COMPILER TECHNIQUES FOR OPTIMIZING COMMUNICATION AND DATA DISTRIBUTION FOR DISTRIBUTED-MEMORY MULTICOMPUTERS

BY

DANIEL JOSEPH PALERMO

B.S., Purdue University, 1990
M.S., University of Southern California, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

© Copyright by Daniel Joseph Palermo, 1996

COMPILER TECHNIQUES FOR
OPTIMIZING COMMUNICATION AND DATA DISTRIBUTION
FOR DISTRIBUTED-MEMORY MULTICOMPUTERS

Daniel Joseph Palermo, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Prithviraj Banerjee, Advisor

Distributed-memory multicomputers, such as the Intel iPSC/860, the Intel Paragon, the IBM SP-1 /SP-2, the NCUBE/2, and the Thinking Machines CM-5, offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. However, lacking a global address space, they present a very difficult programming model in which the user must specify how data and computation are to be distributed across processors and determine those sections of data to be communicated to specific processors. To overcome this difficulty, significant research effort has been aimed at source-to-source parallelizing compilers for multicomputers that relieve the programmer from the task of program partitioning and communication generation, while the specification of data distributions has remained a responsibility of the programmer.

The quality of the data distribution for a given application is crucial to obtaining high performance on distributed-memory multicomputers. By selecting an appropriate data distribution, the amount of communication required by an application can be dramatically reduced. The resulting performance using a given data distribution therefore depends on how well the compiler can optimize the remaining communication. In this thesis, we present and analyze several techniques to optimize communication and, based on the performance of these optimizations, automatically select the best data partitioning for a given application.

Previous work in the area of optimizing data distribution used constraints based on performance estimates (which model the communication optimizations) to select high quality data distributions which remain in effect for the entire execution of an application. For complex programs, however, such static data distributions may be insufficient to obtain acceptable performance. The selection of distributions that dynamically change over the course of a program's execution (taking into account the added overhead of performing redistribution) adds another dimension to the data partitioning problem. In this thesis, we present a technique that extends the

static data partitioning algorithm to automatically determine those distributions most beneficial over specific sections of a program while taking into account the added overhead of performing redistribution. Finally, we also present an interprocedural data-flow framework that performs the conversion between a distribution-based representation (as specified by the data partitioner and required for compilation) and a redistribution-based form (as specified by HPF) while optimizing the amount of redistribution that must be performed.

These techniques have been implemented as part of the PARADIGM (PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers) project at the University of Illinois. The complete system strives to provide a fully automated means to parallelize sequential programs obtaining high performance on a wide range of distributed-memory multicomputers.

To my wife and my parents,
without whose love and support
none of this would have been possible.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for his constant support and advice throughout the course of this research. I would also like to thank the members of my committee, Professors Wen-Mei Hwu, David Padua, and Constantine Polychronopoulos, for their valuable comments and suggestions on my work and this thesis.

I would like to thank all of the members of both the PARADIGM and ProperCAD projects for their technical expertise as well as their camaraderie. I would especially like to thank John Chandy, Eugene Hodges, and Ernesto Su, who have all made significant contributions during the course of this research, Amber Roy-Chowdhury and Antonio Lain for their constant feedback on the internal design of various aspects of the compiler, as well as Shankar Ramaswamy for teaching us all that "talk is research." Additionally, I would also like to thank my good friend, Eugene Chen from the Solid State Devices Laboratory, for making me feel at home in Urbana-Champaign from the first moment I arrived.

For their support throughout the course of this research, I would like to thank the National Aeronautics and Space Administration and the Advanced Research Projects Agency. I would also like to thank the National Center for Supercomputing Applications, the San Diego Supercomputing Center, and the Argonne National Laboratory for providing access to their computing facilities.

Most of all I would like to thank my entire family for all of their love, encouragement, patience, and support throughout my studies. Most importantly, I also give thanks to God for the abilities and inspiration which He has given me.

