its right neighbor. (This communication instance is called non-local write and is explained further in Chapter 3.) However, in this case, there are only two non-local references and the corresponding communication can still be placed outside the loop. Thus, the associated communication cost is 2. Similarly, the cost associated with ON_HOME(a$tmp(i+2)) is 3, as shown in Figure 2.6. Since ON_HOME(b(i-1)) has the least associated cost, it is chosen as the computation partitioning for the assignment statement.

Note that the *choose computation partitioning* and *refine communication* phases are called iteratively by the Fortran D95 compiler till the optimal computation partitioning for every statement in the program has been selected.

Finalize communication

The compiler uses the selected computation partitionings to generate communication sets (that is, the non-local data sections corresponding to cross-processor data dependences). In this phase of the compiler, aggressive placement analysis and communication optimizations are performed to reduce communication overhead. The compiler determines the placement for communication primitives that maximizes latency hiding. It also performs various communication optimizations. Most of the techniques presented in this dissertation constitute this phase of the Fortran D95 compiler. Figure 2.7 shows the placement of the SEND and RECV communication primitives corresponding to the computation partitioning chosen in the previous phase.

Code generation

In this phase, the compiler implements the decisions taken in the previous three phases. But, first, it performs program transformations such as loop splitting for increasing the communication and computation overlap. This is achieved by separating the iterations that access only local data from the iterations that might access non-local data. The local iterations are then used to *further* hide

```
SEND a$tmp[ ... ]
RECV a$tmp[ ... ]
do i = 2, 98
   a(i) = b(i-1) + c(i-1) + a$tmp(i+2)
enddo
SEND a[ ... ]
RECV a[ ... ]
```

Figure 2.7 Example program with final communication placement.

the latency of communication primitives; as mentioned earlier, the *finalize communication* phase performs extensive data-flow analysis to detect and utilize opportunites for hiding communication latency.

After performing the program transformations, the Fortran D95 compiler generates SPMD program using Fortran 77 or Fortran 90 and appropriate communication library (for example, MPI for message passing systems). The three tasks performed in this phase are:

1. **Partition computation:** The compiler uses the selected computation partitioning to reduce loop bounds such that each processor executes only the portion of the loop it is responsible for. In addition, if required, the compiler inserts appropriate guards to enforce the selected computation partitioning.

2. **Insert communication:** Based on the communication placement determined in the *finalize communication* phase, the compiler inserts appropriate communication or synchronization code. In addition, if required, the compiler inserts code for packing and unpacking data to and from message buffers. Moreover, the compiler modifies array subsripts for non-local

references such that the references can access non-local data from buffer when required.

3. **Insert storage management code:** The compiler identifies the type and extent of non-local data accesses to calculate storage required for non-local data. The Fortran D95 compiler utilizes different options for receiving non-local data. Non-local data can be received either in-place, in a buffer, or in a hash table. Since the message sizes might be unknown at compile time, the compiler dynamically allocates storage for distributed data.

Most of the loop partitioning as well as computation and generation of packing/unpacking code is based on the **Omega Library** developed at University of Maryland [56]. Omega library is used to develop a convenient notation for expressing and manipulating integer sets. The most important functionality provided by the Omega library, as far its use in the Fortran D95 compiler is concerned, is its ability to generate efficient code to scan arbitrary convex regions represented by integer sets. The computation of communication sets and the corresponding code generation issues are described in more detail in Chapter 3.

The above-mentioned compiler phases are closely related to each other and an important issue in the compilation process is the interaction between the communication generation phase and the computation partitioning and code generation phases. For example, computing communication sets (non-local data sections) requires the knowledge of computation partitioning, and in order to compute communication sets efficiently and precisely, a representation that is suitable to both computation partitioning and communication generation phase must be used. Also, the code generation phase needs to interact with communication generation to take issues such as buffer used for communication, etc. into account.

2.3 Related work

Over the last decade, a lot of research has been done to improve the efficiency of the code generated by the compilers for data-parallel languages as well as to broaden the functionalities provided by these compilers. Initial efforts were devoted towards compiling data-parallel languages for distributed-memory systems that provided message passing libraries. Over the last few years, more and more effort is being directed towards distributed shared memory systems that provide a shared-memory abstraction over the underlying message passing systems. This section outlines some of the data-parallel compilers.

IBM HPF compiler

IBM provides perhaps the most advanced HPF compiler commercially available. The IBM compiler, called pHPF, is a native compiler for IBM SP2, a multiprocessor system based on distributed-memory architecture [38]. It performs various optimizations to improve uniprocessor performance as well. Like the Fortran D95 compiler, it supports scalarizer to convert Fortran 90 array expressions into Fortran 77 code.

Unlike the Fortran D95 compiler, pHPF paritions computations based on owner-computes rule only. However, it performs analysis to recognize and handle reductions efficiently; reductions are *one* of the computation patterns that benefit from relaxing owner-computes rule. Also, it supports data distribution and computation partitioning for *block* and *cyclic* distributions only; pHPF does not support *cyclic(k)* distributions.

It performs various communication optimizations such as eliminating redundant communication, reducing the number of messages for multi-dimensional shift communication, and coarse-grain wave-fronting. It uses novel techniques for mapping scalar variables to expose more parallelism.

However, communication placement and optimizations are based only on the data-dependence information. This restricts the application of communication optimizations to communication primitives arising from individual loop nests only. For example, redundant communication elimination does not recognize redundant messages from different loop nests.

SUIF compiler

SUIF (Stanford University Intermediate Format) compiler is targetted towards efficient compilation of dense linear-algebra, data-parallel codes [7, 4]. Unlike HPF compilers, SUIF does not require users to provide data distributions for the shared arrays. It computes data distribution based on the computation being performed; restricting focus to regular, dense linear-algebra code allows the data distribution and computation partitioning problems to be formulated and solved using linear inequalities [7]. Unlike the Fortran D95 compiler, it assigns a loop iteration to a single processor; that is, it does not evaluate computation paritioning choices for individual statements and assigns same computation partitioning to all statements in a loop body [4].

SUIF compiler uses *last write trees* (LWTs) for computing precise array dataflow information efficiently for commonly occuring special cases [74, 4]. LWT analysis is performed on individual loop nests. Due to the use of LWTs, the communication and computation code generation is limited to programs consisting of loop nests with affine array accesses and loop bounds. The technique cannot handle loops nested inside conditionals.

Communication placement and optimizations in SUIF are also based on the LWT data structure. LWTs allow a limited form of data-flow analysis and allow the loop boundary conditions to be handled efficiently. Since LWTs are based on individual loop nests, SUIF compiler does not support global data-flow analysis.

SUIF includes a package for manipulating linear inequalities. It uses Fourier Motzkin Elimination (FME) to generate loop nests corresponding to a system of linear inequalities [6]. For example, it uses FME to generate pack/send and receive/unpack loops.

PARADIGM compiler

The PARADIGM compiler, like the SUIF compiler, provides an automated means to parallelize programs written in a serial programming model [9]. In addition to automatically performing data distribution for regular computations, it optionally parses HPF data distribution directives. It supports all the three regular distributions provided by HPF [82].

The PARADIGM compiler performs computation partitioning based only on the owner-computes rule. It performs communication placement and optimizations based on dependence information.

The PARADIGM compiler represents the data being communicated with linear inequalities. Unlike the SUIF compiler, it uses an “off-the-shelf” symbolic package for manipulating linear inequalities. The PARADIGM compiler uses

*Mathematica's** symbolic manipulation capabilities to support computation partitioning, communication generation, and loop transformation [82].

Fortran D compiler

Fortran D compiler was the first generation prototype compiler developed for Fortran D language [31] – a HPF-like data-parallel language. The Fortran D compiler implemented several new compilation techniques [50, 49, 52]. However, since it was among the first data-parallel compilers developed, it had several restrictions, as well.

The compiler supported optimizations such as message vectorization, message coalescing, message aggregation, coarse-grain pipelining, and vector message pipelining. It also provided some support for interprocedural reaching decomposition and overlap area analysis.

However, the compiler handled only a single distributed array dimension; thus, it was able to extract parallelism only from a single loop level. It supported only *block* or *cyclic* distributions and array references with single index variables and unit coefficients. In addition, it did not support symbolic number of processors and loop/array bounds. These restrictions were partly because it used regular section descriptors (RSDs) to represent communication and computation partitioning (iteration) sets.

The Fortran D compiler used owner-computes rule as the default computation partitioning scheme. In addition, it also provided very limited support for relaxing owner-computes rule in case of certain special patterns. All the communication placement decisions as well as optimizations were based on dependence information.

**Mathematica* is registered trademark of Wolfram Research Inc.

Vienna Fortran compiler

Vienna Fortran compiler, along with the Fortran D compiler, were among the first efforts to develop a compiler for multiprocessor system. The source language used by this project was Vienna Fortran [16, 17]; one of the predecessors of HPF. The Vienna Fortran group pioneered various compilation techniques [33, 35, 86].

The compiler introduced owner-computes rule and message vectorization. It also introduced the concept of overlap areas for regular, stencil-based computations. However, like the Fortran D compiler, it was restricted in its scope. It supported only *block* and *cyclc* distributions. Also, like the Fortran D compiler, the communication placement and optimizations were performed based on the dependence information.

Others

A lot of other research projects have contributed significantly to the field of data-parallel compilation. We briefly describe some of these.

PREPARE is a subset-HPF compiler, currently being developed at University of Vienna. The compiler supports Fortran 90 array statements and is designed to provide a portable interface to a variety of parallel architectures. The compiler partitions computations based on owner-computes rule only. Though it does not support data-flow analysis for communication placement, the compiler splits loops into local and nonlocal parts and uses the local part to hide communication latency [11].

Fortran 90D compiler was originally targetted towards providing Fortran 90 support in addition to the data distribution directives (this was before HPF was developed) [19]. Over the last few years, the Fortran 90D compiler has developed a compilation framework to provide an efficient, portable run-time library. The compiler includes efficient libraries for redistribution as well as for supporting scalable I/O and out-of-core computations [13, 12].

The compilation effort at University of Maryland has been directed at towards providing providing efficient run-time support for irregular references [45, 42]. They have developed PARTI/CHAOS run-time library to support computation and communication generation efficiently [23]. Over the last few years, they have developed techniques that are useful for compiler analysis also; the Fortran D95 compiler uses the Omega Library develoed at University at Maryland for representing communication and iteration sets as well as a basis for code generation [57].

Chapter 3

Communication Analysis

Communication analysis is the first step in the communication generation process. Communication analysis includes identification of references that access non-local data, representation and computation of the non-local data (communication) sets that need to be fetched from a remote processor, and collection of information related to non-local accesses. This information is required in later phases, particularly in the code generation phase. This chapter discusses the communication analysis techniques implemented in the Fortran D95 compiler. It also describes the new communication set representation used by the Fortran D95 compiler.

As described in Section 2.1, HPF allows shared arrays to be partitioned among processors using *align* and *distribute* directives. In HPF terminology, a processor *owns* the section of the data (shared array) that is assigned to it. The owner processor is made responsible for maintaining the updated values of the data sections assigned to it. Further, the section of data that is assigned to a processor is *local* to that processor while the remaining sections are *non-local* (in other words, off-processor).

A references that accesses non-local data can either correspond to a *rhs* reference or a *lhs* reference. The references that access non-local data are referred to as "non-local" references throughout this dissertation. A non-local *rhs* reference reads non-local data; thus the corresponding communication is called *non-local read*. A non-local *lhs* reference, on the other hand, writes to data elements that are owned by some other processor(s); thus the corresponding communication is called *non-local write*.

Figure 3.1(a) shows an example of non-local write communication instance. Arrays a and b are distributed with *block* distribution across two processors. Assume that the assignment is performed by the processors that owns b(i). Now, since processor 0 owns b(1:4), it assigns b(1) to a(2), b(2) to a(3), b(3) to a(4), *and* b(4) to a(5). However, since a(5) is owned by processor 1, processor 0 need to send the updated value of a(5) back to processor 1. The resulting communication is shown in Figure 3.1(b), where solid, processor crossing arrow corresponds to interprocessor communication. This is an important feature of the Fortran D95 compiler: it allows processors to update array sections that they don't own. But, the Fortran D95 compiler requires that non-owner processors send back the updated section to the owner processor. In other words, it does not allow a non-owner processor to hold updated values and make *it* send them to another non-owner processor when required [39].

In the Fortran D95 compiler, communication instances are represented using *implicit representation* during the computation partitioning and communication optimization phases. Only when more detailed information is required, for example in the code generation phase of the compiler, *explicit* form of communication representation is generated. In the following sections, we describe both the implicit and explicit representations used by the Fortran D95 compiler.

3.1 Implicit representation

As mentioned above, before the code generation phase, communication sets are represented using *implicit communication descriptors*. Implicit descriptors represent communication sets using:

1. list of non-local reference(s),

2. computation partitioning, and

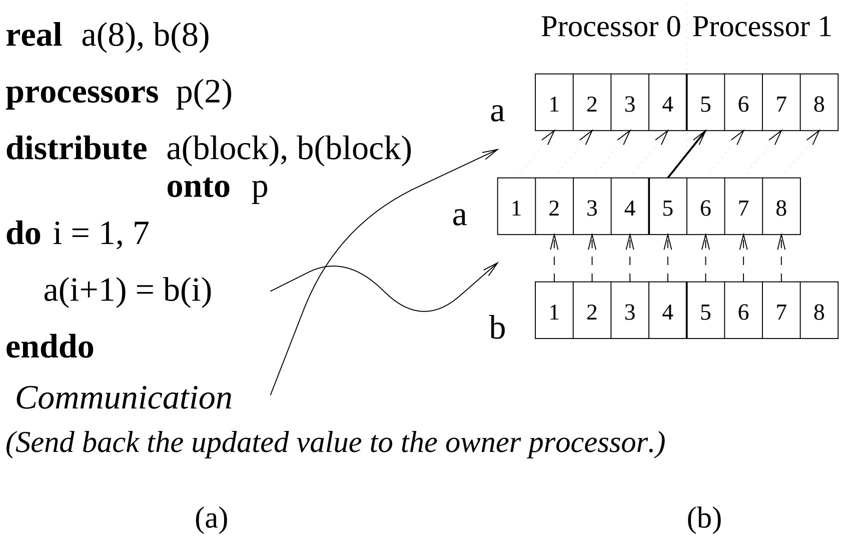


Figure 3.1 Non-local write communication.

3. whether the communication is due to non-local read or non-local write.

Every non-local reference has an associated communication descriptor that represents the communication corresponding to the reference. However, there can be many references associated with the same descriptor because of the message coalescing optimization, which combines overlapping (including identical) messages into a single message (Chapter 6). In other words, before messages are coalesced, there is an one-to-one correspondence between implicit descriptors and non-local references; however, afterwards, a communication descriptor can contain a list of non-local references.

Non-local references are used to extract the variable name, subscripts (if not a scalar variable), and the corresponding distribution information. Thus, the list of non-local references (with the relevant loop context) together with the computation partitioning information and communication type (that is, whether non-local read or write) provide the necessary information for computing both the explicit communication representation as well as the processors involved in the communication.

For example, as shown in Figure 3.1, processor 0 assigns to a(2:5). However, according to the *block* distribution, it owns a(1:4). Thus, it needs to send a(2:5) − a(1:4) = a(5), where "−" is the set difference operator, back to the processor that owns a(5) – that is, processor 1. This informally describes how implicit communication descriptor is used to compute communication sets; Section 3.3 gives a detailed and formal description of this process.

Further, since we assume that the computation is performed by the processor that owns b(i) (that is, ON_HOME(b(i))), the processors corresponding to ON_HOME(b(i)) must send back the computed value to the processors that own a(i+1) (represented as ON_HOME(a(i+1))). In other words, for non-local writes, computation partitioning in the implicit descriptor gives the set of processors that need to execute SEND primitives while ON_HOME(non-local reference) gives the set of processors that need to execute the corresponding RECV primitive.

Now, let's assume that in Figure 3.1, the processor that owns a(i+1) performs the computation; in other words, the the assignment is executed ON_HOME(a(i+1)). In this case, b(i) corresponds to a non-local read reference. The processor that owns b(i), that is ON_HOME(b(i)), must send the value of b(i) to ON_HOME(a(i+1)) (the processor that accesses it). Thus, for non-local reads, computation partitioning in the implicit descriptor corresponds to the set of processors that execute the RECV primitive while ON_HOME(non-local reference) gives to the set of processors that execute the corresponding SEND primitive.

3.2 Message classification

An additional information stored in implicit communication descriptors is communication pattern. The communication pattern encapsulates the pattern of non-local data transfer among the processors.

```
real a(12), b(0:11), c(12)
processors p(4)
template t(12)
align a(i) with t(i)
align b(i-1) with t(i)
align c(i) with t(i)
distribute t(block) onto p

do i = 1, 11
   a(i) = b(i) + c(1)
enddo
```

Figure 3.2 Example program to illustrate message classification.

Before performing message classification, the compiler first translates all references from array space to the template space. For example, consider the program fragment shown in Figure 3.2. As shown, arrays a and c are perfectly aligned while the *i*th element of array b is aligned with the *(i+1)*th element of template t. The three arrays are distributed using *block* distribution across four processors. Let's assume that the assignment statement is executed ON_HOME(a(i)). Since arrays a and c are aligned perfectly with the template t, the subscripts for the references a(i) and c(1) are the same in the two spaces. However, b(i) gets translated to b_t(i+1) in the template space. Figure 3.3(a) shows the program fragment in the template space. (Note that the array space to template space translation is performed during non-local reference identification also, but neither that nor the translation process itself are part of this dissertation.)

From Figure 3.3(b), we can see that both b_t(i+1) and c_t(1) are non-local references. The resulting communication for both the arrays is shown in Figure 3.3(b) where the solid, processor crossing arrows correspond to interprocessor communication. This example illustrates the need for translating references to the template space before performing message classification.

Message classification captures the data movement pattern by classify the messages into a set of *logical* patterns. Note that the communication pattern is the *logical* communication pattern and *not* the physical communication pattern. In other words, the pattern need not correspond to the communication primitive that is used to implement the communication on the target architecture. In fact, the target architecture might not even support a communication primitive similar to the logical pattern. In addition, even if the corresponding communication primitive exists, the code generation phase of the compiler might decide not to use to the logical communication pattern information and implement the communication using some other appropriate communication primitive.

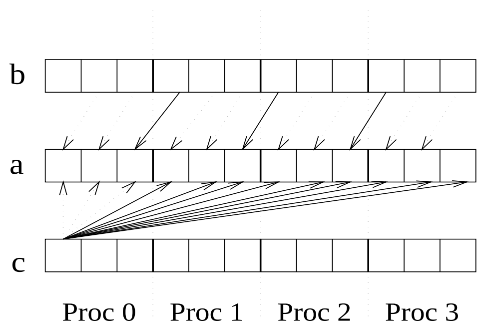
Communication can be classified as either one-to-one, one-to-many, many-to-one, or many-to-many where the first one/many corresponds to the number of the sender processors while the second one/many corresponds to the number of the receiver processors. The Fortran D95 compiler refines these patterns further. For example, one-to-one communication pattern is further classified as either single send/receive (in which only one processor sends the data and only one data receives it) or shift (in which, a processor communicates with its neighboring processor(s)). One-to-many

do i = 1, 11

  $a_t(i) = b_t(i+1) + c_t(1)$

enddo

(a)



(b)

Figure 3.3 Shift and broadcast communication patterns.

pattern is further refined by evaluating whether *all* processors get involved in the communication. Thus, one-to-many is refined to be either broadcast (one-to-all) or multi-spread (one-to-many, but not all) patterns. Similarly, many-to-many patterns can also be refined. In addition, the Fortran D95 compiler performs extensive analysis to identify the reduction communication pattern, which is a special form of many-to-one communication pattern [73].

Since the communication pattern depends on both the number of sender and number of receiver processors, we must know not only the non-local reference but also the corresponding computation partitioning in order to compute the pattern for a communication descriptor. As described in Section 2.2, since computation partitioning for the statements in the program is not known during initial communication placement phase, communication pattern information is not computed during that phase.

Message classification is performed *on-demand* after the computation partitioning for the statements in a program is selected. Besides using the message classification information in the code generation phase of the Fortran D95 compiler, we also use it to identify the messages that can be combined in the message coalescing optimization (refer to Section 6.2 for more details). In the current implementation of the Fortran D95 compiler, only the messages with identical communication patterns are combined. For example, a message with broadcast communication pattern is not combined with a message with shift communication pattern. This is because, although the two communication instances might cause an overlapping data section to be communicated, the set of processors involved in the communication are different and, on distributed-memory systems, combining these messages often does not lead to any benefit. For example, in Figure 3.2, if the last term were b(1), then although the broadcast and shift communication instances for processor 0 would have overlapping data, the compiler would not have been able to derive any benefit by combining the two communication instances. Further research needs to be done to identify the cases when the compiler can derive benefits by coalescing messages with different patterns.

3.3 Explicit representation

When required, for example during the code generation phase of the compiler, the implicit communication descriptors are converted into explicit communication sets. A lot of research has been done to represent array sections efficiently [29, 8, 46, 7, 6, 5, 82]. Explicit representations can be classified as either precise or imprecise representations. The latter category of representations tend to sacrifice precision for efficiency. Imprecise representations use closed-form formulae or a restricted class of geometric patterns to approximate given array section with a convex region. This kind of representation is suitable for representing array sections during the dependence analysis phase of the compiler; in fact, most of these techniques were developed to support efficient array section representation during the dependence analysis phase of vectorizing compilers.

However, for communication set representation, we often need an exact representation. For example, the array section that is updated non-locally (that is, array elements corresponding to non-local write), needs to be represented precisely. An approximate representation can cause old and incorrect array values to be sent back to the owner processor.

A precise representation uses a set of inequalities to represent array sections. Clearly, this is the most powerful representation and can be used to represent any set. The problem of representing precise array data-flow information was first formulated by Feautrier [29, 30]. Feautrier proposed a parametric integer programming algorithm that can find such perfect information in the domain of loop nests where the loop bounds and array indices are affine functions of loop indices. However, operations such as intersection and union have exponential time complexity in the worst case. But, it has been shown that most scientific programs have simple access patterns and the operations on the corresponding array sections often require only simple inequality manipulations [57].

The two new precise representations defined in the context of the Fortran D95 compiler are:

1. **Periodic access sequence:** for *cyclic(k)* distribution, data sets are concisely and efficiently represented using periodic sequences.

2. **Omega-based set representation:** used as a general integer set representation for communication sets (that is, when they can not be represented precisely using traditional representations).

3.3.1 Previous work

This section summarizes existing representations for array sections.

Regular section descriptors

Regular section descriptors (RSDs) are the simplest form of array section representations. RSDs use Fortran 90 style triplet notation to represent *regular sections*: $[l_i : u_i : s_i, \ldots]$ where l_i, u_i, and s_i are the lower bound, upper bound, and step in the ith dimension of the RSD, respectively. Intuitively, regular sections correspond to rectangular or right-triangular array sections and their higher dimension analogs [15, 46]. RSDs can also represent regular sections with constant stride. RSDs were originally developed for summarizing side effects across procedure boundaries and represent array sections using an imprecise representation. RSDs were used extensively in the Fortran D compiler [85]. The union and intersection of RSDs can be computed in time proportional to the number of section dimensions. RSDs are closed under intersection but not union.

Data access descriptors

Data access descriptors (DADs) are more general than RSDs and can be used to represent a convex polyhedra that is bounded by hyperplanes that are either orthogonal to the axis or at 45^o angles with a pair of axes [8]. Thus, unlike RSDs, DADs can represent triangular array sections as well as diagonal and banded diagonal array sections. The intersection and union operations on DADs can take time proportional to the number of dimensions of the section. DADs are closed under intersection. The union operation returns a conservative approximation of the unioned DAD sections.

Processor tagged descriptors

Processor tagged descriptors (PTDs) are used in the PARADIGM compiler [83] to incorporate processor information along with the array section information. PTDs are useful in expressing the distributed array sections; bounds of an array section for a processor can be parameterized with the processor number. The array section representation component of PTDs is more general than DADs and a wider range of sections with arbitrary boundary angles (not just multiples of 45^o) as well as some non-convex regions can also be represented. PTDs are closed under intersection while union and difference can result in a list of sets.

Fourier-Motzkin elimination

A lot of researchers have proposed precise array section representations based on the Fourier-Motzkin elimination method [29, 6, 7, 57, 82]. These representations represent array sections with a set of linear inequalities. Operations on array sections, such as union, intersection, and difference are performed by manipulating the linear inequalities; thus, these operations can take exponential time in the worst case. However, empirical evidence has shown that Fourier-Motzkin elimination is quite efficient for commonly encountered scientific codes [78].

3.3.2 Periodic access sequences

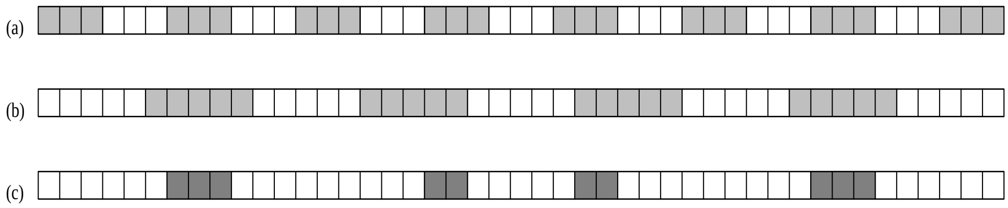
Figure 3.4(a) shows array a distributed with *cyclic*(3) distribution across two processors. Processor 0 owns elements [1:3], [7:9], [13:15], and so on. Since this set is an union of regular sections (in other words, it is a non-convex region), it can not be represented using either the RSD, DAD, or PTD representations.

A possible notation for describing sequences of indices that arise with *cyclic(k)* distributions is a quadruplet notation, [l:u:k:pk], based on the Fortran 90 triplet notation. The components are, respectively, lower bound, upper bound, block size, and the cycle length which is equal to the product of the block size and the total number of processors. The array section owned by processor 0 can be represented as [1:n:3:6] in quadruplet notation. The quadruplet notation can also represent *block* and *cyclic* distributions by setting $k = \lceil \frac{n}{2} \rceil$ for a *block* distribution and $k = 1$ for a *cyclic* distribution. Sections of multidimensional arrays can be represented using an instance of the [l:u:k:pk] notation for each array dimension.

However, as can be seen from Figure 3.4(c), *cyclic(k)* sets are not closed under intersection. In other words, intersection of two *cyclic(k)* sets need not result in a *cyclic(k)* set. In addition, the figure shows that the quadruplet notation is insufficient to represent the *cyclic(k)* sets; the result of the intersection can not be represented using the quadruplet notation.

This section now describes the *periodic access sequence* method that can be used to represent *cyclic(k)* array sections efficiently. To understand the basis of the repre-

real a(n), b(n)
distribute a(*cyclic*(3)), b(*cyclic*(5))
do i = 1, 45
    a(i) = $\mathcal{F}$(b(i))
enddo



(a) Elements of b needed by processor 0
(b) Elements of b owned by processor 1
(c) Elements processor 1 needs to send to processor 0 $\equiv$ (a) $\cap$ (b)

Figure 3.4 Data ownership and communication for *cyclic(k)* distributions.

sentation, let's consider the following example:

real a(0:95)
processors p(3)
distribute a(*cyclic*(4)) **onto** p
do i = 0, 95, 5
    a(i) = ...
enddo

Array a is distributed block-cyclically among 3 processors, with a block size of 4. The layout of array a in processor memories is depicted in Figure 3.5. The figure shows the one-dimensional array a arranged as a two-dimensional matrix; where each row of the matrix has $p\,k$ elements; p is the number of processors and k is the block size. In other words, in Figure 3.5, we have stacked the blocks assigned to a particular processor one under another. This view of the *cyclic(k)* distribution was first suggested by Chatterjee et al. [18].

Processor 0				Processor 1				Processor 2			
$\boxed{0}$	1	2	3	4	$\boxed{5}$	6	7	8	9	$\boxed{10}$	11
12	13	14	$\boxed{15}$	16	17	18	19	$\boxed{20}$	21	22	23
24	$\boxed{25}$	26	27	28	29	$\boxed{30}$	31	32	33	34	$\boxed{35}$
36	37	38	39	$\boxed{40}$	41	42	43	44	$\boxed{45}$	46	47
48	49	$\boxed{50}$	51	52	53	54	$\boxed{55}$	56	57	58	59
$\boxed{60}$	61	62	63	64	$\boxed{65}$	66	67	68	69	$\boxed{70}$	71
72	73	74	$\boxed{75}$	76	77	78	79	$\boxed{80}$	81	82	83
84	$\boxed{85}$	86	87	88	89	$\boxed{90}$	91	92	93	94	$\boxed{95}$

Figure 3.5 Elements of array a with *cyclic*(4) distribution on 3 processors. The squares mark the array elements accessed by a($5i$) ($i \geq 0$).

Once we arrange the block-cyclically distributed array as shown in Figure 3.5, it can be seen that there exists a pattern in the array accesses. Notice that the pattern starts repeating after the iteration that accesses a(60) on processor 0. Subsequent iterations completely mimic the pattern of the previous iterations. In fact, it can be shown that for arbitrary stride s, number of processors p, and block size k, there exists a pattern that repeats with period spk.

This is the basis of *periodic access sequences:* the accesses in the first *cycle* together with the *period* of the cycle completely specify the sequence of accesses made by processors. For example, the array elements accessed by processor 0 in Figure 3.5 can be specified with the sequence [0,15,25,50] and the period 60.

The operations such as union and intersection as well as the use of periodic access sequences in communication set computation are described in Chapter 7.

3.3.3 Omega based integer set representation

As described earlier, researchers have used various representations for expressing array sections precisely. Though early research was targeted towards in-house development of tools for expressing and manipulating linear inequalities, recent research has been directed towards using general integer set manipulation packages. An example of this is the PARADIGM compiler which uses the in-built functionality of *Mathematica* to implement linear inequalities required for expressing array sections [82].

The Fortran D95 compiler utilizes an existing package for expressing the integer sets corresponding to computation and communication sets. Thus, we directed our intellectual efforts towards developing efficient mode of expressing and computing communication sets and *not* on the development of the necessary underlying machinery for manipulating integer sets. We use the Omega library [56] for expressing integer sets and tuple relations. The representation is used for a wide range of applications including analysis for communication set computation as well as generating code for communication primitives (packing and unpacking code) and program transformations.