TABLE OF CONTENTS

CHAPTER **PAGE**

1 INTRODUCTION **1**
1.1 Overview of High Performance Fortran 2
1.2 Overview of the PARADIGM Project 5
1.2.1 Program analysis 5
1.2.2 Automatic data partitioning 6
1.2.3 Compiling regular computations 6
1.2.4 Generic library interface 7
1.3 Thesis Contributions 8
1.4 Thesis Organization 9

2 RELATED WORK **11**
2.1 Optimizing Communication 11
2.2 Optimizing Data Distribution 13
2.2.1 Static partitioning 13
2.2.2 Dynamic partitioning 14
2.3 Optimizing Data Redistribution 18
2.4 Summary 19

3 OPTIMIZING COMMUNICATION **20**
3.1 Message Coalescing 21
3.2 Message Vectorization 22
3.3 Message Aggregation 24
3.4 Message Pipelining 26
3.4.1 Pipeline performance analysis 29
3.4.2 Serial computation cost estimation 33
3.5 Summary 34

4 OPTIMIZING DATA DISTRIBUTION **35**
4.1 Static Performance Estimation 35
4.2 Static Distribution Selection 37
4.2.1 Alignment 38
4.2.2 Block/cyclic distribution 38
4.2.3 Block size selection 38

4.2.4 Mesh configuration . . . 39
4.3 Dynamic Distribution Selection . . . 39
4.3.1 Motivation for dynamic distributions . . . 39
4.3.2 Overview of the dynamic distribution approach . . . 41
4.3.3 Phase decomposition . . . 43
4.3.4 Phase and phase transition selection . . . 49
4.3.5 Hierarchical Component Affinity Graph (HCAG) . . . 55
4.4 Summary . . . 58

5 OPTIMIZING DATA REDISTRIBUTION . . . 59
5.1 Data Redistribution Analysis . . . 61
5.1.1 Computing reaching distributions . . . 63
5.1.2 Constructing the distribution flow graph . . . 65
5.1.3 Representing distribution sets . . . 66
5.2 Interprocedural Data Redistribution Analysis . . . 68
5.2.1 Distribution synthesis . . . 70
5.2.2 Redistribution synthesis . . . 73
5.2.3 Static Distribution Assignment (SDA) . . . 79
5.3 Implementation . . . 83
5.3.1 Source level data flow . . . 84
5.3.2 Virtual clones . . . 85
5.3.3 Array reshaping . . . 85
5.3.4 Examples . . . 86
5.4 Summary . . . 91

6 EXPERIMENTAL RESULTS . . . 92
6.1 Optimizing Communication . . . 92
6.1.1 Results of run-time resolution . . . 95
6.1.2 Results of message coalescing and loop bounds reduction . . . 96
6.1.3 Results of message vectorization . . . 96
6.1.4 Results of message aggregation . . . 97
6.1.5 Results of all non-pipelined optimizations . . . 98
6.1.6 Results of message pipelining . . . 100
6.2 Optimizing Data Distribution and Redistribution . . . 107
6.2.1 2-D Alternating Direction Implicit (ADI2D) iterative method . . . 107
6.2.2 Shallow water weather prediction benchmark . . . 110
6.3 Summary . . . 114

7 CONCLUSIONS . . . 115
7.1 Summary of Contributions . . . 115
7.2 Future Work . . . 117
7.2.1 Optimizing communication . . . 118
7.2.2 Optimizing data distribution . . . 118
7.2.3 Optimizing data redistribution . . . 120

7.2.4 Distributed-shared memory architectures 120

REFERENCES **122**

VITA **133**

LIST OF TABLES

TABLE **PAGE**

3.1 Communication model parameters 21

4.1 Communication estimate costs models 36
4.2 Communication primitives 46
4.3 Detected phases and estimated execution times (sec) for ADI2D 52

6.1 Data partitioning for initial tests 93
6.2 Program efficiencies vs. optimization 99
6.3 Mesh configurations 101
6.4 Data partitioning for pipeline tests 102
6.5 Comparison of the granularity estimates to the optimal granularity 103
6.6 Empirically estimated time (ms) to transpose a 1-D partitioned matrix 110

LIST OF FIGURES

FIGURE **PAGE**

1.1 The HPF data mapping model . . . 3
1.2 Example data distributions . . . 4
1.3 PARADIGM Compiler Overview . . . 5

3.1 Message vectorization . . . 23
3.2 Message vectorization algorithm . . . 23
3.3 Message aggregation . . . 24
3.4 Communication placement algorithm . . . 25
3.5 Code example requiring message pipelining . . . 27
3.6 Examples of message pipelining . . . 28
3.7 Estimation framework for coarse grain pipelining . . . 30