The Omega library is a set of C++ classes for manipulating integer tuple relations and sets. The Omega library represents relations and sets using Presburger formulas (possibly with limited uses of uninterpreted function symbols). Presburger formulas are formulas that can be constructed by combining affine constraints on integer variables with logical operations $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$ [56].

To compute the data section that needs to be communicated, we express the communication set as an integer set: the set of array indices that need to be communicated. Communication sets corresponding to scalars are represented as sets with zero set variables.

We first compute the array section owned by a processor. Note that this set is the only set that depends directly on the data distribution and alignment directives. All the remaining sets depend only on non-local references and the appropriate computation partitioning.

As an example, recall that in Figure 3.1, the arrays a and b are distributed with *block* distribution. The owned sections for the two arrays are:

$$Local_{\mathrm{a}} = \{\ [i] : myid * block_{\mathrm{a}} + 1 \leq i \leq (myid + 1) * block_{\mathrm{a}}\ \} \text{ and}$$
$$Local_{\mathrm{b}} = \{\ [i] : myid * block_{\mathrm{b}} + 1 \leq i \leq (myid + 1) * block_{\mathrm{b}}\ \}.$$

Next, we find the set of iterations that are executed by a processor. In Figure 3.1, since we assume that the computation is performed ON_HOME(b(i)), the local iteration set is:

$$LocalIterations = \{\ [i] : myid * block_{\mathrm{b}} + 1 \leq i \leq (myid + 1) * block_{\mathrm{b}}\ \}.$$

To compute the set of array elements accessed by the non-local reference, we define a mapping from the iterations to the array space. (Actually, the mapping is from the iteration space to the template space. Since Figure 3.1 assumes perfectly aligned arrays, the mapping from iteration space to array space is identical to the mapping from that between iteration and template spaces.) For the non-local reference a(i+1), the mapping is:

$$NonLocalRef_{\mathrm{a}} = \{\ [i] \rightarrow [i + 1]\ \}.$$

We apply this mapping to the *LocalIterations* set to determine the set of data elements accessed by the non-local reference.

$$DataAccessed_{\mathrm{a}} = \{\ [i] : myid * block_{\mathrm{b}} + 2 \leq i \leq (myid + 1) * block_{\mathrm{b}} + 1\ \}.$$

This set is *not* just the non-local data elements; it corresponds to both the local and non-local data elements accessed by the reference a(i+1).

To compute the non-local array elements accessed by a processor, we subtract off the local array section, that is $Local_{\mathrm{a}}$ set. In this case, since $block_{\mathrm{a}}$ and $block_{\mathrm{b}}$ are identical, we get:

$$NonLocal_{\mathrm{a}} = \{\ [i] : i = (myid + 1) * block_{\mathrm{b}} + 1\ \}$$

This is the set of data that needs to be communicated.

3.4 Summary

This chapter presented the communication analysis techniques used by the Fortran D95 compiler. The compiler uses implicit communication descriptors to uniformly represent all communication requirements in a program. For example, the communication requirement for both the regular and irregular references can be represented using implicit descriptors. The implicit descriptors also facilitate uniform implementation of communication placement and optimizations. The communication placement framework described in the next chapter determines placement for implicit descriptors. Moreover, communication optimizations are also performed on implicit descriptors; the descriptors are used to derive the necessary information such as the communication pattern as well as the sending and receiving processors.

Implicit descriptors encapsulate the information necessary during later phases of the compiler. In addition to deriving the communication pattern information, it is also used to derive explicit communication set information. Both the communication pattern and explicit sets are used during code generation: communication patterns to select appropriate communication primitives and communication sets to generate the message packing and unpacking loops.

The chapter also describes two new explicit communication set representations. The periodic access sequence representation is useful for representing communication and access sets corresponding to *cyclic(k)* distributions. It is a compact representation that allows exploiting the repetitive access pattern inherent in array accesses with *cyclic(k)* representations; more general representations, such as the Omega-based representation described in the chapter, do not provide any mechanism to record and utilize this information. Chapter 7 describes efficient local address as well as communication generation techniques based on the periodic access sequence described in this chapter. The Omega-based representation, the second explicit set representation described in this chapter is used as a general-purpose representation for communication sets. A lot of other researchers have also used similar representations before. In the Fortran D95 compiler we use an off-the-shelf package as a basis for this representation.

One of the issues that has not been addressed in this dissertation in the communication analysis for irregular references. The current implementation of communication descriptors is targeted essentially towards regular references. Implicit communication descriptors need to be extended to incorporate the information corresponding to inspectors for irregular references. However, care was taken to design the descriptors such that they can be extended to support irregular communication instances as well.

Chapter 4

Communication Placement

Chapter 3 describes the analysis used by the Fortran D95 compiler to identify non-local references and to compute the relevant communication-related information. This chapter describes a communication placement framework that determines the placement of communication primitives corresponding to non-local references. As discussed in Chapter 1, a performance-critical task of communication generation is to minimize communication overhead. The communication placement framework described in this chapter treats communication primitives as candidates for code motion and determines a judicious placement of communication primitives [63].

The framework has been implemented in the Fortran D95 compiler where it is used to determine the placement of communication primitives (SEND and RECV) for distributed-memory systems. It reduces communication overhead by decreasing communication frequency. Frequency is reduced by hoisting communication to the outermost placement location. In addition, it reduces overhead by eliminating redundant communication and by hiding communication latency. Latency is hidden by determining non-atomic placement of SEND and RECV primitives. SENDs are initiated as early as possible and RECVs are executed as late as possible to expose the opportunities for overlapping communication with computation. Moreover, the framework ensures that every program execution path contains matching SEND and RECV pairs; that is, it ensures that SENDs and RECVs are *balanced* [44].

The important features of the communication placement framework presented in this chapter are as follows:

- First, the framework incorporates data-dependence information in the placement analysis to simplify the data-flow equations. This approach differs from previous global data-flow techniques [40, 44, 59], which use dependence information only to evaluate whether different references have data-dependent (overlapping) communication sets.

- Second, the framework shows that placement can be determined by a sequence of simple uni-directional analyses. Unlike previous techniques, our framework uses separate data-flow equations to determine the placement of SEND and RECV primitives. Also, the framework has independent equations to address separate concerns such as safety, balance, etc.

- Finally, the framework allows incorporating communication optimizations such as message coalescing (which combines overlapping messages), elimination of redundant communication elimination, and message pipelining optimization. Message pipelining optimization is an important optimization for programs with loop-carried dependences [48]. Chapter 6 describes how the data-dependence information is integrated with data-flow analysis to improve the precision and scope of these optimizations.

The rest of this chapter is organized as follows. Section 4.1 introduces the graph structure and predicates used by our framework. Section 4.2 presents an overview of our framework. Sections 4.3 and 4.4 present equations for SEND and RECV placement. Section 4.5 shows the linear time complexity of our data-flow algorithm. Sections 4.6 and 4.7 describe the *balance* placement criterion and its use in ensuring the correctness of communication placement. Section 4.8 presents results to demonstrate the benefits of latency hiding. Section 4.9 compares our framework with previous work and Section 4.10 summarizes the framework.

4.1 Preliminaries

This section describes the graph structure and the predicates used by our framework.

4.1.1 Interval-flow graph

The communication placement framework (Section 4.2) is based on interval analysis [2]. Interval analysis incorporates program structure into data-flow equations to enable their non-iterative and efficient solution. However, unlike classical interval analysis, we do not construct a sequence of interval graphs by recursively collapsing intervals into single nodes. Instead, we use an *interval-flow graph*, which is the control-flow graph with an interval structure imposed on it. An interval is just a collection of nodes; it is defined formally below. While solving data-flow equations, we summarize appropriate information for an interval and record it in the interval header. Our interval-flow graph is similar to that used by the GIVE-N-TAKE framework [44]; it differs only in the way critical edges are eliminated (described below).

The important properties of the interval-flow graph, $G = (N, E)$, are as follows. We assume that G is reducible; that is, each loop has a unique header node. In this chapter we assume that G corresponds to a structured program; however, the placement framework can be extended to handle jumps out of loops.

We assume that G uses Tarjan intervals [84], where a Tarjan interval $T(h)$ is a set of control-flow nodes that correspond to a loop body in the program text. A Tarjan interval has a unique header node h, where $h \notin T(h)$. Tarjan intervals include only nodes that are part of a loop; that is, together with the header node h, the nodes in $T(h)$ form a strongly connected region. As described by von Hanxleden [43], this differs from the Allen-Cocke intervals [28, 53, 58] used in classical interval analysis. Allen-Cocke intervals include nodes whose predecessors are all in $T(h)$; that is, they might include an acyclic structure dangling off the loop.

Interval-flow graph differs from a control-flow graph in the way in which edges $\in E$ are classified. An edge in E is classified as entry, back, or forward edge:

Entry edge: An edge from an interval header to a node within the interval.

Back edge: An edge from a node within the interval to the interval header.

Forward edge: An edge that is neither entry or back edge.

Note that a forward edge has both the source and sink of the edge in the same interval; forward edges do not cross interval boundaries.

We assume that G does not have *critical edges*. A critical edge connects a node with multiple successors to a node with multiple predecessors [67]. Figure 4.1 illustrates how critical edges can lead to incorrect communication placement. Figure 4.1(a) shows the earliest placement of SEND x and the latest placement of RECV x. Clearly, this is an incorrect placement because if the right branch is taken, then the RECV x will not have a matching SEND x.

It is well known that the code motion process *may* be blocked by critical edges [24, 27, 79]. Critical edges are eliminated by splitting edges as follows: every edge leading to a node with more than one predecessor is split by inserting a *synthetic node*.† Figure 4.1(b) shows how critical edges are eliminated and how that leads to the correct, balanced placement of communication primitives. (Section 4.6 describes the *balance* placement criterion in more detail.) Note that Figure 4.1(a) corresponds to an

†The GIVE-N-TAKE framework, on the other hand, splits only the edges that connect a node with multiple successors to a node with multiple predecessors.

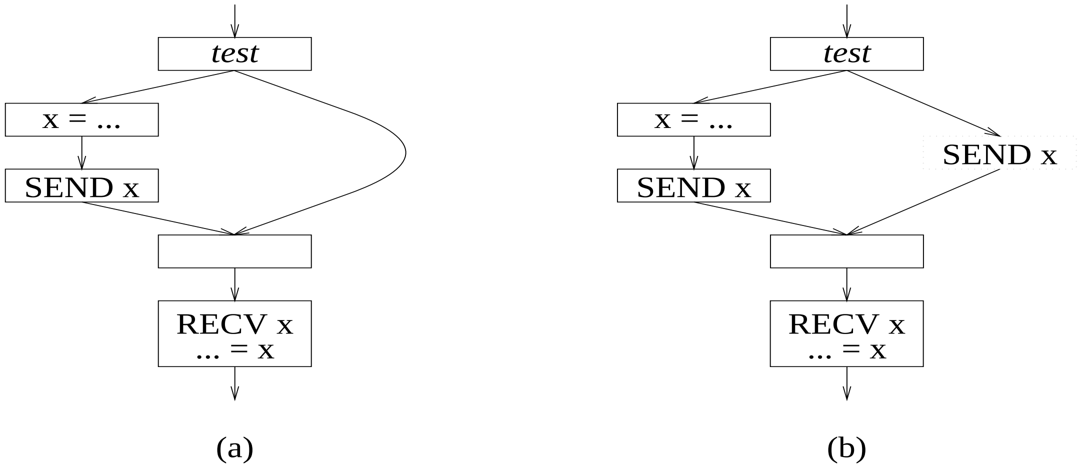


Figure 4.1 Example to illustrate how critical edges can lead to incorrect placement. The figure on right shows how to eliminate a critical edge with insertion of a synthetic node.

if ... **endif** construct while Figure 4.1(b) corresponds to an **if** ... **else** ... **endif** construct. We assume that critical edges due to this and other constructs are eliminated before placement analysis is performed. It can be proved that after the elimination of critical edges, communication (in general, code) placement can be restricted only to node entries [66].

Finally, for every non-empty interval $T(h)$, there exists a unique $g \in T(h)$ such that $(g, h) \in E$; that is, there is only one back edge out of $T(h)$. This is achieved by adding a *post-body node* to $T(h)$ [44].

The interval-flow graph shown in Figure 4.2 is used in this chapter to illustrate the communication placement framework. Though not shown in the figure, the graph is implicitly partitioned into intervals; each loop body in Figure 4.2 corresponds to an interval. For example, nodes 6, 7, and 8 constitute an interval; they belong to the interval with header node 5. We assume that all nodes at the outermost level belong to an interval with dummy header *Root* (not shown) and that N has s and

```
a = 1
if (test?) then
   do i = 2, 256
      w[i] = z[i-1]
      z[i] = ...
   enddo
   b = 2
else
   do j = 1, 1000
      x[j] = ...
   enddo
   c = 3
   do k = 1, 999
      w[k+1] = x[k]
   enddo
endif
do i = 1, 999
   w[i+1] = x[i]
   w[i+1] = y[i]
enddo
```

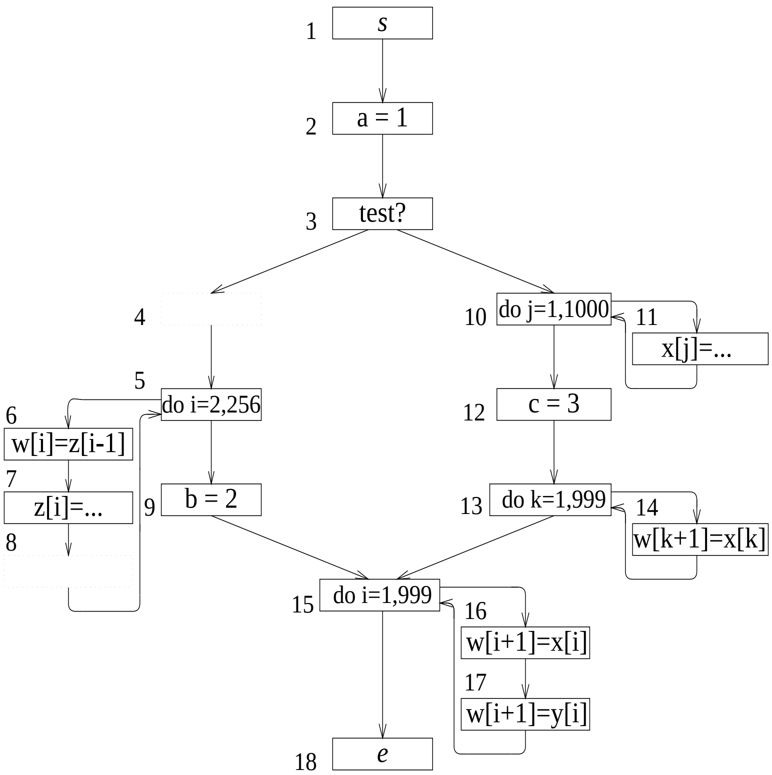


Figure 4.2 Example program and the interval-flow graph used to illustrate communication placement analysis.

e as unique start and end nodes. Note that the synthetic nodes that do not play any role in the placement are not shown in the figure. Actually, synthetic nodes are inserted between the nodes 3 and 5 (that is, node 4), 3 and 10, 9 and 15, 12 and 13, 13 and 15 and along every back edge (for example, node 8 inserted along the back edge from node 7 to node 5).

Let arrays w, x, y, and z be partitioned with the same distribution; that is, assume that the four arrays are divided identically among the processors. Also, for the sake of convenience, assume that for every assignment, the *lhs* reference is made responsible for performing the computation (that is, assume that the *owner computes* rule is used to partition the computation). Thus, the *rhs* terms in nodes 6, 14, 16, and 17 correspond to references that access remote data. The graph shows assignments to local data (nodes 2, 9, and 12), the references that need communication (nodes 6, 14, 16, and 17), and the statements that define the data being communicated (nodes 7 and 11). Note that there is a loop-carried true dependence from node 7 to node 6 on array z; thus, the communication for the reference z[i-1] needs to be placed inside the "**do** i" loop (node 5).

The references x[k] and x[i] (nodes 14 and 16) require identical non-local data. We assume that the non-local data fetched for node 14 are stored in a buffer and reused at node 16. However, the data are *not* available if the program executes the left **if** branch. When the non-local data are available along some but not all paths to a node, the data are said to be *partially* available. Our framework supports *partially redundant communication elimination*: it allows the non-local data to be fetched only once along both the branches and to be used twice along the right branch. Chapter 6 explains how this optimization is incorporated in the framework.

4.1.2 Definitions

We use the semi-lattice $L = \{\top, \bot\}$ with the operators $\cup$, $\cap$, and $\neg$ defined in the standard way. Section 4.4 extends L to support analysis for balanced placement.

Let COMM be the universe of communication descriptors. Each reference that requires communication is associated with a distinct descriptor; Section 4.2 gives more details. In the following, communication descriptor "d is referenced/modified" is used to indicate whether the data section corresponding to the descriptor is referenced/modified. For every node n and descriptor $d \in$ COMM, we define predicates: $Used(n, d) =_{df} \top$ if a subset of d is referenced (used) at n and $Transp(n, d) =_{df} \top$ if a subset of d is not modified at n (that is, n is transparent to d). For example, $Used$(14, x[k]) $= \top$, $Transp$(14, x[k]) $= \top$, $Used$(11, x[j]) $= \bot$, and $Transp$(11, x[j]) $= \bot$. For header node h, $Used(h, d)$ is set to $\top$ if some node in $T(h)$ uses a subset of d and to $\bot$ otherwise. Similarly, $Transp(h, d)$ is set to $\top$ if no node in $T(h)$ defines subset of d and to $\bot$ otherwise. For example, $Used$(10, x[k]) and $Transp$(10, x[k]) are initialized to $\bot$ and $Used$(13, x[k]) and $Transp$(13, x[k]) are initialized to $\top$. These initializations assume that the header nodes do not use or modify non-local data; a pre-analysis program transformation achieves this (Section 4.2). Without the transformation, the initialization should be done based on $T(h) \cup \{h\}$; moreover, the equation to determine the safety of header nodes (Section 4.3) also requires a minor change. Finally, $\forall d \in$ COMM, we initialize $Transp(s, d) = \bot$, $Used(s, d) = \bot$, $Transp(e, d) = \top$, and $Used(e, d) = \top$.

The array sections are analyzed during the initialization phase to determine whether communication descriptors have overlapping sections. This allows the *Used* and *Transp* variables to be initialized precisely. However, in case of irregular references (that is, references with indirection; for example: x[y[i]]), we conservatively assume that any two irregular references overlap. We inspect the subscripts of array references and initialize *Transp* variables such that the communication primitives corresponding to arrays are blocked by the statements that modify the indirection arrays as well. For example, for reference x[y[i]] and a node n that modifies an overlapping section of y, *Transp*(n, x[y[i]]) is initialized to $\perp$. Note that array sections are compared only to initialize the *Used* and *Transp* variables; later phases do *not* perform any operations on array sections.

Let $\textsc{Succs}(n)$ and $\textsc{Preds}(n)$ correspond to the set of successor and predecessor nodes of n. $\textsc{Succs}^F(n)$ and $\textsc{Succs}^E(n)$ then correspond to the forward and entry edge successors of n. Additionally, the edges induce the following traversal orders over G [44]: given a forward edge (m, n), a FORWARD order visits m before n, and a BACKWARD order visits m after n. Similarly, for an entry edge (m, n) or a back edge (m', n'), an UPWARD order visits m after n and m' before n', whereas a DOWNWARD order visits m before n and m' after n'. Let *last node* be the unique post-body node of a loop nest (the node that is the source of the back edge). Also, for node n, let $\textsc{Header}(n)$ be the header node h of the enclosing interval $T(h)$.

4.2 Overview

Before performing the data-flow analysis, all references that cause non-local accesses are identified. To each non-local reference, we associate a A distinct communication descriptor, $d \in \textsc{Comm}$, is associated with each non-local reference. For example, though nodes 14 and 16 access identical sections of array x, two separate communication descriptors are associated with them. Since there exists a unique communication descriptor for each non-local reference, we will denote a communication descriptor with the corresponding non-local reference in the dissertation.

Non-local references can correspond either to *rhs* references (non-local reads) or *lhs* references (non-local writes). Figure 4.2 has communication requirement corresponding to non-local reads only. In case of non-local write, a processor computes the value(s) that need to be assigned to off-processor array element(s). The processor assigns the computed value(s) to a local copy of the data and sends back the updated local copy to the owner of the data. Both non-local read and write communication require placement of SEND and RECV pairs. Non-local reads require the data elements to be communicated *before* they are used (that is, before the off-processor data are referenced). Non-local writes, on the other hand, require the non-local data to be communicated back to the owner processor *after* they have been computed (that is, after the data that needs to be communicated have been assigned the updated values). Thus, non-local reads and writes are duals of each other. In this chapter we will concentrate on the *before* placement problem corresponding to non-local reads. Non-local writes require a very similar analysis.

The placement framework has two phases, one each for send and receive placement. If required, either of the two phases can be executed alone. Each phase can be further divided into three subphases: first subphase initializes data-flow variables; second solves the data-flow equations; and appropriate communication optimizations are performed in the third subphase. For send placement, the second subphase involves, for each communication descriptor, (i) computing the set of safe nodes and (ii) selecting the earliest safe node along each program execution path. Partially redundant communication elimination and message coalescing optimization are performed in the third subphase of the send placement analysis (Chapter 6). Similarly, for receive placement, the second subphase involves, for each communication descriptor, (i) computing the set of balanced nodes and (ii) selecting the latest balanced node along each path. Vector message pipelining is performed in the third subphase of the receive placement analysis (Chapter 6).

The placement analysis is performed on the interval-flow graph of the original sequential program. Before performing the analysis, the interval-flow graph is canonicalized by ensuring that the loop bounds do not require non-local data. This is achieved by moving references to non-local data from the loop bounds to new assignment statements that are inserted just before the loops. The graph is transformed before starting communication placement analysis. The analysis computes and stores data-flow variables as annotations to the interval-flow graph nodes. Also, the analysis itself only determines the program locations where the communication primitives can be placed; it does not modify the interval-flow graph. Communication primitives are inserted in the graph in the subsequent code generation phase of the Fortran D95 compiler. For the sake of clarity, we will omit the guards around the communication primitives in our figures; actually, guards are generated to ensure that only the appropriate processors send and receive data.

4.3 Earliest send placement

Maximizing latency hiding requires SENDs to be placed as early as possible. We determine the *earliest* SEND placement for each communication descriptor d by first computing the set of *safe* nodes for SEND(d). We then determine the *earliest* safe node for SEND(d).

To incorporate dependence information, we use data-flow variable SAFE(n, d). It is initialized as follows. Let there be a true dependence carried by the loop corresponding to a header node h. For each communication descriptor d that is affected by the dependence, SAFE(h, d) is initialized to $\perp$. As an example, the loop-carried dependence from node 7 to node 6 in Figure 4.2 causes SAFE(5, z[i-1]) to be initialized to $\perp$. Initializing SAFE(h, d) to $\perp$ marks the header node h as *unsafe* for SEND(d) and causes the header node h to block SEND(d). Thus, the initialization causes communication to be placed inside the loop, as required. In other words, the initialization inhibits message vectorization optimization; hoisting communication to the outermost loop level is known as message vectorization in the data-parallel compiler literature [48].

In traditional code motion techniques, placement of computation at a node is considered *safe* if the computed value is used along all terminating paths from the node [76, 67]. In the context of communication placement, the safety criterion states that the $\textsc{Send}(d)$ should be placed at node n only if the communicated data are used along all terminating paths starting at n.[‡] In other words, the node n is a safe placement location for $\textsc{Send}(d)$, if it either (a) contains an use of the communicated data (that is, $Used(n,d) = \top$), or (b) does not modify the data being sent (that is, $Transp(n,d) = \top$) and all its successors are *safe* (that is, $\cap_{s \in \textsc{Succs}^F(n)} \mathrm{SAFE}(s,d) = \top$) [67]. In terms of data-flow equation:

$$\mathrm{SAFE}(n,d) = \mathrm{SAFE}(n,d) \ \cap \ [Used(n,d) \ \cup \ (Transp(n,d) \ \cap \bigcap_{s \in \textsc{Succs}^F(n)} \mathrm{SAFE}(s,d))] \tag{4.1}$$

$\textsc{Succs}^F$ is as defined in Section 4.1.2. If $n =$ *post-body node*, $\textsc{Succs}^F(n) = \emptyset$ and $\cap_{s \in \textsc{Succs}^F(n)} \mathrm{SAFE}(s,d) = \bot$, since no datum is used at the *post-body node* (recall that each interval has a unique post-body node, *post-body node*). The solution to the above equation can be found in a single $\textsc{Backward}$ and $\textsc{Upward}$ traversal of G.

Traditionally, hoisting computation out of a loop with unknown bounds is considered *unsafe* because if the loop body is not executed, it introduces a new value on some execution path(s) of the program. However, since hoisting communication out of a loop can only cause over-communication [44], we relax the safety criterion to hoist communication out of potentially zero-trip loops as follows.

[‡]The framework places send primitives only at the node start.

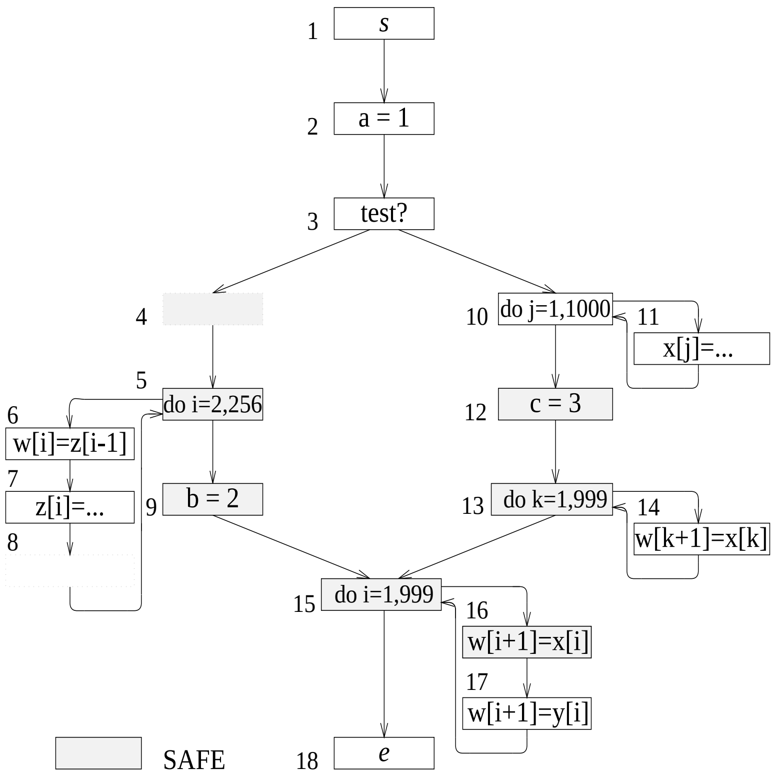


Figure 4.3 The SAFE predicate values for SEND x[i].

For header node h, not only the communication placed at the forward edge successor(s) of h can be hoisted *across* $T(h)$, the communication can also be hoisted *out* of $T(h)$. Now, $\textsc{Succs}^E(h)$ – the unique entry node successor of the header node – is the first node in the interval. For communication descriptor d and $m = \textsc{Succs}^E(h)$, the term $(\mathit{Transp}(m,d) \ \cap \ \mathrm{SAFE}(m,d))$ indicates whether $\textsc{Send}(d)$ can be hoisted across the first node in the interval and, thus, *out* of the interval. We relax the safety criterion for header nodes by combining this term with the above-mentioned equation for safety.

$$\begin{aligned} \mathrm{SAFE}(h,d) = \mathrm{SAFE}(h,d) \cap [\mathit{Transp}(h,d) \cap \bigcap_{s \in \textsc{Succs}^F(h)} \mathrm{SAFE}(s,d) \\ \cup (\mathit{Transp}(m,d) \ \cap \ \mathrm{SAFE}(m,d))] \end{aligned} \tag{4.2}$$

Since the loop header node itself does not reference any non-local data (Section 4.2), $\mathit{Used}(h,d)$ has been dropped from the equation. Also, recall that if a node in $T(h)$ modifies a subset of d, $\mathit{Transp}(h,d) = \perp$; thus, $\textsc{Send}(d)$ gets blocked *after* the interval if it is modified in the interval. On the other hand, if d is not modified in $T(h)$ then the corresponding $\textsc{Send}$ is hoisted across the interval headed by h. Figure 4.3 shows the value of the SAFE predicate for the communication descriptor x[i] (node 16). Relaxed safety requirement allows hoisting $\textsc{Send}$ x[i] from node 16 to the header node 15. Since the section corresponding to $\textsc{Send}$ x[i] is modified in the interval headed by node 10, $\textsc{Send}$ x[i] gets blocked at node 12. Also, $\textsc{Send}$ x[i] cannot be hoisted to node 3 because one of the forward edge successors (node 10) of node 3 is not safe. Conceptually, we place $\textsc{Send}$s just before non-local references and then in a $\textsc{Backward}$ and $\textsc{Upward}$ pass over the graph, hoist them as early as possible to determine the set of safe nodes.

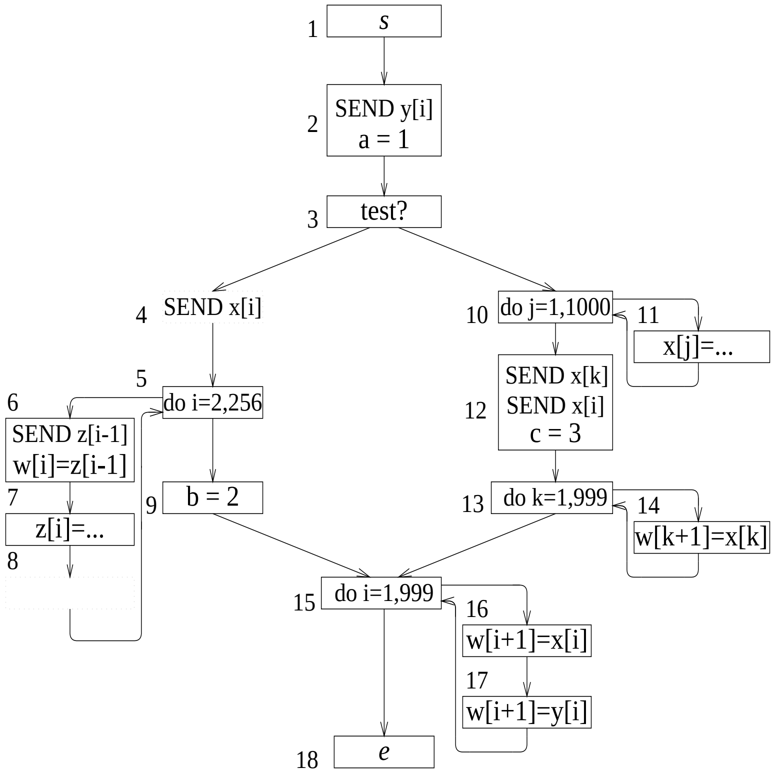


Figure 4.4 Example program with the earliest placement of SEND primitives.

After computing the nodes that satisfy the safety criterion for $\textsc{Send}(d)$, we use a system of data-flow equations to determine the *earliest* safe node for $\textsc{Send}(d)$. A node n is defined to be *earliest* if and only if for every $\textsc{Send}$ placed at the node n, n is the earliest possible location for the send; that is, on every path from the start node s to n, no node prior to n satisfies the safety criterion and communicates the same value as that sent at n.

To find the earliest safe node for each $\textsc{Send}(d)$, we first initialize $\text{EARLIEST}(n, d) = \top$, for all $n \in N$, $d \in \textsc{Comm}$. We then use the following equation to compute the nodes that satisfy the earliest property:

$$\text{EARLIEST}(n, d) = \text{SAFE}(n, d) \ \cap \bigcup_{p \in \textsc{Preds}^F(n)} \neg\text{SAFE}(p, d) \tag{4.3}$$

The above equation recognizes that if all predecessors of a node n are safe locations, then n is not the earliest placement location. For loop header node h, communication set d, and $m = \textsc{Succs}^E(h)$, m is *earliest* if $\textsc{Send}(d)$ is not hoisted out of the loop. In terms of data-flow equation, this is expressed as:

$$\text{EARLIEST}(m, d) = \text{SAFE}(m, d) \ \cap \neg\text{SAFE}(h, d) \tag{4.4}$$

The equations for EARLIEST can be solved in a $\textsc{Forward}$ and $\textsc{Downward}$ traversal of the graph. The solution to the above equation system determines that the nodes 4 and 12 are the earliest safe locations for $\textsc{Send}$ x[i].

Figure 4.4 shows the placement of $\textsc{Send}$s for all non-local references. As explained above, node 4 is the earliest placement location for $\textsc{Send}$ x[i] on the left branch. Since x is modified in the interval headed by node 10, the earliest location for both $\textsc{Send}$ x[k] and $\textsc{Send}$ x[i] along the right branch is node 12. $\textsc{Send}$ y[i] is placed at node 2 because array y is not modified anywhere in the program except at node 1 (by initialization; see Section 4.1.2). Also, $\textsc{Send}$ z[i-1] is blocked by node 5 due to the loop-carried dependence and, thus, is placed at node 6. As mentioned in Section 4.1.2, we use "SEND y[i]", etc. as a convenient notation to represent the communication descriptor for the non-local reference y[i], etc. In reality, index variables are replaced by appropriate ranges when the communication is hoisted out of loops.

4.4 Balanced receive placement

For a given descriptor d, a program execution path can either have no, unbalanced, or balanced instances of SEND(d) and RECV(d) primitives. To represent these three states, we extend the semi-lattice L with an additional lattice value; $L = \{\top, \text{UNBAL}, \bot\}$. The operators $\cap$ and $\cup$ are extended in the obvious way: for example, $x \cup x = x$, $x \cup \bot = x$, and $x \cup \top = \top$. The negation operator is defined as: $\neg\top = \bot$, $\neg\text{UNBAL} = \bot$, and $\neg\bot = \top$. Also, $x - \top = \bot$ and $x - \bot = x$, $\forall x \in L$. Finally, we define a $\wedge$ operator such that $x \wedge y = \bot$, $x \neq y$, $x, y \in L$, and $x \wedge x = x$, $x \in L$.

In order to maximize latency hiding, RECVs must be placed at the latest possible node before the data are used. Also, since every program execution path must have balanced SENDs and RECVs, both communication primitives need to occur at the same loop nesting level. Figure 4.5 shows two examples of unbalanced placement. From the figure, it can be observed that the placement must satisfy balance for each nesting level. For example, since SEND z[i-1] is nested in the **do** loop with header node 5 (Figure 4.4), the corresponding RECV would also have to be placed inside the loop to ensure balance. In other words, the placement of RECVs at a particular loop level can be determined independent of the placement at other level(s). Thus, in the following equations, balance is provided by balancing all execution paths at the same level.

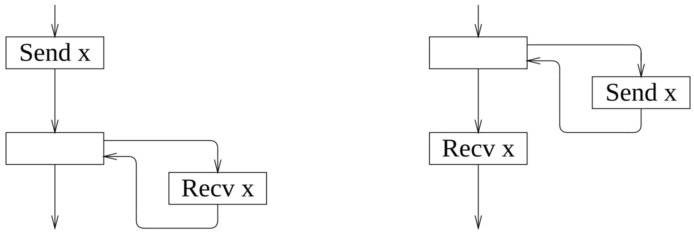


Figure 4.5 Unbalanced SEND/RECV placement.

Let $\text{BALANCED}_{in}(n,d)$ and $\text{BALANCED}_{out}(n,d)$ indicate whether placement of RECV(d) at the start and end of node n will result in balanced placement. As mentioned in Section 4.3, SENDs are placed only at node start. For each node n that contains SEND(d), the data-flow variable $\text{BALANCED}_{in}(n,d)$ is initialized to UNBAL. It is initialized to $\top$ otherwise. For example, BALANCED_{in}(2, y[i]) and BALANCED_{in}(6, z[i-1]) are initialized to UNBAL for the SEND placement shown in Figure 4.4. For receive placement, we need to ensure that: first, RECV(d) is placed before the communicated data are used. Second, if one of the reaching paths at n contains a balanced SEND/RECV pair, including no SEND/RECV, then all paths are balanced before they reach n. These two conditions are expressed by the following two equations, respectively:

$$\text{BALANCED}_{out}(n,d) = \text{BALANCED}_{out}(n,d) \ \cap \ [\text{BALANCED}_{in}(n,d) - \mathit{Used}(n,d)]$$

$$\text{BALANCED}_{in}(n,d) = \text{BALANCED}_{in}(n,d) \ \cap \bigwedge_{p \in \text{PREDS}^F(n)} \text{BALANCED}_{out}(p,d) \tag{4.5}$$

First equation states that if data elements are used at node n, the latest receive location is the start of node n. Figure 4.6 shows how the second equation and the operator $\wedge$ works: if one of the branches (left branch in the figure) reaching a node n has an unbalanced SEND(d) and any other branch has either no or balanced SEND(d)/RECV(d)

pair(s) then $\text{BALANCED}_{in}(n, d)$ gets set to $\perp$. This implies that RECV(d) cannot be placed at n and needs to be placed along the left branch to obtain balanced placement.

For a descriptor $d \in$ COMM, each node n that satisfies either $\text{BALANCED}_{in}(n, d)$ or $\text{BALANCED}_{out}(n, d)$ predicate corresponds to a possible RECV placement location. Let $\text{BALANCED}(n, d) = (\text{BALANCED}_{in}(n, d) \cup \text{BALANCED}_{out}(n, d))$. The latest node for RECV placement can be determined by solving the following equation in FORWARD order:

$$\text{LATEST}(n, d) \;=\; \text{BALANCED}(n, d) \cap \bigcup_{s \in \text{SUCCS}^F(n)} \neg\,\text{BALANCED}_{in}(s, d) \qquad (4.6)$$

The equation for LATEST recognizes that if all the successors of a node n are also valid locations for the RECV placement (that is, $\text{BALANCED}_{in}(s, d) = \text{UNBAL}$, $\forall s \in \text{SUCCS}^F(n)$), then n is not the latest location and, therefore, $\text{LATEST}(n, d)$ gets set to $\perp$. The solution of this equation system determines a unique, latest node where RECV(d) can be placed (as with SEND primitives, RECVs are placed only at the node start).

As shown in Figure 4.7, the solution to the above-mentioned equations gives the following placement: RECV z[i-1] is placed at the start of node 6 (but after SEND z[i-1]) while RECV y[i] is placed at node 15. Section 7.4 describes how we combine

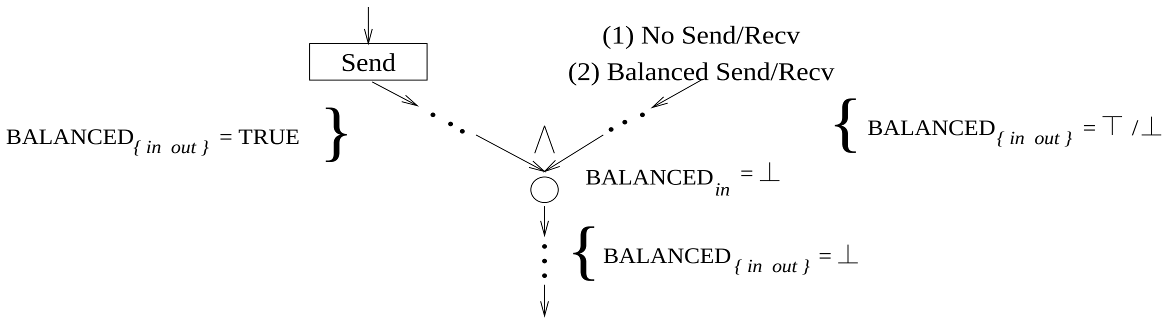


Figure 4.6 Evaluation of $\text{BALANCED}_{\{in\ out\}}$ using the $\wedge$ operator.

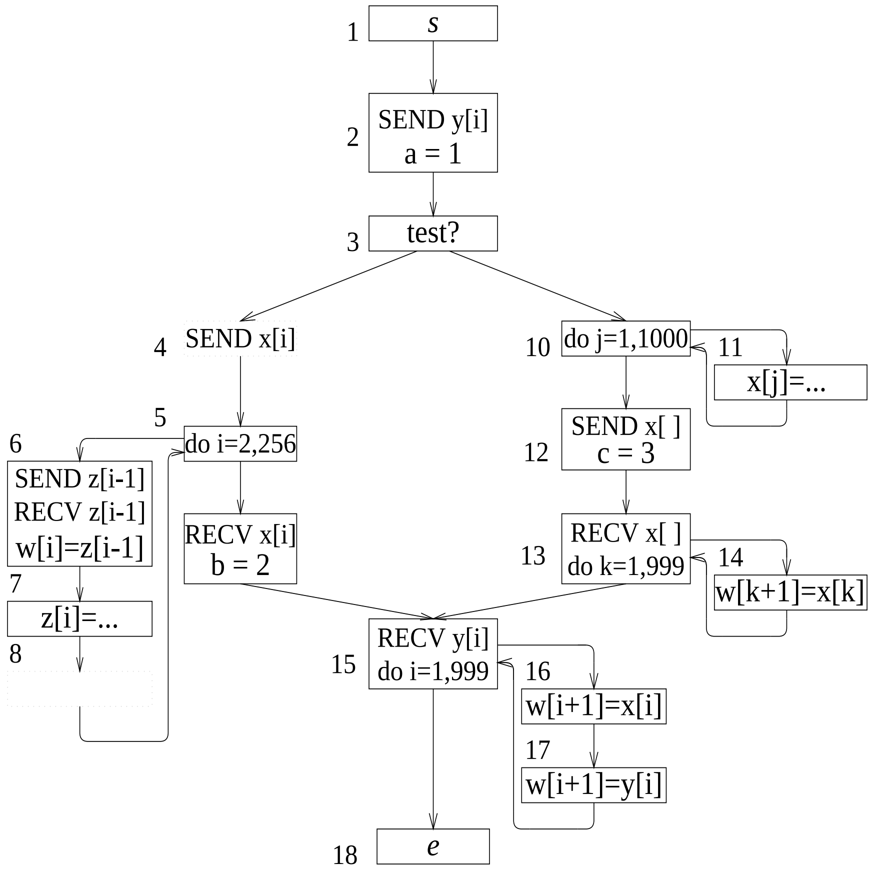


Figure 4.7 Example program with the earliest send placement and the latest balanced receive placement.

the two SENDs to obtain SEND x[ ] at node 12 in Figure 4.4. RECV x[i] is inserted at node 9 and RECV x[ ] for the coalesced message is placed at node 13. Note that RECV x[i] is placed at node 9 because the right branch does not contain SEND or RECV x[i] and contains a balanced placement of SEND and RECV of x[ ].

4.5 Complexity of the data-flow algorithm

As described in Section 4.2, the data-flow equations for the placement of sends as well as receives are solved in the respective second subphases. To solve the data-flow equations, the data-flow algorithm visits every program node using a suitable graph traversal order and solves appropriate equations. Second subphase for both send and receive placement requires two passes each; each of the four passes perform unidirectional analysis. It can be seen that the data-flow equations for the earliest send placement (Section 4.3) as well as the latest, balanced receive placement (Section 4.4) inspect only a subset of incoming/outgoing edges. Therefore, the complexity of our algorithm is $O(E)$. The insertion of synthetic edges to eliminate critical edges increases the size of G and, in the worst case, makes $O(N) = O(E)$. However, it has been noted by several researchers that for typical programs, the average out-degree of graph nodes and the maximal loop nesting depth can be assumed to be bounded by a small constant independent of the size of the program [76, 44]. Therefore, for well structured programs, elimination of critical edges should not increase the size of G significantly. Under the assumption that $O(E)$ is the same as the order of the program size, our algorithm has linear time complexity.

4.6 Correctness of communication placement

An important issue in communication placement analysis is to guarantee that no deadlock is introduced by the compiler and that the communication placement does not exceed the machine-dependent resource constraints. Previous research efforts did not take resource constraints into account and defined correctness under the assumption that sufficient resources were available [72, 43].

Chapter 5 presents the analysis required to ensure that the communication placement does not exceed the resource constraints. Assuming that the resource constraints are satisfied by the placement, correctness requires that the program executes matching number of SENDs and RECVs. Figure 4.1 presented an example of an incorrect placement; it illustrates the importance of *balance* placement criterion.

This section presents correctness proofs for *before* (non-local read) problem. Also, we present proofs for non-header nodes only. The correctness of data-flow equations for header nodes as well as for *after* problem can be proved similarly. Note that the placement analysis is performed on the interval-flow graph of the original, sequential program. Therefore, the correctness proofs presented in this section demonstrate the correctness of communication placement for the original, sequential program. However, the code generation phase performs loop bounds reduction to partition loops among processors. While partitioning computation, the code generation phase can also introduce guards to ensure that only appropriate processors execute each statement in the program. Since the code generation phase modifies the control flow of the program, in general it is possible that though SENDs and RECVs are balanced for the sequential program, they are no longer balanced after the code generation phase. However, it can be shown that the Fortran D95 compiler guarantees the correctness of SEND and RECV placement after code generation phase as well. Section 4.7 presents the related correctness proof.

It is worth stressing that before communication optimizations are performed, there is an one-to-one correspondence between the non-local references and communication descriptors d. Since optimizations are performed *after* computing the earliest placement of SEND primitives, this property holds for safe and earliest equations.

> **Theorem 1** SAFE$(m,d) = \top$ if and only if $\exists\ P\ : m \overset{\mathrm{F}}{\rightsquigarrow} n$ such that $Used(n,d) = \top$ and $\forall q \in P$, $Transp(q,d) = \top$. In other words, node m is a safe placement location if and only if there exists a path of forward edges from m to n such that the data corresponding to d is used at n and is not modified before n.

Proof: From Equation 4.1, SAFE$(m,d) = \top$, if either $Used(m,d) = \top$ or $Transp(m,d) = \top$ *and* all successors of m are SAFE with respect to descriptor d.

By induction over the length of path, it can be shown that every path from m contains a use of d and no node from m modifies d before the statement containing non-local reference corresponding to d.

$\square$

> **Lemma 2** If SAFE(m,d) and SAFE(n,d) are $\top$ and $\exists\ P : m \overset{\mathrm{F}}{\rightsquigarrow} n$, then $\forall q \in P$, SAFE$(q,d) = \top$. In other words, if m and n are safe nodes for descriptor d and there exists a path of forward edges from m to n, then all nodes along the path are safe with respect to the descriptor d.

Proof: We prove the lemma by contradiction. Let's assume that $\exists q \in P$ such that SAFE$(q,d) = \perp$. Also, without any loss of generality, let q be the latest such node along the path P.

SAFE$(q,d) = \perp \Rightarrow Used(q,d) = \perp$ because SAFE is initialized to $\perp$ only for header nodes.

Now, SAFE$(q,d) = \perp$ either because (a) q defines d (that is, $Transp(q,d) = \perp$) or (b) there exists a path from q that does not use d.

By Theorem 1, (a) cannot be true.

If (b) is true, then it implies that no ancestor of q is safe.

$\Rightarrow$ m is not safe; contradiction!

Thus, if m and n are safe nodes for the communication descriptor d and there exists a path of forward edges from m to n, then all nodes along the path are safe with respect to the descriptor d.

$\square$

Corollary 3 If SAFE(m, d) and SAFE(n, d) are $\top$ and $\exists\ P :\ m \overset{\mathrm{F}}{\rightsquigarrow} n$, then $\forall q\ \in\ P$, $\mathit{Used}(q, d) = \bot$. In other words, if m and n are safe nodes for descriptor d and there exists a path of forward edges from m to n, then no node along the path uses data corresponding to d.

Proof: By contradiction. Let's assume $\exists q'$ such that $\mathit{Used}(q', d) = \top$. Since SAFE$(n, d) = \top$, by Theorem 1, there exists a path consisting only of forward edges from n to q'.

However, since $q'\ \in\ P$, we know that $\exists\ q' \overset{\mathrm{F}}{\rightsquigarrow} n$. Thus, there exists a cycle of forward edges! By construction (Section 4.1.1), this can not happen.

Thus, no node along the path of forward edges can have a use of d.

$\square$

Lemma 4 For a descriptor $d \in$ COMM and a node n, if n has multiple forward edge predecessors then EARLIEST$(n, d) = \bot$. In other words, n will not be the earliest node for any communication descriptor d.

Proof: Since n has multiple predecessors and there are no critical edges in G, the critical edge elimination process would have inserted synthetic nodes along every incoming edge to node n.

$\Rightarrow$ Every predecessor of n is a synthetic node.

$\Rightarrow$ By construction, for every predecessor p of node n, (a) $Transp(p,d) = \top$ and $Used(p,d) = \perp$ and (b) p has n as its only successor.

$\Rightarrow$ By Equation 4.1, if $\mathrm{SAFE}(n,d) = \top$ then $\mathrm{SAFE}(p,d)$ will also be $\top$.

$\Rightarrow$ By Equation 4.3, $\mathrm{EARLIEST}(n,d) = \perp$.

$\square$

Theorem 5 If $\mathrm{EARLIEST}(n,d) = \top$, then $\textsc{Send}(d)$ can not be placed at an earlier node.

Proof: The equation for earliest placement criterion:

$$\mathrm{EARLIEST}(n,d) = \mathrm{SAFE}(n,d) \cap \bigcup_{p \in \textsc{Preds}^F(n)} \neg\mathrm{SAFE}(p,d)$$

can also be written as:

$$\mathrm{EARLIEST}(n,d) = \mathrm{SAFE}(n,d) \cap \neg[\bigcap_{p \in \textsc{Preds}^F(n)} \mathrm{SAFE}(p,d)]$$

In other words, node n is marked earliest if *at least* one of n's predecessors is not safe. Also, note that

$$\mathrm{EARLIEST}(n,d) = \top \;\Rightarrow\; \mathrm{SAFE}(n,d) = \top$$

Let's assume that some ancestor m of n is also a safe location (that is, $\mathrm{SAFE}(m,d) = \top$), but n is marked as the earliest location.

As mentioned above, $\mathrm{EARLIEST}(n,d) = \top$ implies that at least one of n's predecessors is *not* safe. Also, by Lemma 4, n has only one predecessor, say q. Thus, $\mathrm{SAFE}(q,d) = \perp$; in other words, q is not safe.

By Lemma 2, all nodes along $m \overset{F}{\leadsto} n$ must be safe. But, since q dominates n, $q \in m \overset{F}{\leadsto} n$ and SAFE$(q, d) = \bot$. Contradiction!

Thus, n is the earliest safe node for SEND(d).

□

Theorem 6 Along every forward edge path that has a use of d, $\exists$ unique n such that EARLIEST$(n, d) = \top$.

Proof: The above theorem can be divided into two subparts:

1. There exists at least one earliest node.
2. There exists at most one earliest node.

The first part is trivial: since the node for which $Used(q, d) = \top$ is always the safe node for SEND(d) placement, every path that contains the use of d also contains at least one node such that SAFE$(q, d) = \top$. Thus, every path will contain at least one earliest node.

We prove the second part by contradiction. Let's assume that there exist two nodes, say m and n along a forward edge path such that EARLIEST$(m, d) = \top$ and EARLIEST$(n, d) = \top$.

EARLIEST$(m, d) = \top \Rightarrow$ SAFE$(m, d) = \top$ and EARLIEST$(n, d) = \top \Rightarrow$ SAFE$(n, d) = \top$.

The remaining proof is similar to the proof of Theorem 5.

□

Theorem 7 BALANCED$_{in/out}(n, d) =$ UNBAL if and only if there exists a path, $P : m \overset{F}{\leadsto} n$ such that SEND(d) is placed at node m and $\forall q \in P$, $Used(q, d) = \bot$. In other words, a node n is a balanced node for d if and only if there exists a preceding $\textsc{Send}(d)$ and the use of d has not been seen.

Proof: We first prove the *if* part and then the *only if* part.

If:

In the absence of jumps, $\forall n \in N$, $\text{BALANCED}_{in}(n, d)$ is initialized to either UNBAL or $\top$.

Since $\textsc{Send}(d)$ is placed at node m, $\text{BALANCED}_{in}(m, d)$ is initialized to UNBAL. Moreover, m must be the earliest safe node for d; in other words, predecessors of m won't have $\textsc{Send}(d)$ attached to them.

$\Rightarrow \forall p \in \textsc{Preds}^F(m)$, $\text{BALANCED}_{out}(p, d) = \top$.

$$\Rightarrow \bigwedge_{p \in \textsc{Preds}^F(m)} \text{BALANCED}_{out}(p, d) = \top.$$

By Equation 4.5, this implies that $\text{BALANCED}_{in}(m, d) = \text{UNBAL}$. If $Used(m, d) = \perp$, $\text{BALANCED}_{out}(m, d) = \text{UNBAL}$.

Now, by induction over the length of path from m to n, it can be shown that if there exists a preceding $\textsc{Send}(d)$ and no use of d along the path, $\text{BALANCED}_{in/out}(n, d)$ gets set to UNBAL.

Only if:

This part is proved by contradiction. Let $\text{BALANCED}_{in}(n, d) = \text{UNBAL}$ and $\exists q' \in P$ such that $Used(q', d) = \top$. Further, without any loss of generality, let's assume that q' is first such node on the path P.

By induction, $\text{BALANCED}_{in}(q', d) = \text{UNBAL}$ or $\top$.

Since $\text{BALANCED}_{out}(q', d) = \text{BALANCED}_{out}(q', d) \cap [\text{BALANCED}_{in}(q', d) - Used(q', d)]$,

$\Rightarrow \text{BALANCED}_{out}(q', d) = \perp$.

$\Rightarrow$ BALANCED$_{in}(q'', d) = \perp$, for each forward edge successor of q'.

Thus, by induction, BALANCED$_{in}(n, d) = \perp$. Contradiction!

$\square$

Based on the proof given above, we can also prove the following corollary.

Corollary 8 If BALANCED(m, d) = UNBAL and BALANCED(n, d) = UNBAL and $\exists$ a path $m \overset{F}{\leadsto} n$, then $\forall q \in m \overset{F}{\leadsto} n$, BALANCED$(q, d)$ = UNBAL. In other words, if m and n are balanced nodes and there exists a path consisting of forward edges from m to n, then all nodes along the path are balanced.

Lemma 9 If a node n with multiple successors is the latest node (that is, LATEST(n, d) = UNBAL), then the data is used at the node (that is, $Used(n, d) = \top$).

Proof: The proof for this lemma is similar to the proof for Lemma 4. Recall that due to critical edge elimination, each successor of node n with multiple successors has a unique predecessor – the node n.

Equation 4.6 states that a node is marked as the latest node if at least one of the successors of the node is not a balanced location.

Since all successors of n have n as their predecessor, by equation for $BALANCED_{in}(s, d)$, BALANCED$_{out}(n, d)$ must be $\perp$.

But, since LATEST(n, d) = UNBAL, BALANCED$_{in}(n, d)$ must be UNBAL.

Now, BALANCED$_{in}(n, d)$ = UNBAL and BALANCED$_{out}(n, d) = \perp$.

$\Rightarrow$ $Used(n, d) = \top$.

Thus, if a node n with multiple successors is the latest node, then the data is used at the node.

□

Theorem 10 For every $\textsc{Send}(d)$, there is exactly one matching $\textsc{Recv}(d)$ along every path.

Proof: The theorem can be divided into two subparts:

1. There exists at least one $\textsc{Recv}(d)$.

2. There exists at most one $\textsc{Recv}(d)$.

Since $\textsc{Recv}(d)$ can always be placed at the node that contains $\textsc{Send}(d)$, at least one node will have $\text{BALANCED}_{in}(n, d) = \text{UNBAL}$ Thus, even if one of the successors of node n is not balanced, node n is a valid candidate for the latest placement of $\textsc{Recv}(d)$. Thus, for every $\textsc{Send}(d)$ there will be at least one $\textsc{Recv}(d)$.

To prove the second part, let's assume that, along some path consisting of only forward edges, there exist two nodes with matching $\textsc{Recv}(d)$. Let m and n be the two nodes and P be the path $m \overset{\text{F}}{\rightsquigarrow} n$. Since both m and n have $\textsc{Recv}(d)$, both $\text{LATEST}(m, d)$ and $\text{LATEST}(n, d)$ must be UNBAL.

Now, $\text{LATEST}(m, d) = \text{UNBAL} \Rightarrow \text{BALANCED}_{in}(m, d) = \text{UNBAL}$ and $\text{LATEST}(n, d) = \text{UNBAL} \Rightarrow \text{BALANCED}_{in}(n, d) = \text{UNBAL}$.

By Corollary 8, $\forall q \in P$, $\text{BALANCED}_{in/out}(q, d) = \text{UNBAL}$.

Let's consider node m. By Lemma 9, m can not have multiple successors: otherwise $Used(m, d) = \top$.

$\Rightarrow \text{BALANCED}_{out}(m, d) = \bot$

$\Rightarrow \forall s \in \textsc{Succs}^F(n)\ \text{BALANCED}_{in}(s, d) = \bot$. Together with Lemma 8, this leads to contradiction.

Thus, m can have only one successor, say s_0.

Now, since m is the latest node, $\text{BALANCED}_{in}(s_0, d)$ must be $\perp$. Once again, this leads to contradiction! Thus, m can't be the latest node!

In other words, there can be at most a single matching RECV(d) for a SEND(d) along each path.

□

4.7 Correctness of communication placement in partitioned (parallel) program

Theorem 10 proved that every program execution path in the original program contains balanced SENDs and RECVs. As mentioned before, however, the code generation phase of the compiler alters the control flow in the program by partitioning the original iteration space and assigning a subset of iterations to individual processors. Thus, after computation is partitioned among processors, processors can be viewed as following different paths through the program. The compiler needs to guarantee the correctness of communication placement after code generation as well.

Correctness is relatively easy to guarantee in the absence of conditionals. However, ensuring that both the sending and receiving processors issue matching communication operations in the presence of arbitrary control-flow is a non-trivial problem. As pointed out by von Hanxleden, correctness requires that "if a processor p reaches a reference requiring communication involving some other processor q, then q also has to reach the reference" [43]. Figure 4.8 presents an example program that can lead to incorrect communication placement. If the "**do** i" loop is partitioned such that processor 0 executes first 10 iterations and processor 1 executes last 10 iterations then the "**if**" *conditional* evaluates to false on processor 0. In this case, though processor 1 executes RECV a(1), processor 0 never executes SEND a(1). We now describe how the Fortran D95 compiler ensures the correctness of communication placement.

It is worth stressing that the Fortran D compiler did not provide correctness. However, for a restricted class of programs, correctness of communication placement can be guaranteed by the communication placement framework itself. The GIVE-N-TAKE framework provided correct placement for the programs with following communication characteristics [43]:

Definition 11 A program is a *valid instance of communication placement problem* if its naïve solution results in a correct problem; that is, the program annotated with SEND/RECV pair immediately preceding each non-local reference is deadlock-free.

The Fortran D compiler did not guarantee correctness for *invalid* instances of communication placement framework. It can be seen that the program shown in Figure 4.8 has an invalid instance of communication placement problem and, therefore, as described above *can* lead to incorrect communication placement.

The correctness requirement characterizes the class of programs for which the correctness of communication placement can be guaranteed by the communication

```
do i = 1, 20                    do i = 1, 20
  if (i .gt. 10) then             if (i .gt. 10) then
    b(i) = a(1)                     if (Owner(a(1)))  SEND a(1)
  endif                             RECV a(1)
enddo                               b(i) = a(1)
                                  endif
                                enddo
```

Figure 4.8 An example program that can lead to potentially incorrect communication placement. If array a is distributed such that processor 0 owns the first 10 elements, then RECV a(1) will not have a matching SEND a(1).

	Computation partitioning
do i = 1, 20	ON_HOME(a(1)) ∪ ON_HOME(b(1:20))
if (i .gt. 10) **then**	ON_HOME(a(1)) ∪ ON_HOME(b(i))
if (*Owner*(a(1))) SEND a(1)	ON_HOME(a(1))
RECV a(1)	ON_HOME(b(i))
b(i) = a(1)	ON_HOME(b(i))
endif	
enddo	

Figure 4.9 Illustration of how incorporating computation partitioning corresponding to communication statements ensures that the communication primitives are balanced and that the placement is correct.

placement framework. The placement framework presented in this chapter also guarantees correct placement if the above-mentioned restriction is imposed.