4.1 Two-dimensional Fast Fourier Transform . . . 40
4.2 Overview of the dynamic distribution approach . . . 41
4.3 2-D Alternating Direction Implicit iterative method (ADI2D) . . . 42
4.4 Phase decomposition . . . 43
4.5 Communication graph and some example initial edge costs for ADI2D . . . 46
4.6 Example graph illustrating the computation of a cut . . . 47
4.7 Partitioned communication graph for ADI2D . . . 48
4.8 Phase transition graph for ADI2D . . . 50
4.9 Pseudo-code for the partitioning algorithm . . . 53
4.10 Hierarchical Component Affinity Graph (HCAG) for ADI2D . . . 57

5.1 Overview of the array redistribution data-flow analysis framework . . . 61
5.2 Splitting CFG nodes to obtain DFG nodes . . . 65
5.3 Distribution set using bit vectors . . . 67
5.4 Example call graph and depth-first traversal order . . . 68
5.5 Distribution synthesis . . . 71
5.6 Interprocedural analysis for distribution synthesis . . . 72
5.7 Redistribution synthesis . . . 74
5.8 Interprocedural analysis for redistribution synthesis . . . 76
5.9 Initialization for redistribution synthesis . . . 77
5.10 Algorithms for detecting non-conforming HPF programs . . . 80
5.11 Example of static distribution assignment . . . 82

5.12 Example loop shown with flow transitions 84
5.13 Synthetic example for interprocedural redistribution optimization 87
5.14 2-D ADI with interprocedural redistribution optimization 89
5.15 Example of loop invariant redistribution 91

6.1 Run-time resolution 94
6.2 Reduced loop bounds and coalescing 94
6.3 Message vectorization 94
6.4 Message aggregation 94
6.5 Comparison of combined optimizations 99
6.6 Performance comparison 101
6.7 Performance comparison of pipelining vs. granularity 105
6.8 Traces from SOR with pipelining on an Intel Paragon 106
6.9 Modes of parallel execution for ADI2D 108
6.10 Performance of ADI2D 109
6.11 Phase transition graph and solution for Shallow 112
6.12 Performance of Shallow 113

CHAPTER 1

INTRODUCTION

Distributed-memory massively parallel multicomputers can provide the high levels of performance required to solve the Grand Challenge computational science problems [1]. Multicomputers, such as the Intel iPSC/860, the Intel Paragon, the IBM SP-1 /SP-2, the NCUBE/2, and the Thinking Machines CM-5, offer significant advantages over shared-memory multiprocessors in terms of both cost and scalability. Unfortunately, extracting all of the computational power from these machines requires users to write efficient software for them, which is a laborious process. One major reason for this difficulty is the absence of a global address space. As a result, the programmer has to manually distribute computations and data across processors and manage communication explicitly.

To overcome this difficulty, significant research effort has been aimed at developing source-to-source parallelizing compilers for multicomputers that relieve the programmer from the task of program partitioning and communication generation. As a result, High Performance Fortran [2] (HPF) has been developed in a collaborative effort between researchers in industry and academia to help standardize the specification of data distribution for such a compiler.

An HPF compiler must partition the computation and generate any required communication, but the actual specification of data distributions still remains a responsibility of the programmer. The quality of the data distribution for a given application is crucial to obtaining high performance on distributed-memory multicomputers. By selecting an appropriate data distribution, the amount of communication required by an application can be dramatically reduced. The resulting performance using a given data distribution therefore depends on how well the compiler

can optimize the remaining communication. In this thesis, we present several techniques to optimize communication and, based on the performance of these optimizations, automatically select the best data partitioning for a given application to relieve the programmer of the burden of selecting the data distribution.

The work in this thesis has been implemented as part of the PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project [3] in order to develop an automated means to parallelize and optimize sequential programs for efficient execution on distributed-memory multicomputers.