We now show how the Fortran D95 compiler preserves the correctness of placement during the code generation phase. The correctness of placement is a result of interaction between balanced "sequential" communication placement, propagation of computation partitioning for the communication statements to the enclosing scopes, and the loop bounds reduction based on the computation partitioning of the statements in the loop.

Figure 4.9 shows the code fragment with each statement annotated with its computation partitioning. As described in Section 3.1, the computation partitionings corresponding to SEND a(1) and RECV a(1) are ON_HOME(a(1)) and ON_HOME(b(i)). In the *choose computation partitioning* phase, the computation partitionings of communication statements are also propagated to the enclosing statements. Thus, the computation partitioning of the enclosing "**if**" is ON_HOME(a(1)) ∪ ON_HOME(b(i)) and, as shown in Figure 4.9, this computation partitioning is propagated to the "**do**" loop. The code generation phase utilizes this computation partitioning while partitioning the loop among processors. The computation partitioning assigned to the loop ensures that every iteration is executed on the processor that owns a(1). If, for a particular iteration of the loop, a processor executes Recv a(1), then the "**if**" *conditional* must have evaluated to true. This implies that the *conditional* will evaluate to true for the processor that owns a(1) also and, therefore, a matching Send a(1) will be executed.

A formal proof to prove the correctness of communication placement can be constructed based on the above argument. In addition, it can be proved that the Fortran D95 compiler provides correctness even with *relaxed* safety criterion. The relaxed safety criterion allows Sends to be hoisted out of conditionals even if the communicated data is not used along all paths. Note that since the Recvs are placed at the latest program location, they might be placed *inside* the conditional. In other words, relaxed safety criterion can cause the matching Sends and Recvs to be placed under different conditionals. However, since our placement framework provides balance, both the Sends and Recvs *will* be placed in the same loop nest. Therefore, the computation partitionings for both Sends and Recvs will be propagated to the common enclosing loop. Thus, as described above, the code generation phase will ensure that the loop partitioning ensures that the correctness is preserved.

4.8 Results

We now demonstrate the benefits of latency hiding with the help of detailed analysis of two benchmark programs: Erlebacher and Disper.

Erlebacher is a 800 line 3D tri-diagonal solver that uses Alternating-Direction-Implicit (ADI) integration. The second dimension of the 3D data arrays in the program was distributed using BLOCK distribution. Erlebacher is not fully data-parallel because it contains computational wavefronts that sequentialize parts of the computation. For example, as shown in Figure 4.10, the tridvpj subroutine contains a loop-carried dependence in the distributed dimension. Thus, ERLEBACHER presents opportunity for VMP. However, it does not contain conditionals and dependence-based placement techniques can also successfully perform the VMP optimization. Since the benefits of the VMP for ERLEBACHER have been studied before [48], we restrict our analysis to the performance gains due to latency hiding achieved by non-atomic placement of SENDs and RECVs.

We translated ERLEBACHER into SPMD node code with the communication placement that achieves latency hiding. The dy3d6p subroutine with the earliest SEND placement and the latest RECV placement is shown in Figure 4.11. We also hand-instrumented the node code to correspond to output program *without* latency hiding optimization. Note that both versions had vector message pipelining. The node code was then compiled using Intel's *if77* cross-compiler (Release 4.0) with -O4 option. Table 4.1 shows the average execution times over 10 runs (measured using the *dclock()* timer). The program was run on Intel iPSC/860 hypercube with various input problem sizes and number of processors. ERLEBACHER with 96 x 96 x 96 data elements was too small for 32 processors.

Table 4.1 shows that non-atomic placement of SENDs and RECVs yields up to 12% improvement in the total execution time. ERLEBACHER has $\Theta(n^2)$ communication and $\Theta(n^3)$ computation. Thus, for a given problem size, as the number of processors is decreased, communication time constitutes a smaller portion of the total execution time and the impact of latency hiding diminishes (except for $n = 160$ and 16 processors, for which we get 12% improvement).

DISPER is a 1500 line stencil computation from the UTCOMP reservoir simulator [71]. Unlike ERLEBACHER, DISPER is highly data-parallel program. Figure 4.12 presents a fragment of code from DISPER. As shown in the figure, it has a lot of conditionals nested inside loop nests. Thus, it requires data-flow analysis to ensure

```
subroutine dy3d6p
parameter (n$p = 8)
double precision uu(n,n,n), uud(n,n,n)
distribute (*,block,*) :: uu, uud
do k = 1, n
  do i = 1, n
    uud(i,1,k) = F(uu(i,2,k), uu(i,3,k), uu(i,n-1,k), uu(i,n,k))
    uud(i,2,k) = F(uu(i,1,k), uu(i,3,k), uu(i,4,k), uu(i,n,k))
  enddo
enddo

do k = 1, n
  do i = 1, n
    uud(i,n-1,k) = F(uu(i,1,k), uu(i,n-3,k), uu(i,n-2,k), uu(i,n,k))
    uud(i,n,k) = F(uu(i,1,k), uu(i,2,k), uu(i,n-2,k), uu(i,n-1,k))
  enddo
enddo

do k = 1, n
  do j = 3, n-2
    do i = 1, n
      uud(i,j,k) = F(uu(i,j-2,k), uu(i,j-1,k), uu(i,j+1,k), uu(i,j+2,k))
    enddo
enddo
enddo
end
```

Figure 4.10 The dy3d6p subroutine from the Erlebacher benchmark.

```
subroutine dy3d6p
parameter (n$p = 8,block_size = n/n$p)
double precision uu(n,-1:block_size+2,n), uud(n,block_size,n)
block_start = my$p*block_size
SEND uu(1:n,n-1:n,1:n) to P_0 ①
SEND uu(1:n,1:2,1:n) to P_{n$p-1} ②
SEND uu(1:n,block_size-1:block_size,1:n) to P_{my$p+1} ③
SEND uu(1:n,block_start:block_start+1,1:n) to P_{my$p-1} ④
RECV uu(1:n,n-1:n,1:n) ①
do k = 1, n
  do i = 1, n
    uud(i,1,k) = F(uu(i,n-1,k), uu(i,n,k), ...)
    uud(i,2,k) = F(uu(i,n,k), ...)
  enddo
enddo
RECV uu(1:n,1:2,1:n) ②
do k = 1, n
  do i = 1, n
    uud(i,n-1,k) = F(uu(i,1,k), ...)
    uud(i,n,k) = F(uu(i,1,k), uu(i,2,k), ...)
  enddo
enddo
RECV uu(1:n,block_size-1:block_size,1:n) ③
RECV uu(1:n,block_start:block_start+1,1:n) ④
do k = 1, n
  do j = 3, n-2
    do i = 1, n
      uud(i,j,k) = F(uu(i,j-2,k), uu(i,j-1,k), uu(i,j+1,k), uu(i,j+2,k))
    enddo
  enddo
enddo
end
```

Figure 4.11 The dy3d6p subroutine from the Erlebacher benchmark with the earliest send placement and the latest receive placement.

Size	# Proc.	milliseconds No overlap	Overlap	% Diff.
160 x 160 x 160	32	138.82	130.46	6.02
	16	272.22	239.29	12.1
	8	365.04	344.89	5.52
128 x 128 x 128	32	75.704	69.768	7.84
	16	143.14	135.94	5.03
	8	214.83	211.04	3.78
96 x 96 x 96	32	-	-	-
	16	40.174	35.186	12.4
	8	58.860	57.432	2.42

Table 4.1 Experimental results to illustrate the benefits of latency hiding for ERLEBACHER. Timings are in milliseconds.

that every program execution path contains balanced communication placement. In order to derive any benefits, communication needs to be moved out of the conditionals – possibly resulting in over-communication; but this allows communication to be hoisted out of loops [71]. In other words, DISPER requires the safety criterion presented in Section 4.3 to be relaxed. However, since the safety has an independent data-flow equation, the equation can be changed easily.

DISPER code expressed in data-parallel style contains a lot of four-dimensional arrays that are all distributed along the second physical dimension with BLOCK distribution. Since DISPER is a regular stencil computation, it has a low communication to computation ratio. Thus, although latency hiding can reduce communication overhead by up to 28%, the decrease in total execution time – averaged over 10 runs – is at most 4.6% (Table 4.2). As explained above, the benefits of communication overhead reduction diminish as the number of processors are decreased; this is reflected in the results presented in Table 4.2.

The results presented in this section are representative of the performance gains we expect due to latency hiding. The benchmark programs presented in this sec-

```
subroutine disper
logical lsat(256,8,8,4)
double precision ddx(256,8,8), ddy(256,8,8), ddz(256,8,8)
double precision pmfr(256,8,8,4,5), gradx(256), grady(256), gradz(256)
template t(256)
align ddx(i,j,k), ddy(i,j,k), ddz(i,j,k) with t(i)
align lsat(i,j,k,l), pmfr(i,j,k,l,m) with t(i)
align gradx(i), grady(i), gradz(i) with t(i)
distribute t(block)
do j = 2, 4
  do i3 = 1, 8
    do i2 = 1, 8
      do i1 = 1, 256
        ...
        if ((i1 .ne. 1) .and. (i1 .ne. 256)) then
          if (lsat(i1-1,i2,i3,j) .and. lsat(i1+1,i2,i3,j)) then
            grady(i1) = (pmfr(i1+1,i2,i3,j,k) - pmfr(i1-1,i2,i3,j,k)) /
                (0.5 * (ddy(i1+1,i2,i3) + ddy(i1-1,i2,i3)) + ddy(i1,i2,i3))
            ...
          endif
        endif
        ...
      enddo
    enddo
  enddo
enddo
end
```

Figure 4.12 Code fragment from the Disper subroutine of the UTCOMP oil reservoir simulation program.

tion require communication placement that hides latency and, thus, demonstrate the usefulness of our framework.

4.9 Related work

Code motion and placement to eliminate partially redundant computations was introduced by the PRE algorithm [76]. The Lazy Code Motion (LCM) technique [67] and the GIVE-N-TAKE framework [44] improved partial redundancy elimination optimization by excluding unnecessary code motion and by providing non-atomic placement regions, respectively. Our framework is based on both these frameworks. A simple extension of either the LCM technique or the GIVE-N-TAKE framework is not possible because of the atomic placement dependent definition of unnecessary code motion used by the LCM technique and the complexity of the GIVE-N-TAKE framework.

The analysis for SEND placement is based on the LCM technique developed by Knoop, Rüthing, and Steffen. We extend their data-flow equations in two ways. First, we incorporate data-dependence information and second, we allow communication to be optimistically hoisted out of loops. Moreover, unlike LCM, which uses iterative analysis, our framework is based on interval analysis.

In terms of functionalities provided, our framework most closely resembles the GIVE-N-TAKE framework developed by Hanxleden and Kennedy. Both determine non-atomic and balanced communication placement. However, the two frameworks take radically different approach in providing these functionalities. Our framework utilizes the dependence information to develop original and *significantly* simpler data-flow equations. Unlike the GIVE-N-TAKE framework, we determine SEND and RECV placement in independent phases using simple uni-directional analyses. As explained in Chapter 6, this allows us to perform message coalescing and redundant communication elimination optimizations easily. SENDs are coalesced before determining

Size	# Proc.	milliseconds No overlap	Overlap	% Diff.
128 x 8 x 8 x 4	32	0.597	0.583	2.34
	8	2.068	2.061	0.34
256 x 8 x 8 x 4	32	1.114	1.062	4.63
	8	4.271	4.176	2.21

Table 4.2 Execution times for DISPER to demonstrate the benefits of latency hiding. Timings are in milliseconds.

the corresponding RECV placement; thus, SENDs can be coalesced without worrying about coalescing the matching RECVs. If placement for both SEND and RECV primitives is determined in the same phase, as is done in the GIVE-N-TAKE framework, ensuring that the coalesced messages are balanced becomes a non-trivial problem. Moreover, and again, as explained in Chapter 6, independent SEND and RECV phases allow us to support VMP. Finally, Chapter 5 describes how the compositional structure of our analysis allows us to incorporate machine-dependent resource constraints such as cache size and buffer size into the placement analysis [64].

The problem of data-flow based communication placement has been addressed by several researchers. Amarasinghe and Lam use a *last write tree* to optimize communication placement and support VMP *within* a single loop nest [4]. Moreover, they don't allow loops within conditionals. Granston and Veidenbaum combine PRE and dependence analysis to eliminate redundant monolithic global-memory accesses across loop nests in the presence of conditionals [37]. Gong, Gupta, and Melhem present a data-flow framework to separate sends and receives by placing sends at "the earliest point at which the communication can be performed" [36]. However, their technique does not eliminate partially redundant communication, handles only singly-nested loops and one-dimensional arrays, and does not provide balanced communication placement.

Gupta, Schonberg, and Srinivasan adapt PRE for communication placement and to develop a unified communication optimization framework [40]. Their framework does not perform any analysis to ensure balanced placement of SEND and (blocking) RECV primitives. To ensure correctness, it initiates non-blocking receives immediately after the sends and uses a wait primitive before *every* non-local reference to block the statement from executing till the data are received. The repeated calls to waits can introduce unnecessary overheads.

4.10 Summary

The framework presented in this chapter determines communication placement with unified dependence and data-flow analysis. The framework determines balanced, non-atomic placement of SENDs and RECVs to maximize latency hiding. In addition, we showed that our placement algorithm has linear time complexity. Finally, we presented results to demonstrate that the latency hiding achieved by non-atomic placement of SENDs and RECVs reduces communication overhead.

The framework presented in the chapter uses two uni-directional passes to determine SEND and RECV placement. As will be described in Chapter 6, two independent phases simplify the implementation of message coalescing optimization. Also, though the chapter describes the placement of non-local read communication primitives only, we have implemented a similar analysis for the placement of communication primitives for non-local writes also in the Fortran D95 compiler.

The framework presented in the chapter is applicable to a broad class of code placement problems, including the classical domains of PRE techniques as well as memory hierarchy related problems. The framework can be easily adapted to other code placement problems – both with and without the balancedness requirement. Two independent phases for SEND and RECV placement facilitate using only one of the two phases, if required. For example, the placement of prefetch instructions for distributed shared-memory architectures requires only the placement corresponding to the earliest SEND placement. Finally, since each placement criterion has a corresponding data-flow equation, a particular placement criterion can be changed by simply modifying the corresponding data-flow equation while leaving other equations unchanged. For example, in order to allow hoisting communication out of frequently executed conditional paths, only the equation for the safety criterion needs to be modified.

Chapter 5

Resource-based Communication Placement

The communication placement framework presented in Chapter 4 ignored machine-dependent resource constraints. This chapter describes how resource cosntraints are incorporated in the placement analysis. Examples of resource constraints include cache size in DSMs, in-core memory size in out-of-core computations, and the number of physical registers in register allocation. Code placement that does not take resource constraints into account can prove too eager in moving code fragments earlier in the program. For example, prefetching data at the earliest program location can cause data to get displaced from cache even before it is used. Moreover, placing communication at the earliest location can increase demands on the system resources. An example of this is the increased buffer requirement due to unconstrained hoisting of messages out of loops. Resource constraints can affect both the performance and correctness of the placement. Prefetch placement that honors cache size resource constraint will yield better performance than the placement that doesn't.

On the other hand, and more significantly, resource-based communication placement is required to ensure the correctness of the placement. For example, the buffer requirement for resource-independent communication placement can exceed the maximum available buffer and, thus, make it impossible to execute the program; resource-based communication placement can guarantee that the memory requirement of the program does not exceed the maximum available in-core memory.

As described in Section 4.9, there have been several efforts to use data-flow analysis to determine communication placement that minimizes communication overhead.

However, to reduce the complexity of the problem being solved, previous frameworks ignored most machine-dependent resource constraints. This chapter describes *resource-based communication placement*. It extends the data-flow analysis framework presented in Chapter 4 to incorporate machine-dependent resource constraints. We demonstrate our approach by imposing buffer size constraint on communication placement.

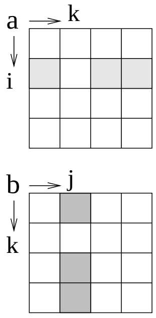
The rest of this chapter is organized as follows. Section 5.1 illustrates the need for resource-based communication placement with the help of benchmark programs. Section 5.2 presents an overview of resource-based communication placement analysis. Section 5.3 shows how the problem of incorporating resource constraints into communication placement can be reduced to the problem of proper initialization of appropriate data-flow variables. Section 5.4 details the analysis to compute the buffer requirement for a given program. Section 5.5 proves that the problem of optimal buffer usage while minimizing communication overhead in NP-complete. It also presents a heuristics for the problem. The heuristics is used in Section 5.6, which describes how the results of resource analysis, together with the stripmining transformation, can be used to improve the efficacy of the communication placement. Section 5.7 gives some implementation details. Section 5.8 summarizes the resource-based placement analysis technique.

5.1 Motivating examples

Resource constraints pose a fundamental and important issue that should be handled by every compiler. Consider the matrix multiply program fragment shown in Figure 5.1(a). Let the three data arrays (a, b, and c) be of size $n \times n$ and distributed with (BLOCK, BLOCK) distribution across a 4×4 processor grid. Let the owner of c(i,j) perform the computation; thus, the references a(i,k) and b(k,j) cause non-local

```
do k = 1, n
   do j = 1, n
      do i = 1, n
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      enddo
   enddo
enddo
```

(a)



(b)

```
SEND/RECV a(i$my$lb:i$my$ub, 1:n)
SEND/RECV b(1:n, j$my$lb:j$my$ub)
do k = 1, n
   do j = j$my$lb, j$my$ub
      do i =i$my$lb, i$my$ub
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      enddo
   enddo
enddo
```

(c)

Figure 5.1 (a) Original matrix multiply loop nest.
(b) Non-local data accessed by processor (1,1).
(c) Matrix multiply with partitioned loops.

accesses. As shown in Figure 5.1(c), communication can be placed outside the loop nest. For the sake of clarity, the guards and processor loops enclosing communication primitives are not shown in the figure.

Figure 5.1(b) shows the non-local sections of arrays a and b required by processor (1,1) to perform its local computation; other processors need to fetch similar non-local sections of arrays a and b. In the distributed-memory systems, non-local data is communicated using messages. Now, each message has a corresponding buffer that the compiler utilizes to unpack the data received from other processor (we will refer to these buffers as non-local buffers). As can be seen from Figure 5.1(b), the communication placement shown in Figure 5.1(c) requires non-local buffers of size $3n^2/8$ per processor. On the other hand, the total size of local sections of arrays a, b, and c per processor is $3n^2/16$. Thus, the non-local buffer required by the communication placement in Figure 5.1(c) *triples* the memory requirement! In general, for a $p_1 \times p_2$ processor grid, the total size of local sections of arrays a, b, and c per processor is $3n^2/p_1p_2$ and the size of non-local buffers per processor is $(p_1 - 1)n^2/p_1p_2 + (p_2 - 1)n^2/p_1p_2$. Thus, for the general case, non-local buffers increase the buffer requirement by a factor of $(p_1 + p_2 + 1)/3$! In other words, the increase in buffer requirement due to non-local buffers is proportional to the size of the processor grid. However, since memory is limited, it might not always be possible to have so much memory available. Therefore, the compiler needs to ensure that the non-local buffers do not cause the node codes (corresponding to Figure 5.1(c)) to exceed the maximum memory available.

Note that the non-local buffer requirement would have been less if the communication were not hoisted (vectorized) to the outermost level. For example, if both SEND a(i,k) and SEND b(k,j) (along with RECV a(i,k) and RECV b(k,j)) are placed inside the "**do** i" loop, each processor needs two buffers with only a single element each; thus, the non-local buffer requirement is 2. Also, if the communication occurs inside the "**do** j" and "**do** k" loops, the buffer requirement is $n/4+1$ and $n/2$, respectively. If, however, communication is placed as shown in Figure 5.1(c), each processor requires non-local buffers of size $3n^2/8$.

Let's assume that $n/2$ is less than the maximum memory available for non-local buffers but $3n^2/8$ exceeds the maximum. (For example, on 16 Intel iPSC/860 processors with 8 MB memory available per processor, this occurs for double precision arrays of size 1280×1280. In other words, the resource-independent placement results in an incorrect placement.) To find a placement that requires less buffer than the maximum non-local buffer size (that is, to satisfy the maximum non-local buffer resource constraint), we would like the data-flow analysis to place SEND/RECV a(i,k) and SEND/RECV b(k,j) inside the "**do** k" loop. In other words, we wan the compiler to incorporate resource constraints in the communication placement analysis.

Further, resource constraints and latency hiding optimizations have conflicting requirements. Consider the dy3d6p subroutine, shown in Figure 4.10, from the Erlebacher benchmark program. Figure 5.2 shows the corresponding pseudo-node-code program along with the communication causing references and the resource-

```
subroutine dy3d6p
parameter (n$p = 8,block_size = n/n$p)
double precision uu(n,-1:block_size+2,n), uud(n,block_size,n)
block_start = my$p*block_size
SEND uu(1:n,n-1:n,1:n) to P_0 ①
RECV uu(1:n,n-1:n,1:n) ①
do k = 1, n
  do i = 1, n
    uud(i,1,k) = F(uu(i,n-1,k), uu(i,n,k), ...)
    uud(i,2,k) = F(uu(i,n,k), ...)
  enddo
enddo
SEND uu(1:n,1:2,1:n) to P_{n$p-1} ②
SEND uu(1:n,block_size-1:block_size,1:n) to P_{my$p+1} ③
SEND uu(1:n,block_start:block_start+1,1:n) to P_{my$p-1} ④
RECV uu(1:n,1:2,1:n) ②
do k = 1, n
  do i = 1, n
    uud(i,n-1,k) = F(uu(i,1,k), ...)
    uud(i,n,k) = F(uu(i,1,k), uu(i,2,k), ...)
  enddo
enddo
RECV uu(1:n,block_size-1:block_size,1:n) ③
RECV uu(1:n,block_start:block_start+1,1:n) ④
do k = 1, n
  do j = 3, n-2
    do i = 1, n
      uud(i,j,k) = F(uu(i,j-2,k), uu(i,j-1,k), uu(i,j+1,k), uu(i,j+2,k))
    enddo
  enddo
enddo
end
```

Figure 5.2 The dy3d6p subroutine from the Erlebacher benchmark with resource-based communication placement.

independent communication placement. As before, the references that require off-processor data are determined based on the assumption that the owners of the *lhs* references perform the computation. For the sake of clarity, we have neither reduced/localized loop bounds nor inserted guards around the loops and communication primitives. The SENDs are placed at the beginning of the subroutine since the array uu is not modified in the subroutine. The data is received just before it is required.

The communication placement shown in Figure 5.2 maximizes the latency hiding by initiating all the SENDs at the start of the subroutine. However, this placement increases the ranges over which the messages are "live". Since a message can be received any time after it has been sent, we assume that the corresponding non-local buffer is live from the time data is sent. This buffer remains live till the last use of the data. Thus, the earliest placement of SENDs increases the live ranges of non-local buffers which, in turn, results in more non-local buffers to be live simultaneously and thus increases the total memory required to allocate non-local buffers. On 8 Intel iPSC/860 processors with 8 MB memory available per processor and n = 112, the placement shown in Figure 5.2 causes the memory requirement to exceed the maximum memory available and the program can not be loaded on the cube. Thus, once again, resource-independent placement leads to an incorrect communication placement. This example illustrates the conflicting goals of the latency hiding optimizations and resource constraints: although the placement in Figure 5.2 maximizes the latency hiding, it also increases the buffer requirement of the program.

Our goal is to develop a placement framework that can maximize latency hiding while honoring resource constraints. For dy3d6p, this can be achieved by initiating SENDs ② – ④ after the first loop nest and by using the same memory for the non-local buffers for messages ① and ②. This decreases the memory requirement by 196

KB and the resulting program fits in the memory. Sections 5.2 – 5.6, describe the placement analysis that handles the above-mentioned issues.

5.2 Overview of resource-based communication placement

As in Chapter 4, we will concentrate on the *before* placement problem; that is, communication placement corresponding to the non-local reads. Non-local writes require a very similar analysis. The resource-based communication placement analysis for non-local reads can be divided into following four phases.

Resource-independent send placement: First, we determine send placement without taking resource constraints into account. As described in Section 4.3, we make two passes over the interval-flow graph to determine send placement. Conceptually, in the first pass, we place SEND(d) at the node that uses the data section corresponding to d and then hoist it up in the interval-flow graph to obtain all the nodes that are safe (that is, satisfy the safety criterion). In the second pass in this phase, we use data-flow equations to select the earliest safe node along each execution path.

Resource analysis: We use the resource-independent send placement to compute resource usage for every node in a program (Section 5.4). As described in Section 5.1, the placement of SEND primitives determines the total buffer usage. Besides increasing the size of non-local buffers, earlier placement of SENDs also increases the range over which the data, and the non-local buffers, need to be live. Longer live ranges of non-local buffers can cause more buffers to be live simultaneously and further increase the total size of non-local buffers required. Moreover, the buffer usage is independent of the receive placement because the live range of non-local buffers depend only on the last use of the communicated data and not on the placement of the RECV primitive. Thus, before performing resource analysis we determine only the resource-independent send placement.

Resource-based send placement: The buffer usage determined by the resource analysis phase is used to find the nodes that exceed the maximum buffer constraint. For each node that exceeds the resource constraint, we initialize appropriate data-flow variables (Section 5.6). Next, using the resource-conscious initialization of data-flow variables, we determine the resource-based send placement. This phases uses the same data-flow equations for send placement as the resource-independent send placement phase with the only difference being the initialization of appropriate data-flow variables. Note that if sufficient resources are available, our framework determines the same placement as that determined by constraint-independent frameworks [44].

Latest balanced receive placement: Finally, as described in Section 4.4, we use the (resource-based) send placement to determine the corresponding receive placement.

Since non-local write is a dual problem, resource-based communication placement for non-local writes also requires the above four phases; but, in reverse order. In other words, we first compute resource-independent receive placement (since RECVs and not SENDs determine the live ranges of non-local write buffers); second, we perform resource analysis; third, we determine resource-based receive placement; and, finally, compute the earliest balanced send placement.

5.3 Safety criterion

This section describes the modifications required to incorporate resource constraints in the data-flow equation used for selecting safe nodes. In our data-flow equations, we use SAFE(n, d) to indicate whether node n is a safe placement location for SEND(d). As described in Section 4.3, for descriptor d, node n can be marked *unsafe* for one of the following reasons: (1) (traditional) data-flow analysis: the communicated data corresponding to d is not used along all terminating paths starting at n or (2) dependence analysis: n is a header node and the corresponding loop carries a true dependence that affects the communication corresponding to the descriptor d. To incorporate resource constraints in our framework, we extend the safety criterion as follows. We deem a node *unsafe* if it either does not satisfy the two safety criterion mentioned above *or* (3) if it causes the non-local buffer requirement to exceed the maximum buffer available.

In the presence of resource constraints, in addition to using the dependence information to initialize SAFE(n, d) variable (Section 4.3), we use the results of resource analysis. For each node n that exceeds resource constraint(s), we initialize appropriate SAFE(n, d) variables to $\perp$. Section 5.6 presents more details about this initialization.

Figure 5.3 shows the example program that is used through out this chapter to illustrate resource-based communication placement analysis. Figure 5.4 shows the earliest placement of SEND primitives and the latest balanced placement of RECV primitives without taking resources into account. The figure does not show the guards that enclose the communication primitives; appropriate guards are inserted later in the code generation phase of the Fortran D95 compiler. Since the array x is not modified in the program, SEND x[ ] can be hoisted to node 2 (by definition, we assume that every data set gets modified at the start node s). SEND y[ ] is placed at node 13 due to the initialization described above. Also, SEND z[ ] can not be hoisted before node 4 because the communicated data is not used along the path {1,2,3,8,9,11,12,15}. Placement of RECV x[ ] depicts what we mean by latest balanced placement; RECV x[ ] is placed at node 9 since x is used in the interval headed by the node 9. Now,

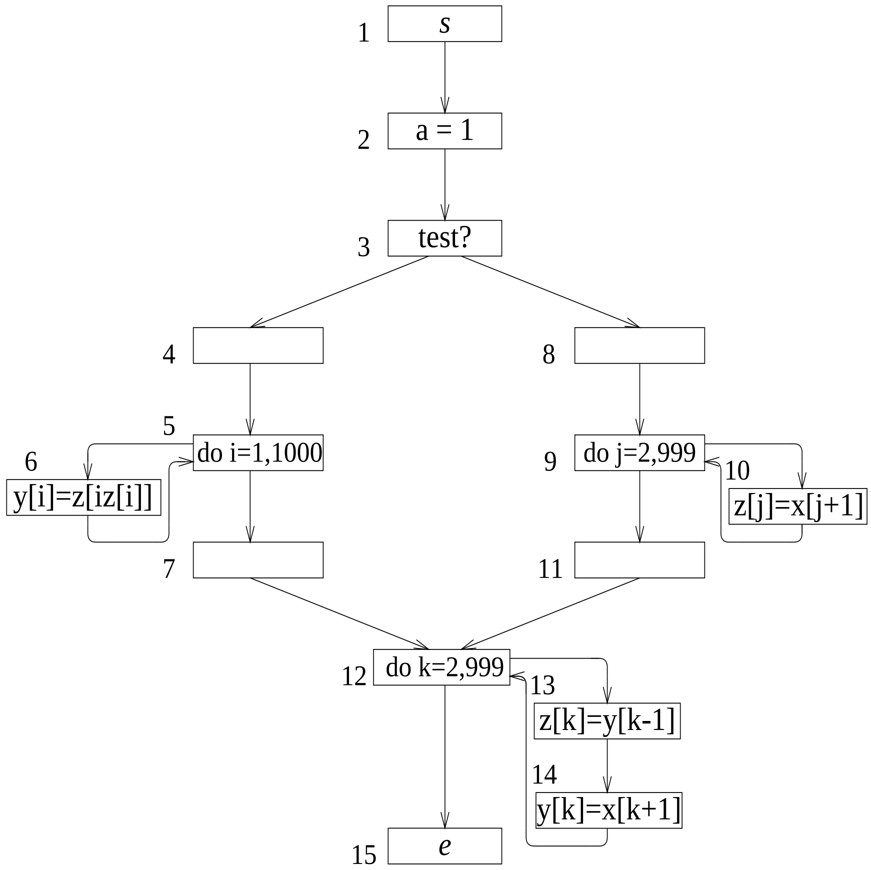


Figure 5.3 Example interval-flow graph used to illustrate resource-based communication placement.