1.1 Overview of High Performance Fortran

High Performance Fortran (HPF) was designed to provide a machine-independent programming model for compiling (initially regular, dense-matrix) scientific programs for efficient execution on distributed-memory multicomputers. By providing directives to describe the distribution of data across the processors of a machine, computation is implicitly distributed (typically using the *owner-computes* rule, which states that the processor owning a data item performs all computations for that item). Any non-local data required in a given computation must be obtained through interprocessor communication. Thus, the programmer is still responsible for choosing partitionings that perform well but is freed from having to write code for the communication of data in the program. As compared to other efforts to standardize distributed-memory programming using message passing, such as the Message Passing Interface [4] (MPI), HPF provides a higher-level programming model using the philosophy that the complexity (of code partitioning and communication generation) can be shifted into the compiler.

In the HPF data mapping model, shown in Figure 1.1, dimensions of distributed arrays are first aligned with one another, and then distributed onto a rectilinear arrangement (or mesh) of abstract processors. The abstract processor mesh is then mapped to physical processors. Each array is either aligned to another, explicitly distributed, or distributed according to the compiler's choice.

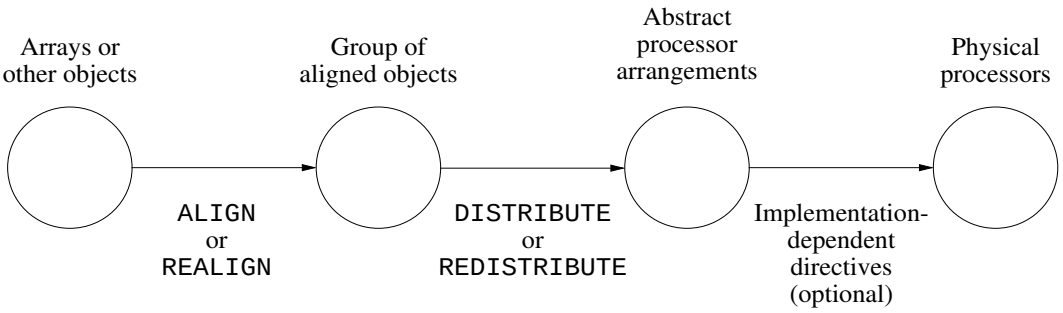


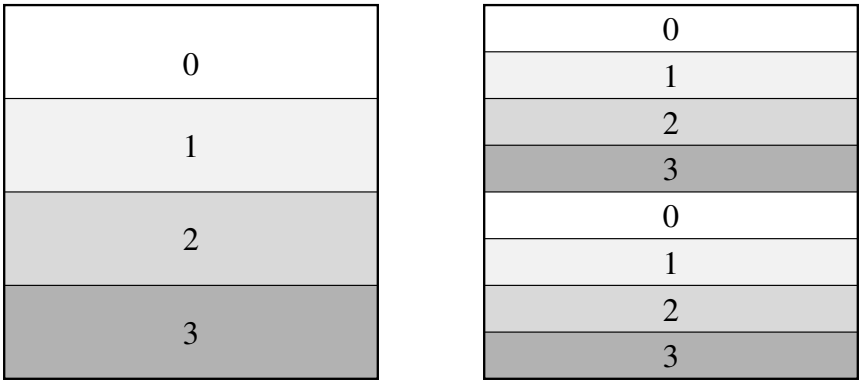
Figure 1.1: The HPF data mapping model

In Figure 1.2, several examples of HPF data distributions are shown for a two-dimensional array distributed onto four processors. This illustrates how each dimension of an array can be given a different distribution. Blocked and cyclic distributions are actually two extremes of a general distribution commonly referred to as block-cyclic (or cyclic(k) where k is the block size). A block-cyclic distribution on a dimension of an array assigns a fixed size block of that dimension to each processor in round-robin fashion until the entire dimension is distributed. A block distribution is equivalent to a block-cyclic distribution in which the block size is the size of the original array dimension (N) divided by the number of processors (P), cyclic($\frac{N}{P}$). A cyclic distribution is simply a block-cyclic distribution with a block size of one, cyclic(1).

In HPF, dynamic distributions can be described explicitly using executable redistribution directives (`REDISTRIBUTE` or `REALIGN`, which specify where new distributions become active) or implicitly by calling functions[1] (which require different data distributions than the calling function). In order to actually compile an HPF program into an efficient form, however, both the redistribution[2] operations as well as the possible distributions for the individual blocks of code must be known. Since the HPF `REDISTRIBUTE` and `REALIGN` directives only specify redistribution information, both the intra- and inter-procedural control flow through a program must be examined to decide exactly which redistribution operations were last performed in order to determine which distributions are active at any given point in the program.