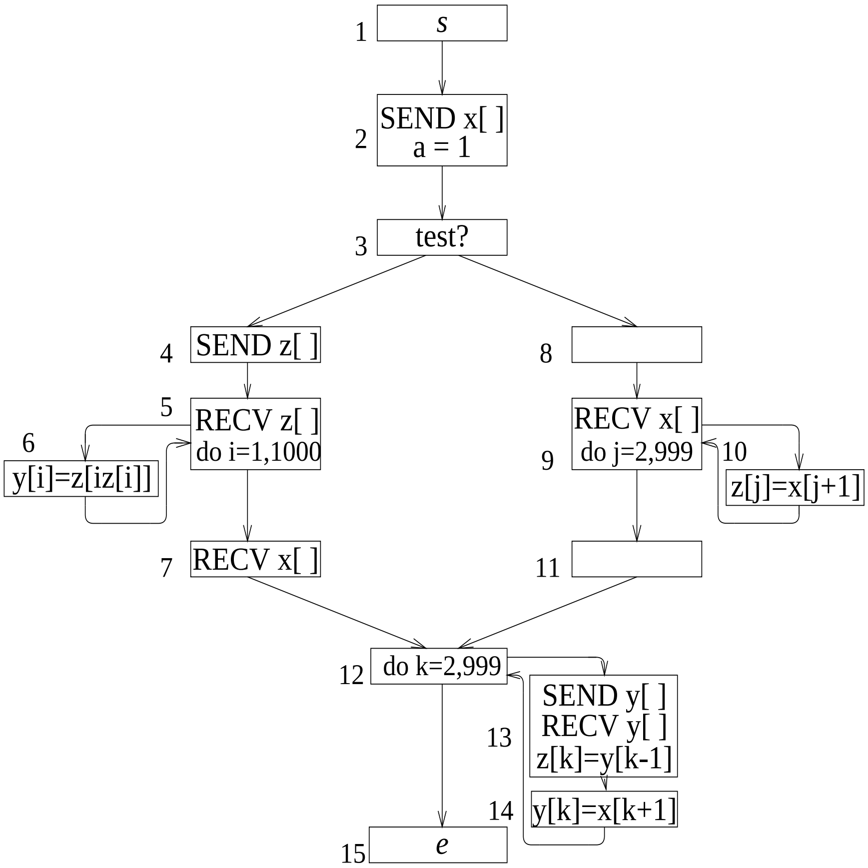


Figure 5.4 Resource-independent communication placement for the example program.

since the right branch {8,9,11} of the conditional contains a matching SEND x[ ] and RECV x[ ] pair, the left branch must also contain a RECV x[ ] for balanced placement. We use data-flow equations to determine node 7 as the latest placement location. Similarly, RECV y[ ] and RECV z[ ] are placed at the latest locations (nodes 13 and 5, respectively).

5.4 Resource analysis

This section describes the analysis required to compute buffer requirement for non-local read accesses. A similar analysis can be performed to determine the buffer required for non-local writes. We compute the buffer requirement for a processor by estimating the number of non-local accesses made by non-local references. However, instead of computing non-local buffer required by individual processors, we compute the *total* buffer required by all the processors. To compute the total buffer requirement, we compute the maximum non-local buffer required by a processor and multiply it by the number of processors. Though this approach forces us to over-estimate the buffer requirement in some cases (for example, in case of multi-spread communication pattern, since it does not involve all processors), we can handle the commonly occurring communication patterns such as shift, broadcast (all except one processor require same amount of non-local buffer), all-to-all communication, etc. precisely.

Let the maximum non-local buffer size allowed per processor, *maxBuffer*, be 250 array elements (for the sake of clarity, we express the size of the buffer in terms of array elements and assume that all arrays in Figure 5.3 are of the same type). Also, let the number of processors be 4. Therefore, the total buffer available among the four processors, *totalBuffer*, is 4*250 = 1000 array elements.

5.4.1 Resource analysis for individual loop nests

We first compute the non-local buffer requirements for individual loop nests by ignoring the statements outside the loop nest and ensure that no single loop nest or message exceeds the specified maximum buffer. This section uses "message d" to imply the message corresponding to the descriptor d. Since the loops can be nested inside other loops, traverse the interval-flow graph in reverse pre-order.

Let $Node(n)$ be the size of the buffer required for non-local accesses at node n. For example, in Figure 5.4, $Node(13) = Node(14) = 4$ (1 array element per processor) corresponding to the references y[k-1] and x[k+1]. $Node(n) = 0$ if there are no non-local accesses at node n.

Let $Buffer_{in}(n)/Buffer(n)$ be the size of live non-local buffers excluding/including the buffers corresponding to the messages sent at node n. Buffer required at node n, $Buffer(n)$, is the sum of the buffer requirement of node n and the size of the live messages at the exit of node n, given by $Buffer_{in}(s)$ for a forward edge successor s of n. To compute the buffer required at the start of node n, $Buffer_{in}(n)$, we subtract the total size of the messages sent at the node n. Equations 5.1 and 5.2 in Figure 5.5 give the corresponding equations for a non-header node n. The $\oplus$ operator is an addition operator that takes overlapping messages into account. Section 5.7.1 describes its implementation in the Fortran D95 compiler. SEND_SET(n) is the set of messages sent at the nodes in $T(n) \cup \{n\}$ and $msg\_size(d)$ is the size of the message d; it is updated as communication is hoisted out of loop(s). ($T(n) = \emptyset$ if n is not a header node.) For example, since node 14 is the last node in the interval, $\textsc{Succs}^F(14)$, the forward edge successors of node 14, is $\emptyset$. Since no messages are sent at node 14, by Equations 5.1 and 5.2, $Buffer(14) = Buffer_{in}(14) = Node(14) = 4$. Now, $Buffer(13)$ is computed as follows: $Buffer(13) = \oplus(Node(13), Buffer_{in}(14)) = \oplus(4, 4) = 8$ since

$$
\begin{aligned}
\mathit{Buffer}(n) &= \oplus_{s\in\textsc{Succs}^F(n)}(\mathit{Node}(n), \mathit{Buffer}_{in}(s)) && (5.1)\\
\mathit{Buffer}_{in}(n) &= \mathit{Buffer}(n) - \sum_{d\in\mathrm{SEND\_SET}(n)} \mathit{msg\_size}(d) && (5.2)\\
\mathit{Path}(n) &= \max_{s\in\textsc{Succs}^F(n)} (\mathit{Path}(s), \mathit{Buffer}(n) - \\
&\qquad \sum_{d\in\mathrm{HOISTED}(h)} \mathit{msg\_size}(d)), h = \textsc{Header}(n) && (5.3)\\
\mathit{Loop}(h) &= \max(\mathit{Loop}(h), \mathit{Path}(n)), h = \textsc{Header}(n) && (5.4)\\
\mathit{Buffer}(h) &= \oplus_{s\in\textsc{Succs}^F(h)}(\mathit{Loop}(h), \mathit{Buffer}_{in}(s), \\
&\qquad \otimes(\text{number-of-iterations}, \mathit{Buffer}_{in}(\textsc{Succs}^E(h)))) && (5.5)\\
\mathit{Total}(n) &= \mathit{Path}(n) + \mathit{Total}(\textsc{Succs}^F(\textsc{Header}(n))) && (5.6)\\
\mathit{Avail}(n) &= \mathit{total\_buffer} - \mathit{Loop}(n) - \mathit{Total}(n) && (5.7)
\end{aligned}
$$

Figure 5.5 Equations for computing the non-local buffer requirement.

the SEND x[ ] and SEND y[ ] do not overlap. By Equation 5.2, $\mathit{Buffer}_{in}(13)$ = 8 - 4 = 4 since SEND y[ ] is placed at node 13.

Let $\mathit{Path}(n)$ be the maximum buffer required for the communication primitives placed on any path from n to the *post-body node* of the interval. As shown in Equation 5.3, it is computed by determining the maximum buffer at the successor nodes and comparing it with the size of messages that are live at the node and are *not* hoisted out of the enclosing loop (that is, $\mathit{Buffer}(n) - \Sigma_{d\in\mathrm{HOISTED}(h)}\mathit{msg\_size}(d)$). For header node h, HOISTED(h) gives the messages that are hoisted out of the interval headed by the node h. Note that $\mathit{Path}(\textit{post-body node}) = 0$. As an example, $\mathit{Path}(14) = \max(0, \mathit{Buffer}(14) - \Sigma_{d\in\mathrm{HOISTED}(12)}\mathit{msg\_size}(d)) = \max(4 - 4) = 0$. $\mathit{Buffer}(14) - \Sigma_{d\in\mathrm{HOISTED}(12)}\mathit{msg\_size}(d)$ = 4 - 0 = 4 since SEND x[ ] has been hoisted out of loop 12. However, $\mathit{Path}(13) = 4$, since SEND y[ ] is placed in the loop.

Let $Loop(n)$ be the non-local buffer size for the messages placed in the loop (interval) with header node n. $Loop(n)$ is initialized to zero for all nodes n. To compute $Loop(h)$ for header node h, we use $Path(n)$ as shown in Equation 5.4 in Figure 5.5.

As shown in Equation 5.5, for a header node h, $Buffer(h)$ is computed as the sum of the buffer required for messages placed inside the loop ($Loop(h)$), messages hoisted across the loop header node ($Buffer_{in}(s)$), and messages hoisted out of the loop ($\otimes$(number-of-iterations, $Buffer_{in}(\textsc{Succs}^E(h))$)). $Buffer_{in}(\textsc{Succs}^E(h))$ gives the size of the messages that can be hoisted across the first node in the loop body. The multiplication operator, $\otimes$, determines the number of non-local accesses made in the loop by taking the overlapping messages corresponding to different references into account (see Section 5.7.1). As an example, node 12 in Figure 5.4 corresponds to a simple application of the $\otimes$ operator: only SEND x[ ] can be hoisted across node 13; the $\otimes$ operator recognizes that the communication pattern is shift and that only the boundary iterations need to fetch non-local data. Thus, $Buffer(12) = \oplus(Loop(12), Buffer_{in}(15), \otimes(1000, Buffer_{in}(13))) = \oplus(4,0,4) = 8$, where $Loop(12) = 4$ since SEND y[ ] is placed inside the loop and, as described above, $Buffer_{in}(13) = 4$.

Precise computation of *Loop*, *Buffer*, and $Buffer_{in}$ variables require the compile-time knowledge of the loop bounds. In case the loop bounds are unknown, the loop bounds can be assumed to be either very small (and all the messages fit in the buffer) or very large (such that no message fits in the buffer) [77].

5.4.2 Resource analysis across loop nests

As described before, besides increasing the size of non-local buffers, earlier placement of SENDs also increases the range over which the non-local buffers need to be live and this further increase the total size of non-local buffers required. The analysis in Section 5.4.1 does not guarantee that the sum of buffer requirement for messages from different loop nests is less than the maximum non-local buffer. This section describes the effect of this requirement on the placement of SENDs.

Let $Total(n)$ be the maximum buffer required for messages placed on any path from n to e and $Avail(n)$ be the size of the buffer available at node n. As shown in Equation 5.6 in Figure 5.5, the size of the buffer required on a path from n to e has two components: size of the messages for a path from n to the last node in the interval *post-body node*, $Path(n)$, and the buffer required for a path from *post-body node* to e, $Total(\textsc{Succs}^F(\textsc{Header}(n)))$.

$Avail$ at a particular node depends on the amount of buffer reserved for messages that have a later use: the sum of buffer required to execute a path from n to e (including the buffer required at node n itself), $Total(n)$, and the buffer required for the communication primitives placed in the interval $T(n)$, $Loop(n)$; Equation 5.7 in Figure 5.5. Clearly, $Avail(e) = total\_buffer$.

Figure 5.6 shows the values of *Buffer* and *Avail*. These are computed by the propagating their values through the interval-flow graph. Let's see how $Buffer(5)$ and $Avail(5)$ are computed. $Buffer(5) = \oplus(Loop(5), Buffer_{in}(7), \otimes(1000, Buffer_{in}(6)))$. Now, since SEND z[ ] has been hoisted out of the loop, $Loop(5) = 0$; $Buffer_{in}(7) = 4$ since SEND x[ ] is live at node 7; $\textsc{Succs}^E(5) = 6$ and $Buffer_{in}(6) = 4$ because SEND z[ ] can be hoisted across node 6. Thus, $Buffer(5) = \oplus(0, 4, 1000) = 1004$ and $Avail(5)$ = 1000-1004 = -4. Note that $\otimes(1000, 4) = 1000$ since, though each iteration of the "**do** i" loop potentially accesses non-local data, there are only 250 iterations per processor (due to the owner-computes rule; Section 4.1.1).

5.5 Resource-based placement: NP-complete problem

The problem of determining a placement that makes optimal use of resources while minimizing communication overheads can be shown to be NP-complete. This problem

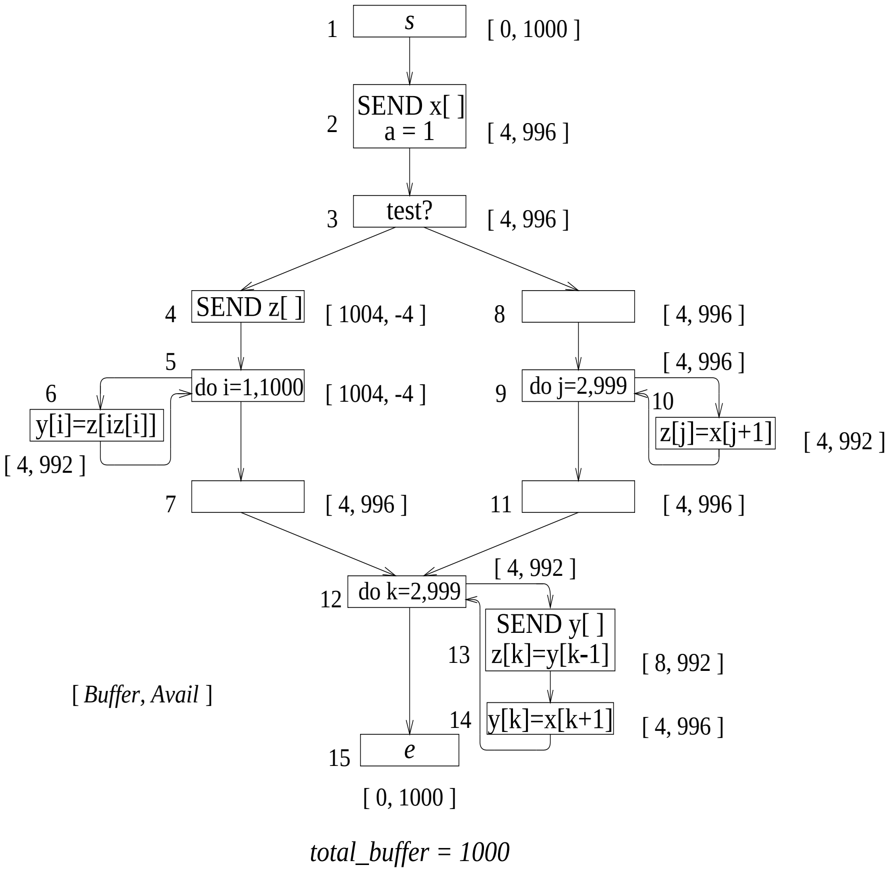


Figure 5.6 *Buffer* and *Avail*, respectively; shown adjacent to the nodes.

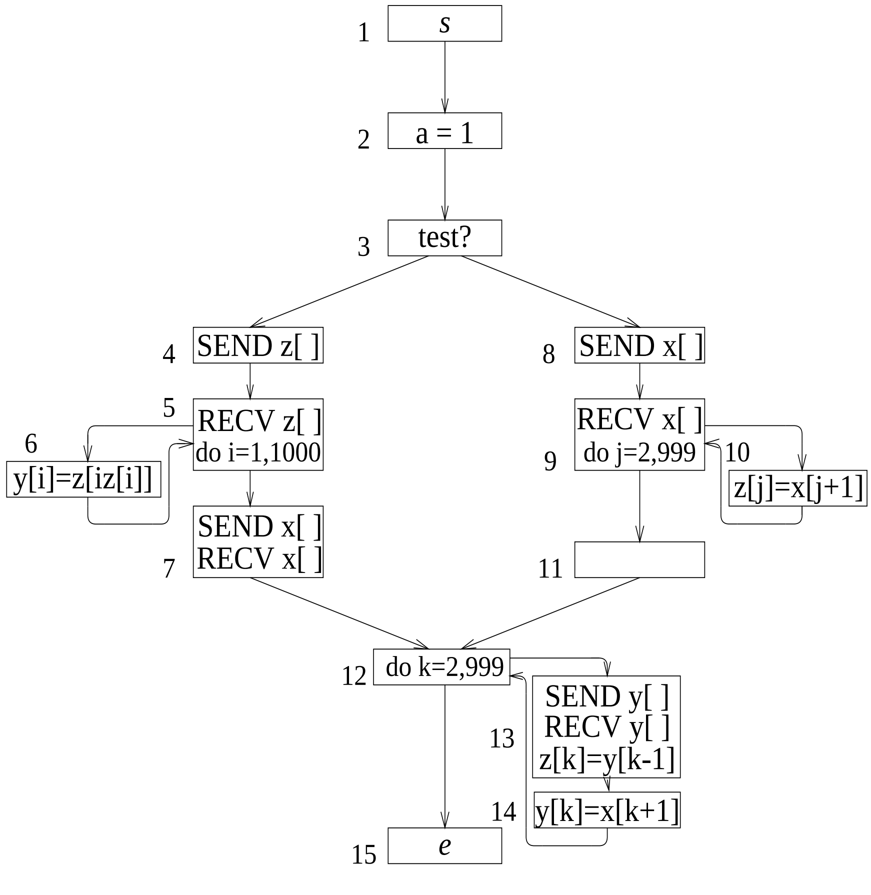


Figure 5.7 Communication placement after taking resources into account.

is very similar to the ones encountered in scheduling and database management (file allocation) problems [32, 10] and can be similarly shown to be NP-complete: the knapsack problem [32] can be trivially reduced to the problem of optimally selecting the messages that should use the available buffer at any given node. The knapsack problem is a classical problem in operations research where a scarce resource (here, the non-local buffer) is to be allocated to a number of competing users.

Since the problem of minimizing communication overhead while maximizing buffer usage is a NP-complete problem, it requires a heuristic to select communication instances that can be ignored while minimizing communication overhead. The heuristic is invoked whenever the data-flow algorithm encounters a node that violates resource constraint(s).

An efficient heuristic for the above problem is as follows. *If buffer usage at a node exceeds the maximum buffer available, block hoisting* SENDs *across the node.* Since the algorithm first hoists communication *out* of the loop intervals and then hoists communication *across* nodes, the heuristic favors reduction of communication frequency over maximizing latency hiding. This heuristic is based on the experiences with compilation for distributed-memory machines: it has been observed that reducing frequency of communication often yields more significant benefits than the latency hiding optimization [51].

Message buffers for communication blocked at a node are deallocated. This frees up buffers and allows communication to be hoisted to the outermost placement locations. Section 5.6 describes the implementation of the above-mentioned heuristic.

5.6 Taking resources into account

This section describes how the results of the resource analysis described in Section 5.4 are used to implement the heuristic. First, if $Loop(h) > total\_buffer$ (Section 5.4.1) for header node h then SAFE(h, d) is initialized to $\perp$ for all descriptors d. This prevents messages to be hoisted across node h and, thus, inhibits both further vectorization as well as movement across the node. Consider the matrix multiply example from Section 5.1. Recall that we had assumed that the non-local buffer requirement for communication placed outside the "**do** k" loop exceeds the maximum buffer available. In other words, *Loop*("**do** k") exceeds the resource constraint and to enforce the constraint, SAFE variables corresponding to the descriptors of arrays a and b are initialized to $\perp$. This causes the communication primitives to be placed inside the "**do** k" loop, as desired.

Second, the value of $Avail(n)$, as computed in Section 5.4.2, can be less than 0. Therefore, the actual assignment is: $Avail(n) = \max(0, Avail(n))$. If $Avail(n)$ is less than zero before the correction then n does not have sufficient buffer available to allow messages to be hoisted across it. Node 5 in Figure 5.6 is an example of such a node. To prevent messages from being hoisted across node 5, the heuristic initializes SAFE(5, x[ ]) = $\perp$. Note that since *Loop*(5) does not exceed *total_buffer*, the heuristic does *not* initialize the SAFE variables corresponding to the messages hoisted *out* of the loop to $\perp$; in other words, we allow message to be hoisted out of the loop and only prevent messages from being hoisted across node 5.

The placement of SENDs with SAFE initialized as described above gives the resource-based placement of SEND primitives. The resource-based placement for our example program in Figure 5.3 is shown in Figure 5.7. SEND z[ ] is hoisted out of the loop and placed at node 4. However, since SAFE(5, x[ ]) = $\perp$, SEND x[ ] gets blocked at node 7. Given this SEND placement, the corresponding RECV placement is determined as described in Sections 5.2 and 5.3. RECV z[ ] is placed outside and before the loop (node 5). And, as described in Section 5.3, RECV x[ ] is placed at node 7 to ensure balanced communication placement.

5.6.1 Stripmining transformation

As described above, if *Loop*(h) exceeds the maximum buffer constraint, the placement framework inhibits message vectorization. However, to reduce communication frequency, we would like to combine as many messages as possible into a single message. To achieve this, we annotate each loop that inhibits message vectorization as a potential candidate for stripmining transformation. A later transformation phase then stripmines the annotated loops. Stripmining transformation reduces the frequency of communication since stripmining allows communication primitives to be placed outside the inner stripmined loop and, thus, communicate only once for each outer stripmined loop iteration. Clearly, the stripmining (blocking) factor depends on the maximum available buffer.

An application of this transformation is the "**do** k" loop in our matrix multiply example from Section 5.1. As described above, *Loop*("**do** k") exceeds the resource constraint; thus the "**do** k" loop is annotated as a candidate for stripmining transformation. We then stripmine the annotated loop and hoist communication out of the inner stripmined loop which reduces the communication frequency from n to n/k$block. If k$block = 1, a later optimization phase can eliminate the inner stripmined loop.

```
do k = 1, n
   SEND/RECV a(i$my$lb:i$my$ub, k)
   SEND/RECV b(k, j$my$lb:j$my$ub)
   do j = j$my$lb, j$my$ub
      do i = i$my$lb, i$my$ub
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      enddo
   enddo
enddo
```

Figure 5.8 Resource-based communication placement for matrix multiply.

```
do kk = 1, n, k$block
   SEND/RECV a(i$my$lb:i$my$ub, kk:kk+k$block-1)
   SEND/RECV b(kk:kk+k$block-1, j$my$lb:j$my$ub)
   do k = kk, kk + k$block - 1
      do j = j$my$lb, j$my$ub
         do i = i$my$lb, i$my$ub
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
         enddo
      enddo
   enddo
enddo
```

Figure 5.9 Resource-based stripmining transformation.

The communcation placement before and after stripmining transformation is shown in Figure 5.8 and 5.9.

5.7 Implementation details

The resource-based communication placement analysis described in the chapter has been implemented in the Fortran D95 compiler where it is currently used to find the placement of distributed-memory communication primitives. To implement the resource-based placement analysis, we associate two sets of data-flow variables with each node. First, *placement-related* data-flow variables such as *Used*, *Transp*, SAFE, etc. (Sections 4.2 and 5.3). Second, *resource analysis* data-flow variables, such as *Node*, *buffer*, etc., that are used to compute resource usage for each node (Sections 5.4 and 5.6).

The data-flow variables mentioned in Chapters 4 – 5 are required to determine resource-based placement of non-local read communication primitives. We associate a similar set of data-flow variables with non-local write communication placement problem. This section describes the implementation details of the buffer analysis for non-local reads.

The current implementation of the placement analysis in the Fortran D95 compiler allows the maximum non-local buffer constraint to be specified as a command-line argument at compile time. Currently, the specified total non-local buffer size is statically partitioned into non-local read and write buffer sizes. However, compiler can use sophisticated analysis to partition the total buffer size into non-local read and non-local write buffer sizes as follows: first, associate (static) communication costs with non-local reads and writes; second, compute the total non-local read and write buffer usage (without enforcing resource constraints); third, test if the total buffer usage exceeds the maximum; finally, if it does, partition the total buffer size into non-local read and write buffer sizes based on the respective buffer requirements and the associated costs.

Both the placement and resource analyses are based on the interval-flow graph described in Section 4.1.1. We select an appropriate traversal order to solve data-flow equations for resource-based placement analysis in a single pass. For example, reverse pre-order traversal of the interval flow graph allows resource analysis for non-local reads (Sections 5.4 and 5.6) to be performed in a single pass.

5.7.1 Implementation of $\oplus$ and $\otimes$ operators

Besides the data-flow variables associated with each node, we associate two sets; one each corresponding to non-local reads and non-local writes. These sets are used to maintain the messages that are live at the current node. The set of live messages for non-local reads is used to record the communication instances for which the use of the communicated data (non-local reference) has been encountered within the current interval (that is, on any path from the node to the *post-body node* of the interval) but not the corresponding SEND. (As mentioned above, we follow reverse pre-order (BACKWARD and UPWARD) traversal for non-local read buffer usage analysis; thus, the use is encountered before the corresponding SEND.) A message is deleted from the live message set after the corresponding SEND is encountered. Recall from Section 5.1 that the non-local buffer corresponding to a message is live between the SEND and the last use of the communicated data. In addition to the above, a message can also be deleted from the set of live messages if the buffer usage exceeds the maximum available buffer. This is because, as mentioned in Section 5.6, when the buffer usage exceeds the specified constraint, data-flow variables are initialized such that appropriate SENDs are *not* hoisted across the node and this forces the resource-based communication placement to insert the corresponding SENDs after the node. We incorporate this knowledge in the set of live messages by deleting the messages that get blocked at a node due to resource constraints.

To implement the $\oplus$ operator, we take the set of live messages at the current node and coalesce the messages in the set. Message coalescing is required because messages are not coalesced when they are added to the set of live messages. Message coalescing optimization combines overlapping messages into a single message. If two non-local references require identical messages, coalescing messages, in effect, eliminates one of them. This allows the $\oplus$ operator to avoid counting redundant messages multiple times. To implement $\oplus$, then, we just calculate the total buffer required by the coalesced messages. To calculate the size of the coalesced messages, we take the loop nesting level of the node into account.

The $\otimes$ operator is implemented as follows. As with the $\oplus$ operator, we take the set of live messages at the current node and coalesce the messages. Since we take the loop nesting level into account, the $\oplus$ operator gets implemented automatically. For example, the size of non-local buffer corresponding to z[iz[i]] at node 6 is 4 (one non-local access times the number of processors) while at node 5 it is 1000 (since all the 250 accesses can be potentially non-local, which is then multiplied with the number of processors). Note that to compute the set of live messages at an interval header (loop) node, we merge the sets of live messages from the entry node in the interval (that is, the first node in the loop body) with the set of live messages from the successor(s) of the header node.

All the equations presented in Section 5.4 inspect only a subset of incoming/outgoing edges. Also, as described above, the $\oplus$ and $\otimes$ operators can be implemented efficiently. Moreover, each of the four phases (Section 5.2) involved in the resource-based communication analysis make at most two uni-directional passes over G. Therefore, as discussed in Section 4.5, the complexity of our algorithm is $O(E)$.

5.8 Summary

This chapter demonstrated the importance of taking resource constraints into account during communication placement analysis. It presented a data-flow analysis technique for resource-based communication placement. The placement framework performs SEND, RECV, and resource analysis in separate phases. A separate phase for resource analysis allows initialization of appropriate data-flow variables to take resource constraints into account. It also described the use of the stripmining transformation to improve the efficacy of communication placement.

The resource-based communication placement analysis has been implemented in the Fortran D95 compiler, where it is currently used to determine placement for distributed-memory communication primitives. The framework also performs the analysis for integrating the stripmining transformation with the placement analysis. Resource-based communication placement presented in this chapter assumes that memory can be allocated and deallocated as required. Thus, placement of allocation and deallocation primitives depends upon the placement of communication primitives. The placement of memory management primitives is similar to the resource-based placement of communication primitives, and can be solved using the framework presented in this chapter.

Though the framework was presented in the context of communication placement for distributed-memory machines, it can be used for other memory hierarchy related code placement problems that involve resource constraints. For distributed shared-memory systems, the framework can be used for prefetch placement. Satisfying both the cache size constraint and the limited number of outstanding prefetches needs to be investigated. Another potential application of the framework includes placement of I/O primitives for programs involving out-of-core I/O. Moreover, the effect of unknown (at compile time) loop bounds on the communication placement needs to be further investigated.

Chapter 6

Communication Optimizations

The communication placement framework described in Chapter 4 incorporates various *communication optimizations* – techniques that reduce communication overhead. Essentially, the optimizations that depend on the placement of SEND and RECV primitives are naturally handled by the framework. In particular, since the framework hoists communication primitives to the outermost placement level, it provides message vectorization. It supports latency hiding optimization by determining non-atomic placement for communication primitives. Also, as described in Section 5.6.1, the framework performs the analysis required to support the loop stripmining transformation for reducing the frequency of communication in the presence of resource constraints.

Since the Fortran D95 compiler is a second generation data-parallel compiler, the communication optimizations described in this chapter have been discussed before in the literature. However, previous research efforts were focussed on developing communication optimizations based on dependence information alone. This chapter describes how data-flow analysis can be combined with dependence information to improve the scope as well as precision of many communication optimizations.

An important issue that has not been addressed so far is the interaction between resource constraints and communication optimizations. Communication optimizations, such as redundant communication elimination and message coalescing, are based on the assumption that the communicated data can be stored in a buffer and reused at multiple references that access identical non-local data. Although the control-flow of a program might indicate that the communicated data can be reused, limited buffer resources might limit the amount of reuse. Thus, communication optimizations depend not only on the control flow of the program but also on the resource constraints. The resource-based placement framework described in Chapters 4 and 5 allows communication optimizations to be based resource constraints as well. To take resource constraints into account, we perform communication optimizations *after* determining resource-based SEND placement (for non-local reads).[§]

Chapter 4.2 explains that communication optimizations are performed in the third subphase of send and receive placement analyses. In other words, communication optimizations are performed in separate phases and are not intertwined with the analyses to determine the placement of communication primitives. This leads to efficient placement analysis: for example, during the analysis, no comparisons are performed to evaluate whether different communication descriptors have overlapping data sets.