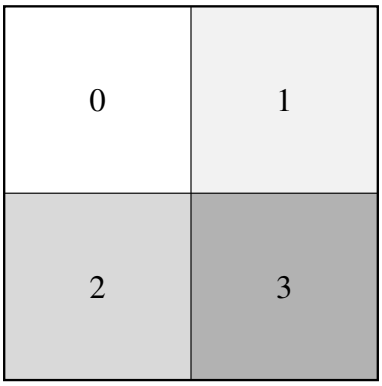
[1]For the purposes of simplifying this discussion, the word *function* will be used throughout this thesis to refer to either a Fortran program or subprogram (function or subroutine) and should be considered to be interchangeable with all of these terms.

[2]In the HPF standard [5], the word *remapping* is used to refer to *redistribution* in order to avoid associating it with only the HPF `REDISTRIBUTE` directive. For this thesis, the word *redistribution* will be used equivalently to refer to remapping due to either the HPF `REDISTRIBUTE` or `REALIGN` directive.

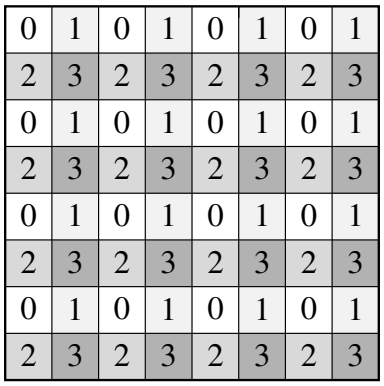


(a) (BLOCK, $*$)

(b) (CYCLIC(k), $*$)



(c) (BLOCK, BLOCK)



(d) (CYCLIC(k_1), CYCLIC(k_2))

Figure 1.2: Example data distributions

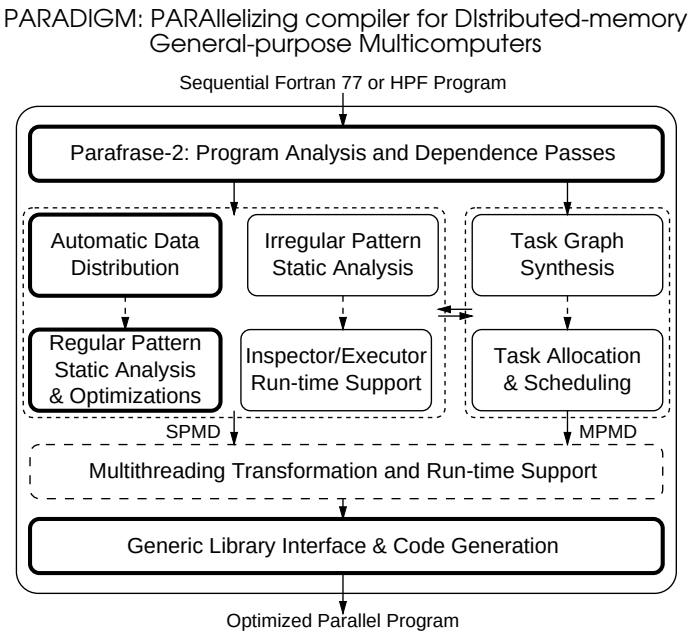


Figure 1.3: PARADIGM Compiler Overview

1.2 Overview of the PARADIGM Project

Figure 1.3 shows a functional illustration of how we envision the complete PARADIGM compilation system [3]. The compiler front-end currently accepts either sequential Fortran 77 [6] or High Performance Fortran [7] and produces an optimized explicit message-passing version (in the form of a Fortran 77 program with calls to the selected message-passing library and the PARADIGM run-time library). The following are brief descriptions of the major phases that are most closely related to the communication and data distribution optimizations presented in this thesis. The remaining phases in the compiler are described in the overview of the PARADIGM system [3].