In addition, performing communication optimizations after determining the placement of communication primitives allows the optimizations to utilize the information gathered by the global data-flow analysis. As an example, consider the program in Figure 6.1. The figure shows the original program on the left. Assume that the compiler picks owner computes rule as the computation partitioning for the assignment statement in the "**do** k" loop. Therefore, all the four *rhs* references access non-local data. It is possible to combine the four overlapping messages for the four references into a single communication instance. However, since the "**do** j" loop modifies data corresponding to two references (b(k+2) and b(100)), combining messages will force the SEND of the coalesced message to be placed just before the "**do** k" loop. This placement does not yield any latency hiding. However, if we first determine the ear-

[§]For non-local writes, optimizations are performed after computing resource-based RECV placement.

liest placement of SEND primitives for each of the four non-local references and *then* combine the messages, the placement of SENDs will indicate the messages that can be combined without decreasing the extent of latency hiding. It can be seen from Figure 6.1 that SEND b[i+1] and SEND b[i+3] as well as SEND b[i+2] and b[100] can be combined while preserving the benefits of latency hiding optimization.

The rest of this chapter is organized as follows. Section 6.1 describes how the framework supports elimination of redundant communication. Section 6.2 describes how it supports the message coalescing optimization. Section 6.3 describes the data-flow analysis performed to support the vector message pipelining optimization.

6.1 Partially redundant communication elimination

As mentioned in Section 4.1.1, our placement framework supports *partially* redundant communication elimination. Recall that when non-local data are available along some but not all paths to a node, the data are said to be *partially* available. From example, in Figure 4.2, the framework allows the non-local data to be fetched only once along both the branches and to be used twice along the right branch.

The framework handles partially redundant communication elimination in a manner identical to the message coalescing optimization (Section 6.2). This is based on two observations. First, redundant communication elimination and message coalescing are *similar* optimizations: redundant communication elimination combines identical messages while message coalescing combines overlapping messages. Second, the framework associates a distinct communication descriptor with each non-local reference and does not eliminate redundant descriptors or combine overlapping descriptors while performing the placement analysis. In other words, both message coalescing (which combines overlapping communication sets) and redundant communication elimination (which, in essence, combines completely overlapping communica-

```
real a(100), b(100)
processors p(4)
template t(100)
align with t :: a, b
distribute t(block) onto p

do i = 1, 100, 2
  b(i) = ...
enddo
⋮
do j = 2, 100, 2
  b(j) = ...
enddo
do k = 2, 96, 2
  a(k) = b(k+1) + b(k+2) +
           b(k+3) + b(100)
enddo
```

```
do i = 1, 100, 2
  b(i) = ...
enddo
SEND b[k+1]
SEND b[k+3]
⋮
do j = 2, 100, 2
  b(j) = ...
enddo
SEND b[k+2]
SEND b[100]
do k = 2, 96, 2
  a(k) = b(k+1) + b(k+2) +
           b(k+3) + b(100)
enddo
```

Figure 6.1 An example program to illustrate the benefits of performing communication placement and optimizations in independent phases.

tion sets; since the communication sets are identical, it "eliminates" one of them) are performed after computing SEND placement for all descriptors. Since the framework hoists SENDs to the earliest program locations even in presence of arbitrary control flow, no additional analysis is required to support partially redundant communication elimination.

6.2 Message coalescing

As mentioned above, message coalescing avoids redundant communication by combining partly or completely overlapping communication sets. It reduces both the communication volume as well as the communication frequency.

Message coalescing optimization is used to coalesce SENDs placed at the same node as follows. After determining the earliest SEND placement for all communication descriptors, the SENDs placed at the same node are compared to test if they can be coalesced. In the Fortran D95 compiler, we coalesce messages only if they have the same communication pattern (Section 3.2) and identical set of sending and receiving processors. If two SENDs can be coalesced, a new descriptor is added to the communication descriptor universe COMM and the two coalesced SENDs are replaced by a new, coalesced SEND. In Figure 4.4, SEND x[i] and SEND x[k] have identical data sections as well as identical set of sending and receiving processors. Thus, the two SENDs are coalesced and SEND x[ ] corresponds to the coalesced SEND. If different SENDs have overlapping (but not identical) data, then the coalesced message is made responsible for communicating the entire data section. On the other hand, if identical data sections are coalesced, as in Figure 4.4, then it corresponds to redundant communication elimination. Finally, we determine the RECV placement corresponding for the SENDs primitives of the coalesced communication sets. Figure 4.7 shows the placement of RECV x[ ] corresponding to the coalesced SEND x[ ].

Two separate and independent phases for SEND and RECV placement simplify the implementation of message coalescing optimization. Since SENDs are coalesced before determining the corresponding RECV placement, SENDs can be coalesced without worrying about coalescing the matching RECVs. If placement for both SEND and RECV primitives is determined in the same phase, as is done in the GIVE-N-TAKE framework [44], ensuring the balancedness of coalesced messages becomes a nontrivial problem. Moreover, unlike the previous dependence-based placement techniques, communication placement using data-flow analysis allows implementation of message coalescing optimization across loop nests.

It must be pointed out that coalescing SENDs can cause the corresponding RECVs to be placed earlier and, thus, reduce the extent of latency hiding. Though more research needs to be done to evaluate the impact of this, message coalescing optimization will be beneficial in general because it reduces the frequency and volume of communication. The benefits from these optimizations should offset the decrease in latency hiding.

6.3 Vector message pipelining

Message pipelining hides communication latency by moving SEND and RECV primitives *across iterations* such that data are sent as soon as they are computed and received just before they are used. This differs from the placement of SEND and RECV primitives described in Chapter 4 because communication primitives can be moved to *earlier iterations*. As mentioned before, in the distributed-memory compilation literature, the optimization that hoists communication to the outermost placement level is known as message vectorization [48]. The placement that combines message vectorization with message pipelining is known as vector message pipelining (VMP) [48]. Traditional dependence-based communication placement techniques performed VMP by placing SEND immediately after the source of the dependence. But clearly, in the presence of conditionals, balance cannot be guaranteed with dependence-based placement. On the other hand, *none* of the previous data-flow based communication placement frameworks [40, 44] provided support for VMP.

We exploit the knowledge about the graph structure and combine it with the equations for SEND placement to support VMP. First, since our framework uses interval analysis, we are able to recognize the loop headers as well as the first and last nodes in an interval. Moreover, our framework has separate equations for determining placement of send primitives. We support VMP by executing SEND placement phase for a second time – *after* the RECV placement phase.

Consider placement of SEND z[i-1] in Figure 4.4. Note that we can hoist SEND z[i-1] *along the entry and back edge* from node 6 to node 8 and then to node 7. This

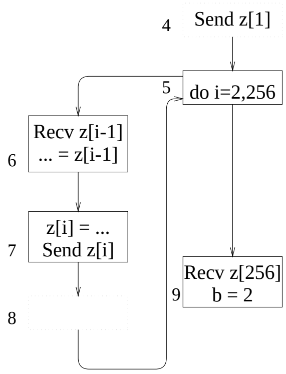


Figure 6.2 Vector message pipelining.

is the key observation and allows us to support VMP as follows. Each SEND(d) that is blocked at the start of an interval due to loop-carried dependence is hoisted along the entry and back edge of the interval. It is always possible to hoist SENDs in this manner due to the presence of the *last node* (recall that a post-body node is added to each interval; Section 4.1.1). The SENDs are then hoisted to the earliest safe location using the equations described in Section 4.3. Since data-flow equations are used to find the placement of SEND(d), our framework can support VMP for programs with arbitrary control-flow.

Hoisting a SEND to the *last node* corresponds to moving it to the previous iteration. Thus, moving SEND z[i-1] to node 8 and then to node 7 (Figure 4.4) requires converting it to SEND z[i], as shown in Figure 6.2. In fact, a SEND can be moved several iterations back so that the data are communicated as soon as they are computed. This can be achieved by using the data-dependence distance (or the minimum distance, if dependence distance is not a constant) to determine the number of iterations across which the corresponding SEND can be moved. Thus, by combining dependence information with data-flow analysis, we require only two extra passes (for SAFE and EARLIEST data-flow problems) over the interval-flow graph to support VMP.

Every time a SEND is hoisted along entry and back edge, it requires placement of an appropriate SEND at the forward edge predecessor(s) of the header node and an appropriate RECV at the forward edge successor(s) of the header node. As shown in Figure 6.2, the movement of SEND z[i-1] from node 6 to node 7 in Figure 4.4 inserts SEND z[1] at node 4 and RECV z[256] at node 9. If a SEND is hoisted across multiple iterations, it requires prologue and epilogue communication loops that initiate appropriate number of SEND and RECV primitives.

It should be stressed that though in Figure 6.2, we achieve only fine-grain pipelining, pipelining can often be combined with message vectorization. Even otherwise, if unlike in Figure 6.2, there were some computation after the assignment to z[i], then initiating SEND at node 7 would have hidden latency by overlapping communication with computation.

Figure 6.3 shows an example of a benchmark program that contains conditionals as well as loop-carried dependence. The code shown is extracted from SUB008, which is a 104 line subroutine in the parallel benchmark code from the Netlib repository. SUB008 presents opportunities for vector message pipelining and, thus, provides an example that requires VMP in presence of conditionals.

6.4 Summary

The applicability of communication optimizations depend not only on the control-flow of the program but also on the resource constraints. Compilers must ensure that communication optimizations don't cause the resource constraints to be violated. This chapter described how communication optimizations can be performed in the presence of resource constraints. Using resources to influence SEND placement before determining RECV placement allows the placement framework to modify SEND

```
distribute d(block), e(block) onto p
do l = 1, n
   ...
   do m = l, n
      ...
   enddo
   if (m .le. l) then
      ...
      p = (d(l+1) − d(l))/(2.0 ∗ e(l))
      ...
      if (l+2 .le. n) then
         do i = l+2, n
            d(i) = d(i) - h
         enddo
      endif
      ...
   endif
   d(l) = d(l) + f
enddo
```

```
do l = 1, n
   ...
   do m = l, n
      ...
   enddo
   if (m .le. l) then
      ...
      RECV d[l+1]
      p = (d(l+1) − d(l))/(2.0 ∗ e(l))
      ...
      if (l+2 .le. n) then
         do i = l+2, n
            d(i) = d(i) - h
         enddo
      endif
      ...
   else
      RECV d[l+1]
   endif
   d(l) = d(l) + f
   SEND d[l]
enddo
```

Figure 6.3 A program fragment extracted from SUB008 subroutine to illustrate the need for vector message pipelining in presence of conditionals.

placements without worrying about the matching RECVs and the balance placement criterion.

The chapter described how data-flow analysis can be combined with dependence information to improve the scope as well as precision of many communication optimizations. It described the implementation of vector message pipelining based on data-flow analysis; dependence distance is used to support vector message pipelining in a single pass over the interval-flow graph.

In addition, the chapter pointed out the benefits of communication placement framework that determines SEND and RECV placements using separate phases. Two independent phases simplify the implementation of redundant communication elimination and message coalescing optimizations. For example, two phases allow SENDs to be coalesced before determining the corresponding RECV placement and, thus, preclude the need for coalescing matching RECVs when SENDs are coalesced. If placement for both SEND and RECV primitives is determined in the same phase, ensuring the balance of coalesced messages becomes a non-trivial problem.

Chapter 7

Cyclic(k) Distributions: Communication Generation and Related Issues

Previous chapters described communication placement and optimization techniques. The techniques work with implicit communication descriptors and, thus, are applicable to all references, independent of the corresponding HPF-specific information such as data distribution directives. However, once the final placement of the communication descriptors has been selected, the compiler needs to compute a variety of explicit information related to the descriptors. Section 3.3 describes how some of the explicit information required by the computation partitioning and code generation phases is computed. We now turn our attention to communication generation techniques for a specific data distribution: *cyclic(k)* distribution. It is the most general regular distribution supported by HPF.

As mentioned in Section 2.1.1, *cyclic(k)* distribution partitions the specified array dimension into blocks of size k and then assigns these blocks to the processors in a cyclic fashion. *Cyclic(k)* distribution is useful in the programs that require both locality and load-balancing. For example, programs with triangular loops, such as Cholesky factorization [75] and Gaussian elimination [47], can benefit from *cyclic(k)* distribution. Unlike *block* distribution, *cyclic(k)* distribution is amenable to load-balancing issues. Moreover, unlike *cyclic* distribution, *cyclic(k)* distribution can yield better locality. Thus, *cyclic(k)* distribution provides the benefits of both *block* and *cyclic* distributions. The disadvantage is, as can be expected, the compilation techniques to support *cyclic(k)* distribution efficiently are more complicated.

Cyclic(k) distribution has only been studied in detail over the last couple of years. Previous research was focused only on compilation strategies for handling block and cyclic distributions efficiently [34, 70, 68, 85]. This chapter demonstrates that the accesses made by *cyclic(k)* references exhibit a repetitive access pattern. It exploits the repetitive pattern to present efficient compilation techniques for *cyclic(k)* distributions.

The rest of this chapter is organized as follows. Section 7.1 summarizes the *periodic access sequence* representation, which is used for expressing, computing, and manipulating *cyclic(k)*-based sets efficiently. Section 7.1.1 describes the *first* algorithm that was developed to compute periodic access sequences in linear time. Section 7.1.2 extends the periodic access sequence representation to handle subscripts with multiple index variables. Section 7.1.3 extends it further to support coupled subscripts. Section 7.2 then shows how the representation can be used during communication analysis. Section 7.3 discusses some of the code generation issues related to *cyclic(k)* distribution. Section 7.4 describes the union operation for periodic access sequences and its use in supporting message coalescing optimization. Finally, Section 7.5 presents experimental results that compare our approach with previous techniques. Section 7.6 describes some of the related work done to support *cyclic(k)* distributions. Section 7.7 summarizes our contributions.

7.1 Computing periodic access sequence

Section 3.3.2 describes the periodic access sequence representation for *cyclic(k)* distribution: elements of an array referenced by a particular reference can be represented using a sequence of accesses in the first cycle and the *period* of the cycle. The key observation, made by Chatterjee et al., is that the offset of an element determines the offset of the next element on the same processor [18]. Since the offsets range between

0 and $k-1$ (where k is the block size of the *cyclic(k)* distribution), by pigeon-hole principle, at least two of the first $k+1$ local memory locations on any particular processor must have the same offset. Since the offset of the next element depends only on the offset of the current array element, it proves that there exists a cycle of memory access gaps.

This section presents efficient algorithms for computing periodic access sequence for an array with *cyclic(k)* distribution and a reference with arbitrary affine subscripts. Section 7.1.1 presents a linear-time algorithm for computing periodic access sequence under certain special conditions and for subscripts with single index variables. Section 7.1.2 and 7.1.3 present efficient sequence construction techniques for subscripts with multiple index variables and coupled subscripts.

7.1.1 Subscripts with single index variable

Figure 7.1 depicts the access pattern corresponding to the following program fragment:

```
real A(1 : 80)
processors p(5)
distribute A(cyclic(4)) onto p

do i = 0, 80, 3
    A(i) = 100.0
enddo
```

As mentioned in Section 3.3.2, it is helpful to view one-dimensional array distributed with *cyclic(k)* distribution as a two-dimensional matrix. Figure 7.1 shows the accesses made by the above-mentioned loop using the two-dimensional matrix view. As can be observed from the figure, the access pattern on any processor can be represented by a sequence of length k $(= 4)$. For example, the memory access sequence for processor 1 is [6,24,27,45] and the *period* is spk (= 60; $s = 3$, $p = 5$, $k = 4$), where s = loop stride and p = total number of processors.

The access sequence for a particular processor m can be determined by computing the elements in the first cycle. In the two-dimensional matrix view, if we assume that columns are numbered starting with 0, columns mk to $(m+1)k-1$ are owned by processor m. Since each row has pk elements, array access $(l+sj)$ has an offset $(l+sj) \bmod pk$ with respect to the start of the row, where l = loop lower bound. (If the lower bound of the array is not zero, it must also be taken into account. However, it does not play any role in the access sequence computation and, thus, has been ignored in this chapter.) For each non-negative j, if the offset lies within the range $[mk, (m+1)k-1]$, it corresponds to an access made by processor m:

$$mk \ \le \ (l+sj) \bmod pk \ \le \ (m+1)k-1$$

The above equation is equivalent to

$$mk - l \ \le \ sj \ - \ pk\,q \ \le \ (m+1)k - l - 1.$$

The above inequality can be written as a set of k linear Diophantine equations in the variables j and q:

$$\{sj - pk\,q = \lambda \mid mk - l \ \le \ \lambda \ \le \ (m+1)k - l - 1\}.$$

Processor 0	Processor 1	Processor 2	Processor 3	Processor 4
$\boxed{0}$ 1 2 $\boxed{3}$	4 5 $\boxed{6}$ 7	8 $\boxed{9}$ 10 11	$\boxed{12}$ 13 14 $\boxed{15}$	16 17 $\boxed{18}$ 19
20 $\boxed{21}$ 22 23	$\boxed{24}$ 25 26 $\boxed{27}$	28 29 $\boxed{30}$ 31	32 $\boxed{33}$ 34 35	$\boxed{36}$ 37 38 $\boxed{39}$
40 41 $\boxed{42}$ 43	44 $\boxed{45}$ 46 47	$\boxed{48}$ 49 50 $\boxed{51}$	52 53 $\boxed{54}$ 55	56 $\boxed{57}$ 58 59
$\boxed{60}$ 61 62 $\boxed{63}$	64 65 $\boxed{66}$ 67	68 $\boxed{69}$ 70 71	$\boxed{72}$ 73 74 $\boxed{75}$	76 77 $\boxed{78}$ 79

Figure 7.1 Elements of array A with $cyclic(4)$ distribution on 5 processors. The squares mark the array elements accessed by $A(3i)$.

The equations can be solved independently (solutions exist for an individual equation if and only if λ is divisible by $\mathrm{GCD}(s, pk)$). The general solution of a linear Diophantine equation can be found using extended Euclid algorithm [20].

The smallest solution for each of these k Diophantine equations gives an access in the first cycle. Since solving the above system of equations requires computing only a single GCD, the accesses in the first cycle can be computed in $O(\log\min(s, pk) + k)$. However, these solutions are not in any particular order. The solutions need to be sorted to compute the sequence of local memory accesses. Based on this idea, Chatterjee et al. developed an algorithm for computing the memory access sequence. Due to sorting, the running time of their algorithm has $O(\log\min(s, pk) + k + k\log k)$ complexity, which is equivalent to $O(\min(k\log k + \log s, k\log k + \log p))$.

We developed the *first* linear-time algorithms for computing periodic access sequence for two special cases. The first case corresponds to the loops with access stride less than the block size: $s < k$. This case is expected to occur most frequently in practice. Figure 7.2 shows an example for this case; the loop shown is extracted from Livermore kernel 2 (Incomplete Cholesky-Conjugate Gradient). The second case, though not expected to occur as commonly as the first one, has s such that $s \bmod pk < k$. This occurs, for example, when s is equal to the array column size and pk divides the column size. Linear algebra codes that access consecutive row elements (instead of the column elements) of a linearized array fall under this category. Since $s \bmod pk < k$ implies $s < k$, the second case subsumes the first.

The important property satisfied by both the cases is that if a processor m executes the ith iteration then the (i+1)th iteration is executed either by processor m itself or by processor $(m + 1) \bmod p$. This observation is used to compute the next access made by a processor without solving Diophantine equations. It allows enumerating accesses made by a particular processor in the ascending order and eliminates the

```
do l = 1, n
   ...
   ipnt = ipntp
   ipntp = ipntp+il
   ...
   do k = ipnt+2, ipntp, 2
      i = i + 1
      X(i) = X(k) - V(k)*X(k-1) - V(k+1)*X(k+1)
   enddo
   ...
enddo
```

Figure 7.2 Livermore kernel 2 (ICCG excerpt).

need for sorting the accesses. This, as will be shown later in the section, reduces the complexity of our algorithm to $O(k)$.

Linear-time algorithm

To develop a linear-time algorithm, we make use of the following observations.

1. Accesses made by a processor can also be represented with the first element in the sequence along with a sequence of *access gaps*. For example, the accesses made by processor 1 in Figure 7.1 can be expressed either as the periodic access sequence [6,24,27,45] and period 60 or as [18,3,18,21] with first access 6. The algorithms presented in this section computes access gap sequence, which is represented as ΔM table.

2. Distributed-memory compilers usually allocate contiguous memory for local portions of the distributed arrays. In other words, consecutive blocks owned by a processor are allocated contiguous memory space. Thus, to iterate over the accesses made by a processor, we actually require the access gap sequence in the

local memory space. For example, the access gap sequence, ΔM table, in local memory space for processor 1 is [2,3,2,5]; there are 2 *local* elements between 6 and 24, 3 *local* elements between 24 and 27, etc.

3. For the two cases that we consider, the first access made by a processor can be computed in constant time without solving either the Diophantine equations presented earlier or the extended Euclid's algorithm.

4. If consecutive accesses are within the same block, the local memory gap between them is the same as the stride s.

5. For case I, since $s < k$, every block of k elements has at least one access; that is, there are no empty blocks.

In this dissertation, we will present details about the algorithm for the first case only. Figure 7.3 gives the algorithm for case I. The algorithm first computes the offset for $start$ – the first element accessed by processor m. Offset is computed with respect to the beginning of the current block; since each row is of size k, the offset is $start \bmod k$. For all accesses that lie in the same block, the algorithm sets local memory gap to s. When the algorithm encounters an access that lies outside current block boundary, which corresponds to an access whose offset is larger than $k - 1$, it computes the first element accessed in the next block owned by the processor. As mentioned before, if $s < k$, the next block is guaranteed to have an access. To compute the first element in the next block, we observe that there are $pk - (mk + \mathit{lastoffset} + 1)$ elements in the current row after the last access in the current block. Also, there are mk elements in the next row before the next block owned by processor m. Thus, there are at least $(pk - (mk + \mathit{lastoffset} + 1) + mk + 1)$ elements between the last access in one block and the first access in the next block. This observation is used in the algorithm to compute the first access in the next block. The offset of the first access in the next block is used to compute the local memory gap. Note that there are $k - lastoffset$ elements within the block after the last access in the current block and *offset* elements before the first access in the next block. Thus, the local memory access gap is $k - lastoffset + offset$.

The algorithm stops when it finds a non-empty table entry; as mentioned before, for a given stride s, block size k, and number of processors p, it can be proved that the offset of an access completely specifies the offset the next access on the processor [18]. Therefore, if the algorithm comes across a non-empty table entry, it is guaranteed that the sequence of access gaps will repeat.

Given the loop lower bound l, we can compute the first element accessed by processor m by the formula

$$start = l + s\lceil ((pk - l \bmod pk + mk) \bmod pk)/s \rceil$$

where $(pk - l \bmod pk)$ are the number of elements in the row after the lower bound l and mk are the number of elements before the first elements in the next row before the block owned by processor m starts. Number of elements that lie between the loop lower bound and the start of processor block are computed as $(pk - l \bmod pk + mk)$ modulo pk to incorporate the case when l and processor m's first block after l belong to the same row.

The algorithm presented in Figure 7.3 determines accesses under the assumption that the block-cyclically arrays are aligned perfectly. In case of an arbitrary affine alignment, as noted by Chatterjee et al., the accesses can be computed by two applications of the algorithm; first application computes the access sequence in the template space while the second computes the access sequence in the array space.

The algorithm for case II follows the same structure as that for case I. The condition $s \bmod pk < k$ implies that (just like $s < k$) that if processor m executes the ith iteration, (i+1)th iteration would be executed either on processor m or $(m+1) \bmod p$.

Input: First access made by the processor ($start$), block size (k), step (s), processor number (m), and number of processors (p).

Output: Access gap table (ΔM) and the length of the table ($length$).

Method:

```
// Initialize length and ΔM table.
length = 0;
for i = 1, k − 1
    ΔM[i] = ⊥;
endfor

offset = start mod k;
while (true) do
    // All the accesses within the same block have gap s.
    while (offset < k) do
        ΔM[offset] = s;
        offset = offset + s;
        length++;
    endwhile

    // Determine the first element accessed by the processor in the next row.
    lastoffset = offset − s;
    offset = (mk + lastoffset + s⌈(pk − lastoffset)/s⌉) mod k;

    // If the pattern begins to repeat, stop.
    if (ΔM[offset] ≠ ⊥) break;

    ΔM[offset] = k − lastoffset + offset;
    offset = offset + s;
    length++;
endwhile
```

Figure 7.3 Linear-time algorithm for computing local memory gap sequence for references with *cyclic(k)* distribution.

However, in this case, there will be empty blocks; the algorithm needs to keeps track of the number of skipped rows to compute the memory gaps correctly. However, this case is not expected to occur as frequently as case I. Therefore, we omit the details about the algorithm for case II; the algorithm has been described before [47].

This section presented a linear-time algorithm for generating local addresses for *cyclic*(k) distributions for the special case that occurs frequently. However, a linear-time algorithm for the general case will simplify handling of *cyclic*(k) distribution. As pointed out by Knies et al. address generation based on the local memory gap table makes a time versus space tradeoff [65]. Since a table is needed for every array reference with different stride or distribution, for large block sizes this can introduce a substantial memory overhead. In certain cases, the overhead of maintaining tables might not be acceptable, in which case techniques need to be developed to support *cyclic*(k) distribution without incurring the space overhead. (To this end, I collaborated with Kennedy and Nedeljković to develop a linear-time algorithm for the general case. Moreover, our techniques do not require tables for handling *cyclic*(k) distributions and support overlap areas for shift communication [62, 60]. The linear-time algorithm for the general case is not part of this dissertation.)

Complexity

First, as described in Section 7.1.1, the maximum length of the ΔM table can be k. Second, since the algorithm presented in Figure 7.3 stops as soon as the first non-empty table entry is encountered, a table entry is filled at most once. (Note that while the maximal possible length of the cycle of local memory gaps is k, the actual cycle length for any given processor depends on the GCD(s, pk) and that processor's starting location; thus, some ΔM table entries might not be filled.) In other words, the inner and outer loops together iterate at most k times. In addition, only a constant amount of computation is performed in each iteration. Therefore, the algorithm has $O(k)$ complexity.

Since the length of ΔM can be k, *any* algorithm for computing access gap sequence will have complexity *at least* linear to the size of the table k. In other words, access gap computation problem has $\Omega(k)$ complexity. Thus, our algorithm has $\Theta(k)$ complexity. Next section presents results to show the efficiency of our linear-time algorithm.

Experimental results

This section compares the address generation method described in Section 7.1.1 with the linear-time algorithm for the general case [62]. The linear-time algorithm for the general case is based on the observation that the accesses form an integer lattice. Thus, all accesses made by an array with *cyclic(k)* distribution can be computed using the basis vectors of the lattice. In addition, we showed that, for *any* processor, given an access made by the processor, the next element accessed by the processor can be determined by testing at most three cases. However, unlike the algorithm presented in Figure 7.3, the algorithm for the general case uses extended Euclid's algorithm to determine the first element accessed by a processor. In other words, it solves Diophantine equations to determine the smallest element accessed by a processor.

Figure 7.4 presents the graphs with the table construction times for the two methods: linear-time algorithm for the special case and the linear-time algorithm for the general case. The table construction times shown are the maximums over all the processors, averaged over 20 executions. The two methods were compiled using the *icc* compiler with $-$O4 optimization level and executed on 16 Intel iPSC/860 processors. The times were measured using *dclock()* timer. Since the special case algorithm handles only $s < k$, we restricted step size to be less than the block size. It can be seen that the algorithm presented in Figure 7.3 executes up to three times faster than

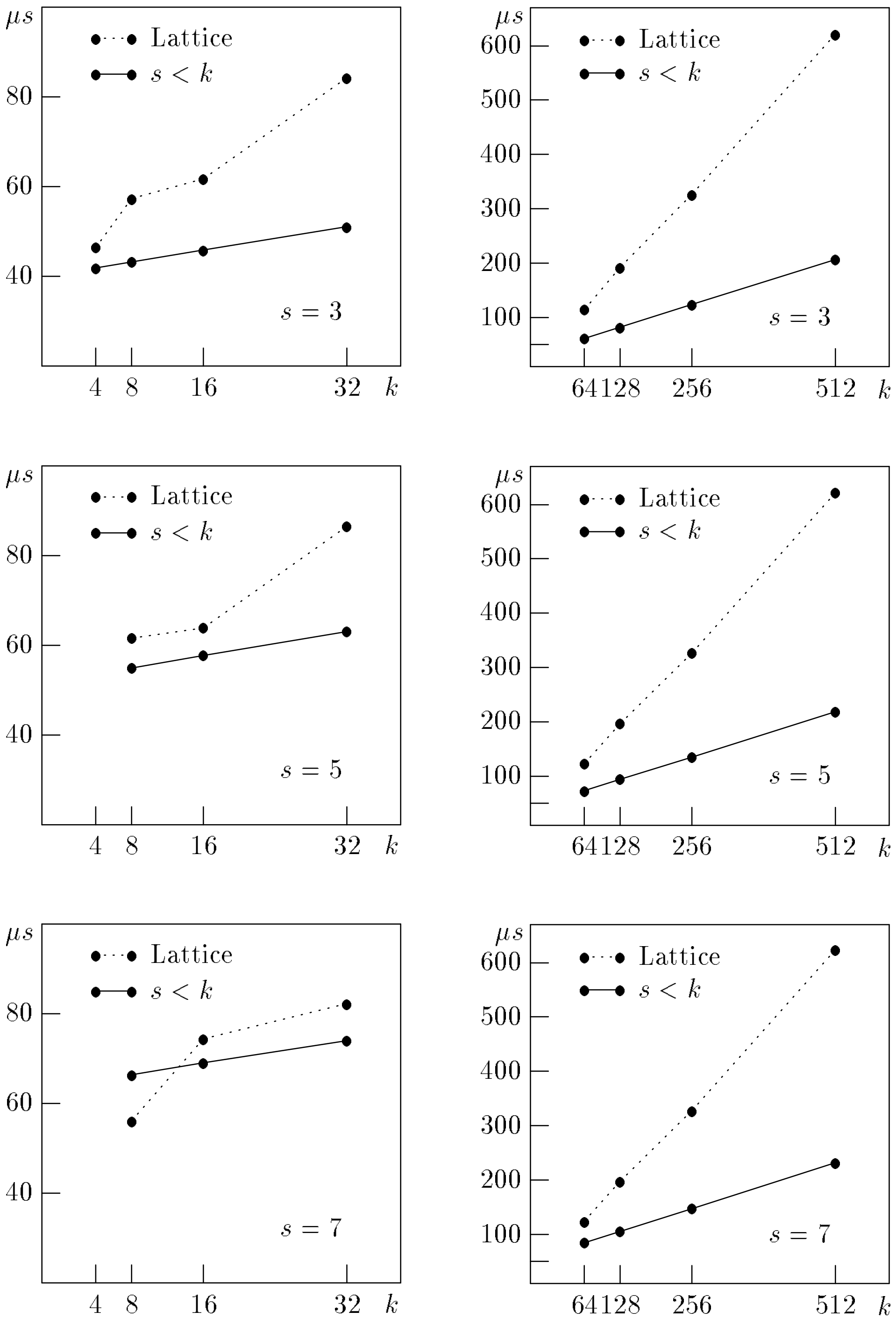


Figure 7.4 Table construction times for the two linear-time algorithms: the special case algorithm presented in Figure 7.3 ("$s < k$") and the lattice-based algorithm [62] ("Lattice").

the general case algorithm. This is because the special case algorithm computes the starting location for a processor without solving the Diophantine equations. It can be observed from the graphs that the table construction times for both the algorithms are independent of the access stride. Further, our experiments showed that the table construction times are also independent of loop lower bound.

We also compared the execution times of the table-based address generation scheme (the SPMD code using ΔM table is shown in Figure 7.5) and two techniques that had been described before: the run-time address resolution (also known as the guarded execution) and the virtual-block scheme proposed by Gupta et al. [41].

In the run-time resolution each processor executes the entire loop, and for each iteration every processor checks if it owns the array element that is being assigned to, in which case it computes the local memory address and performs the assignment.

In the virtual-block scheme the array is treated as being *block* distributed across a large enough number of virtual processors, and these virtual processors are then cyclically mapped onto physical processors. If stride s is not greater than block size k, then all virtual processors own some of the array elements being accessed, i.e., all virtual processors are *active* [41]. In that case, each processor loops over all virtual

Compute ΔM, *start, end*
$i = start$
$locOffset = start \bmod k$
while $(i \leq end)$ **do**
    $A(i) = 100.0$
    $i \mathrel{+}= \Delta M[locOffset]$
    $locOffset = i \bmod k$
endwhile

Figure 7.5 SPMD node code for subscripts containing a single index variable.

processors mapped to it (each of these virtual processors corresponds to a block that the processor owns), computes the lower and upper bound of array elements accessed within each virtual processor, and performs the translation from the virtual processor's local index space to its own local index space. This translation is needed only for the lower and upper bound, since stride s in the index space of a virtual processor on processor m remains unchanged in the index space of m. However, if $s > k$, not all virtual processors are active. In that case, the solution by Gupta et al. is to scan all the active virtual processors (each containing exactly one array access) and, for each one of them, check whether it is located on processor m. This, essentially, is the same procedure as that performed in the run-time address resolution.

All the experiments in this and the following sections were done on an Intel iPSC/860 hypercube, using the *icc* compiler with -O4 optimization level and *dclock* timer. All times are reported in milliseconds and represent maximums over 32 processors. The reported times do not include the time spent in constructing the tables. For several reasons the reported times do not include the time spent in constructing the table. First, if l, u (loop upper bound), and s, as well as distribution parameters p and k, are known compile-time constants, then the tables can be computed by the compiler, without incurring any run-time cost. Second, even if this computation has to be done at run time, same tables would typically be reused for multiple array references in the program. And finally, as can be seen from Figure 7.4, even for values of k as large as 512, table construction takes less than 1 millisecond, and thus would have no significant impact on results presented here. Therefore, the measurements were performed with the lower bound $l = 0$, while the upper bound was scaled in proportion to stride s, so that each processor accessed 10,000 array elements.

Both the virtual-block scheme and the run-time address resolution perform significantly worse than the methods that take advantage of repetitive patterns in local

Block size	Stride	Table lookup	Virtual block	Run time
$k = 4$	$s = 3$	2.3	60.4	1071.8
	$s = 25$	2.6	1094.8	1064.7
	$s = 100$	3.1	1094.6	1065.6
$k = 16$	$s = 3$	2.3	16.3	1076.8
	$s = 25$	2.6	1097.4	1067.2
	$s = 100$	3.1	1095.1	1065.0
$k = 64$	$s = 3$	2.3	5.2	1072.3
	$s = 25$	2.6	32.3	1077.9
	$s = 100$	3.0	1097.7	1067.6
$k = 256$	$s = 3$	2.3	2.3	1063.2
	$s = 25$	2.6	9.3	1075.6
	$s = 100$	3.1	32.8	1078.2

Table 7.1 Performance of different address generation methods for subscripts containing a single index variable.

memory addresses. While the run-time resolution consistently performs poorly, the performance of the virtual-block method depends on the values of s and k. As mentioned above, when $s > k$ the virtual-block scheme effectively reduces to run-time resolution, with small additional overhead incurred by virtual processor emulation. If $s \leq k$, the virtual-block method is significantly better than the run-time resolution, but it is only competitive with the other two methods when block size k is very large, and stride s is very small. In that case there are very few translations from a virtual processor's index space to a processor's index space, and many constant stride accesses are performed within each virtual processor. As can be observed from the Table 7.1, the table-based address generation scheme performs much better than the previously known techniques.

7.1.2 Subscripts with multiple index variables

This section shows how the techniques described in the previous section can be extended to solve the problem of address generation for array subscripts containing multiple index variables (MIV subscripts). Assuming again that the array A is distributed with $cyclic(k)$ distribution over p processors, our task is to generate the sequence of local memory addresses that a given processor m must access while executing its share of iterations of the following canonical loop nest

$$
\begin{array}{l}
\textbf{do } i = l_i,\ u_i,\ s_i \\
\quad \textbf{do } j = l_j,\ u_j,\ s_j \\
\quad\quad A(i+j) = 100.0 \\
\quad \textbf{enddo} \\
\textbf{enddo}
\end{array}
$$

Figure 7.6 shows the array elements accessed in the the loop nest specified by $k = 4$, $p = 4$, $(l_i, u_i, s_i) = (0, 130, 37)$ and $(l_j, u_j, s_j) = (0, 18, 2)$. For a given iteration of the outer loop, a dark shaded square indicates the first array element accessed in that iteration, while lightly shaded squares show starting locations for different processors. For example, the first array element accessed in the second iteration of the outer loop is $A(37)$ owned by processor 1, and the starting locations for processors 2, 3, and 0 are $A(41)$, $A(45)$, and $A(49)$, respectively. All other array accesses are shown in non-shaded squares. The key observation is that for a given iteration of the outer loop, which determines each processor's starting location, the sequence of array accesses for a given processor depends on the stride of the innermost loop only (2, in our example). Therefore, in a manner similar to that described in Section 7.1.1, the access sequence can be generated using the table of memory gaps (ΔM).

To use the ΔM table on processor m, we need to determine m's starting location for each iteration of the outer loop, that is, given a dark shaded square, we need

Processor 0				Processor 1				Processor 2				Processor 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	**37**	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	**74**	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	**111**
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143

Figure 7.6 Elements of an array distributed over 4 processors with *cyclic*(4) distribution accessed in a doubly nested loop with parameters $(l_i, u_i, s_i) = (0, 130, 37)$ and $(l_j, u_j, s_j) = (0, 18, 2)$.

to find the corresponding lightly shaded square that belongs to processor m. As described in Section 7.1.1, this can be done by finding the minimum of the smallest nonnegative solutions of k linear Diophantine equations. However, incurring the $O(k)$ cost for every iteration of the outer loop is hardly acceptable, and therefore we must look for a more efficient solution.

If the inner loop stride s_j is not greater than the block size k, processor m's starting location can be computed in constant time. If processor m owns $A(\mathit{first})$, the first array element accessed in a given outer loop iteration, then its starting location (in global index space) is *first* itself. If this is not the case, then m's starting location $start$ can be computed in the following way:

$$
\begin{array}{l}
gap = mk - \mathit{first} \bmod pk \\
\textbf{if } (\mathit{first} \bmod pk \geq k(m+1)) \textbf{ then} \\
\quad gap = gap + pk \\
\textbf{endif} \\
start = \mathit{first} + \lceil gap/s_j \rceil s_j
\end{array}
$$

As an example, for the second iteration of the outer loop in Figure 7.6, the starting locations for processors 0 and 3 are computed as follows. The first array element accessed is $A(37)$, and since $pk = 16$, we have $first \bmod pk = 5$. When $m = 0$, we get $k(m+1) = 4$, and therefore $gap = 0-5+16 = 11$ and $start = 37+12 = 49$. On the other hand, when $m = 3$, $k(m+1) = 16$ is greater than 5, and thus $gap = 12-5 = 7$ and $start = 37 + 8 = 45$.

If $s_j > k$, then the starting location for processor m can be computed using the following method, which although simple, turns out to be efficient in practice. Processor m needs to find the smallest t such that it owns $A(\mathit{first} + ts_j)$. In naive run-time resolution, this is done by incrementing t until an array element owned by processor m is reached (or $\mathit{first} + ts_j$ exceeds the loop upper bound). The ownership test requires expensive mod and divide operations. These can be avoided by working directly with the offsets. Let $\mathit{offset} = \mathit{first} \bmod pk$, be the offset of the first element with respect to the beginning of the row. Let $moveR = s_j \bmod pk$ and $moveL = pk - moveR$, the right and the left displacement due to incrementing the induction variable by s_j. For our example in Figure 7.6, $moveR = 2$ and $moveL = -14$. The starting location $start$ is found by incrementing and decrementing the offset by $moveR$ and $moveL$ respectively, until it falls within the range of offsets corresponding to processor m. The following procedure computes the starting location for processor m:

$start = \mathit{first}$; $\mathit{offset} = \mathit{first} \bmod pk$
while $(\mathit{offset} < km$ **or** $\mathit{offset} \geq k(m+1))$ **do**
  if $(\mathit{offset} < km)$ **then**
    $\mathit{offset} = \mathit{offset} + moveR$; $start = start + s_j$
  else
    $\mathit{offset} = \mathit{offset} + moveL$; $start = start + s_j$
  endif
endwhile

For the example shown in Figure 7.6, the starting location $start$ for processor $m = 0$ and $first = 74$ is computed as follows. The offset of the first element accessed by the third iteration of the outer loop is $(74 \bmod 16) = 10$, which is greater than $k(m+1) = 4$. Therefore, $moveL = -14$ is added to 10; the resulting $offset = -4$ is less than $km = 0$. Now, two additions of $moveR = 2$ to -4 bring the $offset$ within processor 0's range. Correspondingly, $s_j = 2$ is added three times to $first = 74$ to obtain $start = 80$, the required starting location.

Although we now have ways to compute a processor's starting location that are more efficient than solving k Diophantine equations, we still have not exploited the repetitive pattern of array accesses generated by the outer loop. In the same way that ΔM table reflects this pattern in the inner loop, we would like to construct the table of gaps between starting locations corresponding to consecutive iterations of the outer loop. Since the starting location can possibly have pk different offsets with respect to the beginning of the row, we can find the cycle of these offsets by computing the starting locations for at most $pk+1$ iterations of the outer loop. This cycle can then be used to compute the table (ΔG) of local memory gaps between consecutive starting locations. Although the maximal possible size of ΔG table is pk, the actual table size will depend on the length of the cycle of starting location offsets, and is likely to be much smaller than pk.

For all first elements accessed by the outer loop that have the same offset with respect to the beginning of the row, the gap between the first element and its corresponding starting location for processor m will be constant. Therefore, we can construct a table such that each entry $\Delta S[t]$ contains the number of array elements between the first element accessed by the outer loop iteration $first$, such that $first \bmod pk = t$, and the starting location for processor m. For example, the $\Delta S[15]$ entry of the table for processor 2 is 10 because offset 15 corresponds to the first element, 111, accessed by the fourth iteration of the outer loop and the starting location for processor 2 is 121; therefore, the number of elements that need to be skipped is $121 - 111 = 10$. The starting location for processor 2 is $Local(111 + \Delta S[111 \bmod 16])$ $= Local(121) = 29$, where $Local$ performs the translation from global to local index space.

Following is an $O(pk)$ method (linear in the table size) for constructing the ΔS table. From Figure 7.6, it can be observed that each of the first elements $A(4)$, $A(6)$, $A(8)$, $A(10)$, $A(12)$ and $A(14)$ accessed by the outer loop have $A(16)$ as the starting location for processor 0. Therefore, the ΔS table will have 12, 10, 8 , 6, 4 and 2 as the number of elements skipped for the first elements with offsets 4, 6, 8, 10, 12 and 14 respectively (that is, $\Delta S[4] = 12$, $\Delta S[6] = 10$, and so on). This observation can be used to obtain a linear-time method for computing the ΔS table for processor m: start with the offset km, find the offset of the next element (say, e) accessed for the processor m (using the method described in Section 7.1.1); all the non-local elements accessed (by following stride s_j) between the first element and the element e have e as the starting location on processor m; the number of elements skipped for each offset corresponding to different first elements can be computed easily. This process is repeated for all the offsets between km and $km + k - 1$; the uninitialized ΔS table entries correspond to the first elements accessed by the outer loop that have no corresponding starting location on processor m.

Since the ΔS table is generated based on the stride of the innermost loop only, the table can be used to find the starting location for a particular processor and a given first element (which depends on all enclosing loops whose index variables are present in the reference) irrespective of the number of loops enclosing the innermost loop. However, it introduces a global to local translation for every iteration of the outer loop.

As with the ΔM table, if the distribution and the loop parameters are known at compile time, the ΔS or the ΔG table can be computed at compile time. However, if the tables need to be computed at run time, since the table entries are not modified in the loop, they can be computed outside the loop nest. Note that the upper bound computation is simpler because the local upper bound need not be the actual last element accessed by the processor. The procedures for computing the local upper bound (*GetUpperBound*) and for global to local index conversion (*Local*) were presented in [47].

Experimental Results

This section compares the performance of various address generation approaches for array references with MIV subscripts. Figure 7.7 shows two versions of the SPMD node code corresponding to our canonical loop nest example. The code in Figure 7.7(a) uses the ΔS table to compute starting locations for each iteration of the outer loop, while the code in Figure 7.7(b) does this using the ΔG table. In the inner loop, both versions use the ΔM table to generate local memory addresses. We also implemented the method that uses the ΔM table for accesses within the inner loop, but generates starting locations without any tables using one of the two methods described above. Finally, we compared these methods with naive run-time resolution, since, to the best of our knowledge, this is the only other technique that guarantees that array accesses will not be reordered.

Table 7.2 contains execution times for different values of the block size k and the inner loop stride s_j. The stride of the outer loop s_i was varied together with s_j so that no array element was accessed twice. The upper bounds were scaled proportionally to the corresponding strides so that every processor executed 100 iterations of each loop, resulting in total number of 10,000 array accesses per processor.

```
Compute ΔS and ΔM tables
do i = l_i, u_i, s_i
   first_j = i + l_j
   start_j = Local(first_j + ΔS[first_j mod pk])
   end_j = GetUpperBound(i + u_j)
   j = start_j
   offset = j mod k
   while (j ≤ end_j) do
      A(j) = 100.0
      j = j + ΔM[offset]
      offset = j mod k
   endwhile
enddo
```
(a) ΔS and ΔM tables

```
Compute ΔG (length: lenG) table
Compute ΔM table
Compute start_j
countG = 0
do i = l_i, u_i, s_i
   end_j = GetUpperBound(i + u_j)
   j = start_j; offset = j mod k
   while (j ≤ end_j) do
      A(j) = 100.0
      j = j + ΔM[offset]
      offset = j mod k
   endwhile
   start_j = start_j + ΔG[countG]
   countG = (countG + 1) mod lenG
enddo
```
(b) ΔG and ΔM tables

Figure 7.7 Two versions of the SPMD node code for MIV subscripts.

As expected, the performance of run-time resolution is much worse than any of the other address generation methods. The best performance was achieved using the ΔG table in the outer loop and the ΔM table in the inner loop. The method that uses the ΔS table in the outer loop is somewhat less efficient because it requires an additional translation from global to local index space for every iteration of the outer loop, as shown in Figure 7.7(a). In addition, while the ΔG table contains only those entries corresponding to starting location offsets actually visited by the iteration of the outer loop, the size of the ΔS table is always pk (although some entries might be uninitialized). Since in our tests each processor performed 100 iterations of the outer loop, the size of the ΔG could not exceed 100 even for the large block sizes. On the other hand, for $k = 256$, the ΔS table had 8K elements and table lookups had poor locality, resulting in increased performance degradation.

As mentioned earlier, the size of the ΔG table can be pk in the worst case. In the case, the ΔG table memory overhead is unacceptable, methods that generate starting

		ΔG & ΔM	ΔS & ΔM	ΔM Only	Run time
$k = 4$	$s = 3$	4.3	4.8	5.0	1087.8
	$s = 25$	4.5	4.9	5.5	1080.7
	$s = 100$	5.0	5.5	6.0	1080.6
$k = 16$	$s = 3$	4.2	4.8	4.9	1092.8
	$s = 50$	4.3	4.9	5.2	1083.2
	$s = 100$	4.8	5.4	6.0	1081.0
$k = 64$	$s = 3$	4.2	5.1	4.9	1088.3
	$s = 25$	4.4	5.2	5.1	1093.9
	$s = 100$	4.9	5.7	5.7	1083.5
$k = 256$	$s = 3$	4.3	6.5	4.9	1079.3
	$s = 25$	4.4	6.7	5.2	1091.5
	$s = 100$	5.0	7.2	5.7	1094.2

Table 7.2 Execution times in milliseconds for different versions of loops with the MIV subscript.

locations on a demand-driven basis can be applied [61]. Both of the two approaches, the one using the ΔM table (typically much smaller than ΔS or ΔG table) in the inner loop, and the other without any table space overhead, perform only slightly worse than the address generation based on using both the ΔG and ΔM tables, and therefore should be methods of choice if memory overhead needs to be reduced or completely eliminated [61].

7.1.3 Coupled subscripts

All address generation methods presented so far deal only with one-dimensional arrays. Chatterjee et al. have shown that for multidimensional regular array sections (corresponding to array references with independent subscripts), the memory access problem reduces to multiple applications of the algorithm used for the one-dimensional case [18]. However, this is not necessarily true if subscripts are dependent, i.e., if two or more subscript positions contain the same loop induction variables.

```
do i = l_i, u_i, s_i
   do j = l_j, u_j, s_j
      do k = l_k, u_k, s_k
         A(i + j, i + k) = 100.0
      enddo
   enddo
enddo
```

(a) Original loop.

```
Compute ΔM_j and ΔM_k
do i = l_i, u_i, s_i
   Compute start_j and end_j
   j = start_j; offset_j = start_j mod k_1
   while (j ≤ end_j) do
      Compute start_k and end_k
      k = start_k; offset_k = start_k mod k_2
      while (k ≤ end_k) do
         A(j, k) = 100.0
         k = k + ΔM_k[offset_k]
         offset_k = k mod k_2
      endwhile
      j = j + ΔM_j[offset_j]
      offset_j = j mod k_1
   endwhile
enddo
```

(b) SPMD node code.

Figure 7.8 Example program with dependent MIV subscripts.

The example in Figure 7.8(a) shows a three-deep loop nest and a two-dimensional array reference with both subscripts containing induction variable i, the outermost loop induction variable. We assume that array A is distributed with $cyclic(k_1)$ distribution over p_1 processors along its first dimension and with $cyclic(k_2)$ distribution over p_2 processors along its second dimension. Although two subscripts contain a common loop induction variable, as pointed out in Section 7.1.2, the ΔM table for a subscript depends on the stride of the deepest loop only. Therefore, in this example, we can compute two independent tables: ΔM_j for the subscript in the first dimension and ΔM_k for the second dimension. The resulting SPMD code is shown in Figure 7.8(b).

Based on the above observation we define two subscripts to be *coupled* if and only if the deepest loop induction variables occurring in two subscripts are identical. A typical example of a loop with coupled subscripts is shown below.

$$\begin{array}{l}\textbf{do } i = l, u \\ \quad A(s_1 i + c_1, s_2 i + c_2) = 100.0 \\ \textbf{enddo}\end{array}$$

In order to use table lookups for address generation of array references with coupled subscripts, we first show how the method presented by Chatterjee et al. [18] can be extended to find the starting location. Let $l_1 = s_1 l + c_1$ and $l_2 = s_2 l + c_2$ be the values of the two subscripts in the first loop iteration. The starting location for a given processor (m_1, m_2) corresponds to the smallest non-negative integer j which satisfies both of the following two inequalities

$$\begin{array}{l} k_1 m_1 \leq (l_1 + s_1 j) \bmod p_1 k_1 < k_1(m_1 + 1), \text{ and} \\ k_2 m_2 \leq (l_2 + s_2 j) \bmod p_2 k_2 < k_2(m_2 + 1). \end{array}$$

As shown by Chatterjee et al. [18] for the one-dimensional case, finding such a j is equivalent to finding the minimum of the smallest non-negative solutions of the following set of simultaneous linear Diophantine equations

$$\begin{array}{l} \{s_1 j - p_1 k_1 q_1 = i_1 \mid k_1 m_1 - l_1 \leq i_1 \leq k_1 m_1 - l_1 + k_1 - 1\}, \text{ and} \\ \{s_2 j - p_2 k_2 q_2 = i_2 \mid k_2 m_2 - l_2 \leq i_2 \leq k_2 m_2 - l_2 + k_2 - 1\}. \end{array}$$

Let $d_1 = \text{GCD}(s_1, p_1 k_1) = \alpha_1 s_1 - \beta_1 p_1 k_1$ and $d_2 = \text{GCD}(s_2, p_2 k_2) = \alpha_2 s_2 - \beta_2 p_2 k_2$ where $\alpha_1, \beta_1, \alpha_2, \beta_2 \in \mathcal{Z}$. Two separate applications of the extended Euclid algorithm [20] determine d_1, α_1, β_1 and d_2, α_2, β_2. The general solution for equations from the two sets is given by one-parameter families $j = (i_1 \alpha_1 + p_1 k_1 \gamma_1)/d_1$, $q_1 = (-i_1 \beta_1 + s_1 \gamma_1)/d_1$ and $j = (i_2 \alpha_2 + p_2 k_2 \gamma_2)/d_2$, $q_2 = (-i_2 \beta_2 + s_2 \gamma_2)/d_2$, respectively, where $\gamma_1, \gamma_2 \in \mathcal{Z}$. Since we are looking for the solution $j \geq 0$, we must have $\gamma_1 \geq \lceil -i_1 \alpha_1 / p_1 k_1 \rceil$ and $\gamma_2 \geq \lceil -i_2 \alpha_2 / p_2 k_2 \rceil$. The simultaneous solution of the two equations satisfies the equation

$$(i_1\alpha_1 + p_1k_1\gamma_1)/d_1 = j = (i_2\alpha_2 + p_2k_2\gamma_2)/d_2,$$

which is equivalent to

$$d_2p_1k_1\,\gamma_1 - d_1p_2k_2\,\gamma_2 = d_1i_2\alpha_2 - d_2i_1\alpha_1.$$

Let $i = d_1i_2\alpha_2 - d_2i_1\alpha_1$ and $d = \textsc{gcd}(d_2p_1k_1, d_1p_2k_2) = \alpha d_2p_1k_1 + \beta d_1p_2k_2$, $\alpha, \beta \in \mathcal{Z}$. The general solution for γ_1 and γ_2 is given by $\gamma_1 = (i\alpha + d_1p_2k_2\gamma)/d$ and $\gamma_2 = (-i\beta + d_2p_1k_1\gamma)/d$, where $\gamma \in \mathcal{Z}$. The minimal value of parameter γ that satisfies constraints on γ_1 and γ_2 is

$$\gamma = \max\left\{\left\lceil\frac{\lceil\frac{-i_1\alpha_1}{p_1k_1}\rceil d - i\alpha}{d_1p_2k_2}\right\rceil, \left\lceil\frac{\lceil\frac{-i_2\alpha_2}{p_2k_2}\rceil d + i\beta}{d_2p_1k_1}\right\rceil\right\}.$$

We can back-substitute this value to compute γ_1 (or γ_2) and, consequently, the desired solution for j. The minimum of the smallest non-negative solutions for j (across all pairs of equations that have solutions) determines the first array element $A(l_1 + s_1j, l_2 + s_2j)$ accessed by processor (m_1, m_2). The complete algorithm to determine the processor's starting location is shown in Figure 7.9.

Using the same idea as described by Chatterjee et al. [18] for the one-dimensional case, the table of memory gaps can be obtained by first sorting the initial sequence of array accesses and then computing the distances between every two consecutive locations. The ΔM table (analogous to that presented in Section 7.1.1) will be a two-dimensional table, with each entry containing two values, one for each array dimension. Since an access sequence can be of length k_1k_2 in the worst case, table construction based on sorting this sequence has the complexity $O(k_1k_2\log(k_1k_2))$.

The techniques presented in this section can be extended to handle coupled subscripts with multiple index variables in a manner similar to the way the techniques for single index variables, presented in Section 7.1.1, were extended to handle multiple index variables in Section 7.1.2.

Input: Distribution parameters (p_1, k_1), (p_2, k_2), loop parameters (l_1, s_1), (l_2, s_2), and processor number (m_1, m_2).

Output: The starting location $(start_1, start_2)$.

Method:

$min = \infty$
$(d_1, \alpha_1, \beta_1) \leftarrow$ EXTENDED-EUCLID(s_1, p_1k_1)
$(d_2, \alpha_2, \beta_2) \leftarrow$ EXTENDED-EUCLID(s_2, p_2k_2)
$(d, \alpha, \beta) \leftarrow$ EXTENDED-EUCLID$(d_2p_1k_1, d_1p_2k_2)$

do $i_1 = k_1m_1 - l_1,\ k_1m_1 - l_1 + k_1 - 1$
    do $i_2 = k_2m_2 - l_2,\ k_2m_2 - l_2 + k_2 - 1$
        $i = d_1i_2\alpha_2 - d_2i_1\alpha_1$
        if $(i_1 \bmod d_1 = 0$ **and** $i_2 \bmod d_2 = 0$ **and** $i \bmod d = 0)$ **then**
            $\gamma = \max\left\{\left\lceil \frac{\lceil \frac{-i_1\alpha_1}{p_1k_1} \rceil d - i\alpha}{d_1p_2k_2} \right\rceil, \left\lceil \frac{\lceil \frac{-i_2\alpha_2}{p_2k_2} \rceil d + i\beta}{d_2p_1k_1} \right\rceil\right\}$
            $j = (i_1\alpha_1 + p_1k_1(i\alpha + d_1p_2k_2\gamma)/d)/d_1$
            if $(j < min)$ $min = j$
        endif
    enddo
enddo

$(start_1, start_2) = (l_1 + s_1\ min, l_2 + s_2\ min)$

Figure 7.9 Algorithm to compute the starting location for processor (m_1, m_2).

Experimental Results

No previous techniques supported coupled subscripts. To demonstrate the benefits of our approach, we compare the address generation method described here with the run-time resolution technique. Figure 7.10 shows the SPMD code for loops with coupled subscripts.

Table 7.3 contains performance results for our canonical loop example with coupled subscripts on an 8×4 processor grid. In addition to the execution times, for each combination of a block size and a loop stride, we show the number of processors that are actually performing some array accesses. Since accesses due to array references with coupled subscripts are relatively sparse, it is quite likely that not all the processors will participate in the loop execution. Moreover, while we scaled the loop upper bound so that the maximum number of array accesses per processor is 10,000, not all the processors were accessing exactly 10,000 elements.

These facts had a particularly strong influence on the performance of the run-time resolution. It is very clear from Table 7.3 that the run-time resolution performs better as the number of active processors decreases. Furthermore, the performance is not uniform across different values of loop parameters that result in the same number of active processors. This is the consequence of the uneven distribution of array accesses among active processors. While the performance of the method with table lookups is, in essence, proportional to the maximum number of array accesses per processor, the performance of the run-time resolution is proportional to the sum of array accesses of all the processors.

Although in the worst case the ΔM table can be of size $k_1 k_2$ and thus incur significant memory overhead, our experience indicates that this is not very likely to happen in practice. In all our test runs, where block and processor grid sizes were

$$
\begin{array}{l}
\text{Compute } \Delta M \\
\text{Compute } start_1 \text{ and } start_2 \\
\text{Compute } end_1 \text{ and } end_2 \\
i_1 = start_1;\ i_2 = start_2 \\
\mathit{offset}_1 = i_1 \bmod k_1;\ \mathit{offset}_2 = i_2 \bmod k_2 \\
\textbf{while } (i_1 \le end_1 \textbf{ and } i_2 \le end_2) \textbf{ do} \\
\quad A(i_1, i_2) = 100.0 \\
\quad (i_1, i_2) = (i_1, i_2) + \Delta M[\mathit{offset}_1, \mathit{offset}_2] \\
\quad \mathit{offset}_1 = i_1 \bmod k_1 \\
\quad \mathit{offset}_2 = i_2 \bmod k_2 \\
\textbf{endwhile}
\end{array}
$$

Figure 7.10 SPMD node code with coupled subscripts.

always powers of 2 (which is arguably the most common case), the table size never exceeded the maximum of k_1 and k_2.

7.2 Computing communication sets

Consider the following normalized loop with the arrays A and B having $cyclic(k_A)$ and $cyclic(k_B)$ distributions:

$$
\begin{array}{l}
\textbf{do } i = 0,\ u \\
\quad A(i\, s_A + l_A) = f(B(i\, s_B + l_B)) \\
\textbf{enddo}
\end{array}
$$

Chatterjee et al. show that for each reference, the sequence of array elements accessed by any given processor can be computed using a table of local memory gaps whose size does not exceed the block size of the array's distribution [18]. Using the algorithms described in previous sections and in the related work [62], we can construct the sequences of accessed array elements in both the local and the global index space (Local and Global, respectively), as well as their corresponding iteration numbers (Iter).

$k_1 \times k_2$	(s_1, s_2)	Active procs	ΔM table	Run time
4×4	$(1,1)$	8	6.6	337.6
	$(1,3)$	24	6.6	639.8
	$(10,5)$	8	6.4	333.6
4×8	$(1,1)$	8	6.5	337.5
	$(1,3)$	16	6.5	438.8
	$(10,5)$	16	6.4	629.7
8×8	$(1,1)$	8	6.7	337.0
	$(1,3)$	24	6.7	838.3
	$(10,5)$	8	6.6	334.8
8×16	$(1,1)$	8	7.1	335.4
	$(1,3)$	16	6.8	835.8
	$(10,5)$	16	6.7	631.9
16×16	$(1,1)$	8	7.3	335.4
	$(1,3)$	24	7.2	835.8
	$(10,5)$	8	7.2	334.3

Table 7.3 Execution times in milliseconds for different versions of loops with coupled subscripts.

As an example, we consider the arrays A and B distributed onto $p = 3$ processors with the block sizes $k_A = 4$ and $k_B = 8$. In Figure 7.11 we show the elements of array A that are accessed due to the reference $A(5i)$ ($s_A = 5$, $l_A = 0$). Similarly, Figure 7.12 depicts the access pattern for the reference $B(7i+2)$ ($s_B = 7$, $l_B = 2$).

In order to compute the sets of data that processor 0 must send to other processors, we first determine the sequence of accesses to array B (in the global index space) that processor 0 owns, $\text{GLOBAL}_B(0) = [2, 30, 51, 72, 79, 100, 121, 149]$ with the period 168, and the corresponding iteration sequence $\text{ITER}_B(0) = [0, 4, 7, 10, 11, 14, 17, 21]$ with the period 24. After multiplying the iteration numbers by the stride s_A, we get the global index sequence of corresponding left-hand-side (lhs) accesses, $\text{IMAGE}_{lhs}(0) = [0, 20, 35, 50, 55, 70, 85, 105]$ with the period 120. Assuming that the computation is partitioned using the owner-computes rule, the sequence of processors that processor 0 sends data to is $\text{PSEND}(0) = [0, 2, 2, 0, 1, 2, 0, 2]$. (For any index j in the GLOBAL_B on

Processor 0				Processor 1				Processor 2			
$\boxed{0}$	1	2	3	4	$\boxed{5}$	6	7	8	9	$\boxed{10}$	11
12	13	14	$\boxed{15}$	16	17	18	19	$\boxed{20}$	21	22	23
24	$\boxed{25}$	26	27	28	29	$\boxed{30}$	31	32	33	34	$\boxed{35}$
36	37	38	39	$\boxed{40}$	41	42	43	44	$\boxed{45}$	46	47
48	49	$\boxed{50}$	51	52	53	54	$\boxed{55}$	56	57	58	59
$\boxed{60}$	61	62	63	64	$\boxed{65}$	66	67	68	69	$\boxed{70}$	71
72	73	74	$\boxed{75}$	76	77	78	79	$\boxed{80}$	81	82	83
84	$\boxed{85}$	86	87	88	89	$\boxed{90}$	91	92	93	94	$\boxed{95}$

Figure 7.11 Elements of array A with $cyclic(4)$ distribution on 3 processors. The squares mark the array elements accessed by $A(5i)$.

a given processor, the corresponding entry in PSEND is $Owner_A(((j - l_B)/s_B) \times s_A + l_A)$, where $Owner_A(x) = (x \text{ div } k_A) \bmod p$.) For example, processor 0 owns the elements $B(30)$, $B(51)$, $B(100)$, and $B(149)$ referenced by processor 2, and the set of data that processor 0 needs to send to processor 2 is $\text{DSEND}(0 \to 2) = B[30, 51, 100, 149]$ with the period 168, which corresponds to $B[30, 51, 100, 149, 198, 219, \ldots]$.

The sets of data that processor 0 must receive from other processors are computed similarly; we build the table of global indices for accesses made by processor 0, $\text{GLOBAL}_A(0) = [0, 15, 25, 50]$ with the period 60 and $\text{ITER}_A(0) = [0, 3, 5, 10]$ with the period 12. Since the iteration periods of ITER_A and ITER_B are differ-

Processor 0								Processor 1								Processor 2							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167
168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191

Figure 7.12 Elements of array B with $cyclic(8)$ distribution on 3 processors. The circles mark the array elements accessed by $B(7i+2)$ $(i \geq 0)$.

ent, we expand the $\text{I}\textsc{ter}_A$ sequence so that its period is equal to the period of $\text{I}\textsc{ter}_B$. (In general, it may be required to expand both sequences so that the resulting iteration period is the least common multiple of $pk_A/\text{GCD}(pk_A, s_A)$ and $pk_B/\text{GCD}(pk_B, s_B)$, but in practice we do not expect the length of the expanded sequence to exceed the larger of the two block sizes.) After expansion, we get $\text{I}\textsc{ter}_A(0) = [0, 3, 5, 10, 12, 15, 17, 22]$ with the period 24, and the sequence of corresponding rhs accesses is $\text{I}\textsc{mage}_{rhs}(0) = [2, 23, 37, 72, 86, 107, 121, 156]$ with the period 168. By computing the owners of the elements of array B with these indices we can construct the table $\text{PR}\textsc{ecv}(0)$ = [0,2,1,0,1,1,0,1], as well as $\text{DR}\textsc{ecv}$ sets, for example, $\text{DR}\textsc{ecv}(0 \leftarrow 1) = B[37, 86, 107, 156]$ with the period 168.

Not all the tables shown here need to be actually allocated and assigned values. We need tables for storing the array access sequences corresponding to both the reference. The only table essential for packing outgoing messages is $\text{PS}\textsc{end}$ since it eliminates the need for computing the owner processor for every array reference. In addition, this table can be used to compute the lengths of the $\text{DS}\textsc{end}$ sequences. Similarly, the $\text{PR}\textsc{ecv}$ table is needed to generate the efficient code at the receiving end and find the lengths of the $\text{DR}\textsc{ecv}$ sequences.

7.3 Code generation

When generating communication for irregular problems, Das et al. allocate buffers for the non-local data immediately following the space for local array elements and translate the off-processor references to point to buffer addresses [22]. Using a similar idea, we allocate the space for the local and non-local rhs references contiguously, but we have a separate, appropriately sized buffer for each processor in $\text{PR}\textsc{ecv} \cup \text{PS}\textsc{end}$. Each overflow segment is used to both pack the outgoing message and receive the incoming message from the corresponding processor. If this imposes unacceptable constraints on the communication schedule, we can pack messages into temporary buffers and exploit the overflow segments only in the receiving phase. The size of each overflow area can be computed using the length and the period of the DRECV (DSEND) sequence and the loop upper bound. If the necessary information is not available at compile time, the compiler can try to estimate the required space and let the run-time system resize the segments if needed.

Chatterjee et al. show that only one pass over the locally owned elements of array B is needed to build all outgoing messages [18]. However, an ownership computation involving integer divisions is performed for each array access, in order to determine the destination processors for the accessed elements. We reduce the number of required computations to the length of the access cycle for the *rhs* reference by using the PSEND table when packing messages. In the code for processor m shown below, *buffer* is an array of pointers to overflow segments for all destination processors. Processor m scans the elements of B it owns using the *NextIndex* function, which can be implemented using either the table of the local memory gaps between consecutive array accesses [18] or our demand-driven address generation method [61].

$index_B = start_B$; $count_P = 0$
while $(index_B < end_B)$ **do**
  if $(\text{PSEND}[count_P] \neq m)$ **then**
    $*(buffer[\text{PSEND}[count_P]])\text{++} = B[index_B]$
  endif
  $count_P = (count_P + 1) \bmod Length(\text{PSEND})$
  $index_B = NextIndex(index_B)$
endwhile

Although expensive integer divisions are eliminated (the mod operation is shown only for simplicity and would, in practice, be replaced by a simple wrap-around test), the message-packing code can be further improved. Firstly, when computing the access sequence for the *rhs* reference, we can separate the elements that are to be sent to other processors from those that are used locally; in this way the PSEND table for processor m can be compressed to contain only the entries that are different from m, which, in turn, allows us to eliminate the **if** statement test from the loop. Secondly, by strip-mining the loop based on the length of the PSEND sequence, we can significantly reduce the number of the **while** loop tests, since they will be necessary only in the epilog of the strip-mined loop.

After all the messages are packed, the interprocessor communication takes place. We do not consider the issue of communication scheduling, but instead focus on the code generation at the receiving end. Contiguous memory allocation for non-local array elements allows each processor to receive data directly into the appropriate overflow segment, thereby eliminating the overhead of unpacking messages. However, in order to preserve the original execution order, the non-local references need to be converted to access the appropriate memory locations in the overflow region.

The repetitive access pattern is exploited once again for efficient mapping function generation. Since the local and non-local elements of B are stored contiguously, we can construct the $\Delta\mathrm{M}_B$ table that contains the memory gaps between consecutive *rhs* accesses (this table is of the same size as PRECV). Its entries are computed in a linear scan of the IMAGE_{rhs} sequence by taking into account the sizes of the local array area and the non-local overflow segments.

Since the lengths of the DRECV sequences corresponding to different processors may vary, the memory gap between two fixed consecutive accesses in the *rhs* sequence is not necessarily constant across different cycles; in the example shown in Figure 7.13, the gap between $B(2)$ and $B(23)$ is 70, while the gap between the corresponding accesses in the next cycle, $B(170)$ and $B(191)$, is 15. The *correction* to the access gaps is equal to the difference between the local cycle lengths correspond-

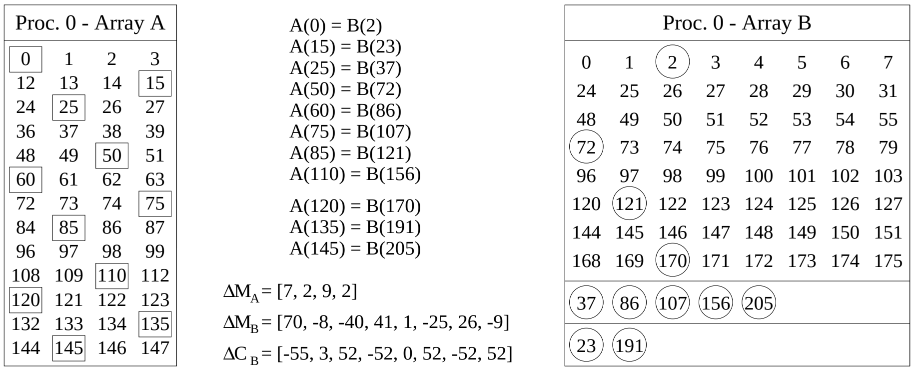


Figure 7.13 Layout of array elements after communication and assignments performed by processor 0 for the statement $A(5i) = B(7i+2)$.

ing to the consecutive accesses, 1 for $B(23)$ and 56 for $B(2)$ (the local memory gap between $B(2)$ and $B(170)$, which starts the next cycle, is 56, while the cycle length for elements received from processor 2 is 1). These gap corrections are stored in the ΔC_B table and applied at every *rhs* access.

$$
\begin{array}{l}
index_A = start_A;\ index_B = start_B;\ i_A = 0;\ i_B = 0 \\
\textbf{while}\ (index_A < end_A)\ \textbf{do} \\
\quad A(index_A) = f(B(index_B)) \\
\quad index_A = index_A + \Delta M_A[i_A] \\
\quad i_A = (i_A + 1) \bmod \mathit{Length}(\Delta M_A) \\
\quad index_B = index_B + \Delta M_B[i_B] \\
\quad \Delta M_B[i_B] = \Delta M_B[i_B] + \Delta C_B[i_B] \\
\quad i_B = (i_B + 1) \bmod \mathit{Length}(\Delta M_B) \\
\textbf{endwhile}
\end{array}
$$

In the loop shown above, which each processor executes after receiving all the messages, the original non-local references are converted to accesses into the overflow region where the received data is stored. Since we assume the *owner computes* rule, the *lhs* accesses are traversed using the ΔM_A table of local memory gaps, as described by Chatterjee et al. In order to simplify the code, the access gap tables can be expanded to be of the same size, in which case a single index is sufficient to traverse all three tables. If this is done, then the strip-mining, as described for the message-packing loop, is also applicable and will result in the more efficient code.

7.4 Communication optimization: message coalescing

It is not unusual for assignment statements to have multiple *rhs* terms referencing the same array, as in

$$\begin{array}{l} \textbf{do } i = 0,\ u \\ \quad A(5i) = f(B(7i+2), B(7i+5)) \\ \textbf{enddo} \end{array}$$

Although it is possible to repeat the communication step for each term, significant savings in communication time can be achieved by grouping the data corresponding to different references so that each processor sends only one message to any other processor. Neither Chatterjee et al. [18] nor Gupta et al. [41] deal with this issue. Stichnoth [80] indicates that combining communication steps would be profitable, but he does not describe the necessary analysis. In contrast, we show how our approach can support message coalescing optimization [48] to reduce the communication cost.

In order be able to pack the messages corresponding to different *rhs* terms in a single pass over the array, we must find the union of PSEND sequences corresponding to different *rhs* references. Section 7.2 showed that $\text{GLOBAL}_{B(7i+2)}(0) = [2, 30, 51, 72, 79, 100, 121, 149]$ with the period 168 and $\text{PSEND}_{B(7i+2)}(0)$, the sequence of processors that processor 0 sends data to, to be $[0, 2, 2, 0, 1, 2, 0, 2]$. Similarly, for $B(7i + 5)$ we can compute $\text{GLOBAL}_{B(7i+5)}(0) = [5, 26, 54, 75, 96, 103, 124, 145]$ with the period 168, which results in $\text{PSEND}_{B(7i+5)}(0) = [0, 0, 2, 0, 1, 2, 0, 2]$. By merging the two GLOBAL sequences we get $\text{GLOBAL}_B = [2, 5, 26, 30, \ldots, 124, 145, 149]$ with the period 168, and the unioned $\text{PSEND}_B(0) = [0, 0, 0, 2, 2, 2, 0, 0, 1, 1, 2, 2, 0, 0, 2, 2]$.

From this we can compute the sets of elements that a processor needs to send for both rhs references, for example, $\text{DS}\textsc{end}_B(0 \rightarrow 2) = [30, 51, 54, 100, 103, 145, 149]$ with the period 168. Using the $\text{PS}\textsc{end}_B$ set, the messages can be packed in a single scan over the array B, as described in Section 7.3. This scheme interleaves the elements corresponding to the two rhs references in the packed message and allows the possibility of reducing message sizes by eliminating multiple occurrences of an element in the $\text{G}\textsc{lobal}_B$ sequence.

However, before merging the two GLOBAL sequences, we may have to replicate them so that they have the same period. If the strides of the two rhs references are s_{B_1} and s_{B_2}, then the sequence period after expansion is the least common multiple of $pk_B s_{B_1}/\text{GCD}(pk_B, s_{B_1})$ and $pk_B s_{B_2}/\text{GCD}(pk_B, s_{B_2})$, and based on our experiments, the increased sequence length significantly increases the overhead. Even without the sequence replication, we can pack the messages in a single pass by effectively merging the GLOBAL sequences on the fly, but the control overhead associated with merging has to be incurred for each element packed in the message.

An alternative to interleaving of array elements from two messages into a single coalesced message is to pack the messages for different rhs references in separate areas of the same contiguous buffer. Although this precludes the possibility of eliminating redundancies, we choose this method because it keeps the construction of access gap tables for the rhs references simple. Since the messages are packed in a contiguous buffer, we still reduce the number of the messages by sending a single message for multiple rhs references. Moreover, as described in Section 7.3, we allocate contiguous overflow regions to directly receive the message and avoid the unpacking overhead.

For code generation we construct an access gap table for each rhs reference. Since the non-local array elements for two references are received and buffered in separate overflow regions, the access gap tables (ΔM_{B_1} and ΔM_{B_2}) and the correction tables

(ΔC_{B_1} and ΔC_{B_2}) for the two *rhs* references can be computed in exactly the same way as described in Section 7.3.

7.5 Experimental results

In order to validate the previously described methods we implemented a prototype system for computing communication sets and generating code for array assignment statements. This allowed us to evaluate the overheads incurred by our techniques, as well as compare them with other existing methods, in particular the virtual processor approach by Gupta et al. [41]. Their solution is based on treating a *block-cyclic* distribution as either a *block* or *cyclic* distribution of data onto virtual processors, and mapping of the virtual processors to the physical processors in either a *cyclic* or *block* fashion. Depending on how arrays are distributed on the virtual processors, the scheme is called either virtual-block or virtual-cyclic, and each array can be visualized under either of the two views.

Table 7.4 gives the table generation, message packing, communication, and loop execution times for 160,000 iterations of the normalized loop described in Section 7.2 for our method and the virtual processor approach. (The code that implements the virtual processor scheme was provided by S.D. Kaushik [54].) Once again, the experiments were performed on 32 processors of an Intel iPSC/860 hypercube, using the *icc* compiler with –O4 optimization level and *dclock* timer. The parameters l_A and l_B were kept constant at 0 since the lower bounds have little effect on the execution times, while the parameters k_A, k_B, s_A, and s_B were varied to compare our technique against all possible virtual views adopted by the virtual processor approach. Reported times, shown in milliseconds, represent maximums over all 32 processors.

Since we used the identical complete exchange algorithm as in the virtual processor implementation, the communication times were similar for both schemes and

k_A	k_B	s_A	s_B	Table Generation		Message Packing		Loop Execution		Comm.	View
				Rice	OSU	Rice	OSU	Rice	OSU		
64	256	3	11	3.70	47.63	4.04	5.04	2.51	2.93	30.11	vb-vb
		5	5	3.69	59.18	3.32	4.04	2.77	2.68	43.64	vb-vb
		7	3	3.70	72.71	2.78	4.19	2.42	3.52	26.84	vb-vb
1000	5	1	1	11.43	89.86	2.61	4.56	2.98	3.74	25.02	vb-vc
		1	3	11.51	91.83	3.06	5.12	3.06	3.76	24.75	vb-vc
		1	7	11.45	93.10	3.82	6.12	3.11	3.82	24.95	vb-vc
16	15	2	3	0.89	8.86	2.62	3.08	2.32	1.68	31.57	vc-vc
		3	5	0.95	11.57	3.04	3.78	2.37	1.69	29.36	vc-vc
		4	4	1.07	10.98	2.99	3.73	2.33	1.73	27.56	vc-vc
4	1200	1	1	13.42	56.25	3.26	4.28	3.17	3.45	30.35	vc-vb
		3	2	7.29	160.68	2.86	6.40	2.78	5.42	26.06	vc-vb
		5	3	5.25	313.03	2.91	8.02	2.66	7.41	25.53	vc-vb

Table 7.4 Breakdown of execution times in milliseconds for the array assignment $A(0 : 160000s_A - 1 : s_A) = B(0 : 160000s_B - 1 : s_B)$ obtained on a 32 processor Intel iPSC/860 hypercube using our method (Rice) and the virtual processor approach (OSU).

we report them only for our method. The execution times for the virtual processor approach correspond to the best visualization view selected separately for each parameter set. The last column shows the virtual views for the *rhs* and *lhs* arrays that were chosen to minimize the estimated indexing overhead, where vb and vc correspond to virtual-block and virtual-cyclic.

Table 7.4 shows that the table construction overhead of our technique is consistently and significantly smaller than that incurred by the virtual processor approach. We construct all the tables in time linearly proportional to their sizes and take advantage of the repetitiveness of array accesses. While Gupta et al. derive closed-form expressions for the communication sets of virtual processors, the mapping of these sets onto physical processors can incur a substantial overhead.

Although our table generation is in most cases at least one order of magnitude faster than in the virtual processor scheme, with the increase in the block size the time needed to construct the tables may represent a sizable overhead relative to the time spent in the interprocessor communication. However, the table generation time is independent of the number of iterations executed in the loop, and thus, for larger data sets the overhead would be less significant.

In the virtual processor scheme, each processor performs a separate pass over its portion of the *rhs* array for each message that it must build. With our approach, all outgoing array elements are packed into corresponding messages in a single pass over the array, which results in the shorter message packing time. During the execution of loop iterations in the virtual processor approach non-local *rhs* array elements are simply unpacked from the receive buffers. In contrast, we do not need to unpack the received messages, but our approach requires table lookups in the loop. Since the overheads due to indirect addressing are small [62], the loop execution in our scheme is faster than with the virtual processor method, except in the case when virtual-cyclic view was selected at both the sending and the receiving side.

7.6 Related work

Several efforts to address some of the difficulties in compiling programs with $cyclic(k)$ distribution have been described in the literature. We have extensively cited and described two of the previous works in the chapter: the virtual-block scheme proposed by Gupta et al. [41] and the approach described by Chatterjee et al. [18].

Ancourt et al. use a linear algebra framework for compiling *independent* loops in HPF [5]. Although they can handle arbitrary affine array subscripts, the generated loop bounds and local array subscripts can be quite complex, and thus introduce a significant overhead. Furthermore, the assumption of independent parallelism allows them to enumerate loop iterations in any order, which is, in general, not always possible. Gupta et al. address the problem of array statements involving block-cyclic distributions. In their virtual-cyclic scheme, array elements are accessed in an order different from the order in a sequential program. In the virtual-block scheme array accesses are not reordered, but if the array section stride is larger than the block size, their method effectively reduces to the run-time address resolution. Stichnoth et al. use intersections of array slices for communication generation [81]. Their approach is similar to, and has the same drawback as, the virtual-cyclic scheme mentioned above.

7.7 Summary

The chapter described efficient techniques for generating local addresses for array references with arbitrary affine subscripts. It presented the first linear-time algorithm developed to compute access gap sequence for array references with *cyclic(k)* distributions. However, the algorithm presented in the chapter works only when the access stride is less than the block size of the *cyclic(k)* distribution. The chapter presented extensions to the periodic access sequences to demonstrate that they can be used for references with multiple loop index variables and coupled array subscripts.

The chapter described a general and efficient communication generation approach for $cyclic(k)$ distributions and showed how it can support the message coalescing optimization. It also presented the experimental results to show that the low overheads incurred by our method compare favorably to those of an already existing technique, the virtual processor approach. The generality of our technique and the efficiency with which different subscripts are handled, as demonstrated by our extensive experimental results, make it suitable for inclusion in compilers and run-time systems for HPF-like languages.

Chapter 8

Conclusions

This dissertation describes machine-independent communication generation techniques for data-parallel programs. The techniques presented in the dissertation tackle the issues related to communication analysis, placement, and optimization. These techniques were implemented in the Fortran D95 compiler, the research compiler being developed at the Rice University. In addition, the dissertation describes efficient compilation techniques to support *cyclic(k)* distribution, the most general regular distribution supported by High Performance Fortran (HPF). It presents extensive experimental results to prove the efficacy of techniques presented in the dissertation.

The dissertation illustrates the importance of taking resource constraints into account during communication generation. Previous compilation techniques ignored most machine-dependent resource constraints in order to simplify the related analysis. However, as illustrated in this work, ignoring resource constraints can lead to incorrect communication placement. Thus, resource constraints such as in-core memory size *must* be taken into account during communication generation.

The dissertation presents a data-flow analysis framework to incorporate resource constraints during communication placement and optimization. It describes an efficient data-flow algorithm that hoists communication to the least frequently executed program location(s), eliminates redundant communication, and achieves latency hiding by maximizing the separation between SEND and RECV primitives. In addition, it guarantees balanced placement; that is, it ensures that every program execution path contains matching number of SEND and RECV communication primitives. As explained in the dissertation, the balanced communication placement, together with the careful propagation of information related to the processors involved in communication, helps the Fortran D95 compiler to provide correct communication placement. The dissertation presents proofs to demonstrate the correctness of the communication placement in the Fortran D95 compiler.

The dissertation describes the use of dependence information in simplifying data-flow equations for communication placement and optimizations. The data-flow algorithm presented in the dissertation consists of uni-directional analysis, with an independent set of data-flow equation system for each placement criterion. The modularity and simplicity of the framework allows placement criteria to be modified easily and, thus, makes the framework applicable to a wide variety of problems. For example, it can be adapted to problems that don't impose balance placement criterion by merely dropping the appropriate set of data-flow equations. In addition to the placement of communication primitives, the resource-based placement framework is used to determine the placement of memory allocation and deallocation primitives in the Fortran D95 compiler. (Memory primitives are used to manage the memory required by message buffers.)

The dissertation presents efficient compilation techniques for *cyclic(k)* distributions. It shows that the array accesses can be specified by a sequence by length k, together with the period of the sequence. It describes a representation for expressing and manipulating *cyclic(k)* sets efficiently. It describes novel techniques to compute access sequences for references with multiple index variables and coupled subscripts. It exploits the repetitive access pattern exhibited by *cyclic(k)* references to determine communication sets precisely. In addition, it describes innovative techniques to compute subscripts for non-local references. It also describes the use of overflow areas as message buffers for *cyclic(k)* distributions.

This dissertation raises some compilation issues that deserve further investigation. Although the resource-based framework was presented in the context of communication placement, it can be adapted for other memory hierarchy related code placement problems that involve resource constraints. Besides placement, the framework can also be used to select appropriate prefetch or block-transfer primitives for data movement on distributed shared-memory machines. The effects of the cache size constraint and the limited number of outstanding prefetches remain to be investigated. Another potential application of the framework includes placement of I/O primitives for programs involving out-of-core I/O. For the cases where our framework does not completely hide the latency of data movement, other techniques and transformations need to be developed to maximize the communication and computation overlap. Moreover, better reuse analysis techniques need to be developed to fully exploit the buffer constraint.

The dissertation presented a linear-time algorithm for generating local access sequence for $cyclic(k)$ distributions for the special cases that occur frequently. However, a linear-time algorithm for the general case will simplify handling of $cyclic(k)$ distributions [62]. Moreover, in certain cases, the overhead of maintaining tables might not be acceptable, in which case techniques need to be developed to support $cyclic(k)$ distribution without incurring the space overhead. In addition, the techniques presented in the dissertation use overflow regions irrespective of the communication pattern. It has been shown that for *block* and *cyclic* distributions, overlap areas provide improved performance by simplifying the non-local address calculation. Similar techniques can be developed to support shift communication pattern efficiently in presence of *cyclic(k)* distributions [60]. Finally, regarding code generation, various buffering schemes for non-local data and subscript translation need to be compared to evaluate relative merits of different schemes.

Bibliography

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

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

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

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

[5] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[6] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[7] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[8] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[9] P. Banerjee, J. Chandy, M. Gupta, J.G.Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory message passing multicomputers. In *First International Workshop on Parallel Processing*, Banglore, India, December 1994.

[10] D. Bell and J. Grimson. *Distributed Database Systems*. Addison-Wesley Publishing Company, Great Britain, 1992.

[11] S. Benkner, P. Brezany, and H. Zima. Processing array statements and procedure interfaces in the PREPARE High Performance Fortran compiler. In *Compiler Conference*, Edinburgh, April 1994.

[12] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Santa Barbara, CA, July 1995.

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

[14] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling high performance fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 704–709, San Francisco, CA, February 1995.

[15] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

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

[17] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran — A Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

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

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

[20] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[21] CONVEX Computer Corporation. *Exemplar Architecture*. CONVEX Press, Richardson, Texas, first edition, 1993.

[22] R. Das, M. Uysal, J. Saltz, and Y-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. Technical Report CS-TR-3163, Dept. of Computer Science, Univ. of Maryland, College Park, October 1993.

[23] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[24] D.M.Dhamdhere. A fast algorithm for code movemenet optimization. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[25] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 372–379, Williamsburg, VA, April 1992.

[26] J. Dongarra and D. Walker. MPI: The Message Passing Interface standard. *International Journal of Supercomputing Applications*, 8(3):151–180, 1994.

[27] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoise's "global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[28] F.E.Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[29] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, September 1988.

[30] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

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

[32] M. Garey and D. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY, 1979.

[33] M. Gerndt. Array distribution in SUPERB. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[34] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

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

[36] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[37] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[38] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K-Y. Wang, W-M. Ching, and T. Hgo. An HPF compiler for the ibm sp2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[39] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

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

[41] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report OSU-CISRC-4/94-TR19, Department of Computer and Information Science, The Ohio State University, April 1994.

[42] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[43] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Dept. of Computer Science, Rice University, December 1994.

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

[45] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[46] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[47] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, July 1994.

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

[49] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[50] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[51] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[52] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[53] J.Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, 1970.

[54] S. D. Kaushik. Private communication, April 1995.

[55] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.

[56] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical Report Version 0.95, University of Maryland, June 1995.

[57] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[58] K. Kennedy. A global flow analysis algorithm. *International Journal of Computer Mathematics*, 3:5–15, 1971.

[59] K. Kennedy and N. Nedeljković. Combining dependence and data-flow analyses to optimize communication. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[60] K. Kennedy, N. Nedeljković, and A. Sethi. Communication generation for *cyclic*(k) distributions. In *Third Annual Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR'95)*, Troy, N.Y., May 1995.

[61] K. Kennedy, N. Nedeljković, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[62] K. Kennedy, N. Nedeljković, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[63] K. Kennedy and A. Sethi. A communication placement framework with unified dependence and data-flow analysis. In *Proceedings of the Third International Conference on High Performance Computing*, Trivandrum, December 1996.

[64] K. Kennedy and A. Sethi. Resource-based communication placement analysis. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[65] A. Knies, M. O'Keefe, and T. MacDonald. High Performance Fortran: A practical analysis. *Scientific Programming*, 3(3):187–199, Fall 1994.

[66] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[67] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[68] C. Koelbel. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[69] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[70] C. Koelbel and P. Mehrotra. Compiling global name-space programs for distributed execution. ICASE Report 90-70, Institute for Computer Application in Science and Engineering, Hampton, VA, October 1990.

[71] U. Kremer and M. Ramé. Compositional oil reservoir simulation in Fortran D: A feasibility study on Intel iPSC/860. *International Journal of Supercomputer Applications and High Performance Computing*, 8(2), Summer 1994.

[72] J. Li and M. Chen. Automating the coordination of interprocessor communication. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[73] B. Lu. Compiling reductions in the Fortran D95 compiler. M.S. Thesis (under preparation), October 1996.

[74] D. Maydan, S. Amarasinghe, and M. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Charleston, SC, January 1993.

[75] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of DSM systems via compiler involvement. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[76] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[77] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[78] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[79] B. Rosen, M. Wegman, and K. Zadeck. Global value numbers and redundant compuations. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[80] J. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[81] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[82] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[83] E. Su, D. Palermo, and P. Banerjee. Processor tagged descriptors: A data structure for compiling for distributed-memory multicomputers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Montreal, Canada, August 1994.

[84] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[85] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[86] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.