1.2.1 Program analysis

Parafrase-2 [6] is used as a preprocessing platform to parse the sequential program into an intermediate representation and to analyze the code to generate flow, dependence, and function call information that is used throughout the rest of the compiler. A number of machine-

independent code simplifying transformations, such as constant propagation and induction variable substitution, are also performed at this stage.

1.2.2 Automatic data partitioning

The compiler can currently select a static distribution of data using a constraint-based algorithm [8], which determines both the best configuration of an abstract multidimensional mesh topology along with how program data should be distributed on the mesh. As the interactions between the selection of data distributions and the use of available communication optimizations are tightly linked, estimates of the time spent in computation and communication drive the selection of the data distribution. Available communication optimizations performed by the compiler are reflected in the estimates in order to correctly determine the best distribution.

For complex programs, static data distributions may be insufficient to obtain acceptable performance on distributed-memory multicomputers. By allowing the data distribution to dynamically change over the course of a program's execution, this problem can be alleviated by matching the data distribution more closely to the different computations performed throughout the program. Such dynamic partitionings can yield higher performance than a static partitioning when the redistribution is more efficient than the communication pattern required by the statically partitioned computation. The selection of distributions that dynamically change over the course of a program's execution adds another dimension to the data partitioning problem. As part of this thesis, we present a technique that extends the static data partitioning algorithm to automatically determine those partitionings most beneficial over specific sections of a program while taking into account the added overhead of performing redistribution.

1.2.3 Compiling regular computations

Using the *owner-computes rule*, the compiler divides computation across processors according to the selected data distribution and generates inter-processor communication for required non-local data. A direct application of this rule without further optimizations leads to *run-time resolution* [9], which results in code that computes the ownership and communication for each

reference at run time. To avoid the overhead of computing ownership at run time, static analysis is used to partition loops at compile time (known as *loop bounds reduction* [10]) allowing processors to execute only those iterations that have assignments that write to local memory.

PARADIGM applies the owner-computes rule to distribute computation across processors by statically partitioning loops. Given a program's data distribution information, which can be either manually specified with HPF directives or automatically selected by the compiler, the ACCESS sets of all references enclosed in loops are first computed. The ACCESS set of a reference with respect to a particular processor is the set of loop iterations for which that reference accesses data which is locally owned by that processor. According to the owner-computes rule, a processor only performs the computations in which the left-hand side (LHS) of an assignment statement references data which is owned by that processor. Loops are partitioned by letting each processor execute only those iterations which are in the union of the ACCESS sets of the LHS references in the loop body (also potentially masking each statement in the body by its own LHS ACCESS set). This union forms the *reduced iteration set* of the loop which is used to compute the *reduced loop bounds* [10–12].

The references in assignment statements are also analyzed to detect the need for communication. Communication descriptors are constructed to summarize the iterations requiring communication of non-local data, the processors involved, and the exact regions of the arrays to be sent or received. To determine which iterations require interprocessor communication, PARADIGM computes the set difference between the left-hand side and right-hand side ACCESS sets [11, 12]. Once the communication descriptors have been computed for individual references, a number of communication optimizations (which are presented as part of this thesis) can be performed to reduce the overhead of communication [13].

1.2.4 Generic library interface

Support for specific communication libraries is provided through a generic library interface. Since more information is carried internally than is required by any specific implementation, the mapping is obtained by selecting a subset of the available parameters. For each supported

library, abstract functions are mapped to corresponding library-specific code generators at compile time.

Library interfaces have been implemented for Thinking Machines CMMD, Parasoft Express, MPI [4], Intel NX, PVM [14], and PICL [15]. Execution tracing, as well as support for multiple platforms, is also provided in Express, PVM, and PICL. The portability of this interface allows the compiler to easily generate code for a wide variety of machines.

1.3 Thesis Contributions

The contributions made in this thesis are focused in three main areas of performance optimization compilation techniques for distributed-memory multicomputers, each of which builds upon the previous one.

(1) Optimizing communication,

(2) Optimizing data distribution using the available communication optimizations, and

(3) Optimizing redistribution for dynamic data distributions.

Since the resulting performance using a given data distribution depends heavily on how well the compiler can optimize the remaining communication, we first implemented several existing optimization techniques in the prototype PARADIGM compiler and evaluated their performance on several distributed-memory architectures. As a result of this effort we have also developed a more robust message vectorization algorithm based on the dependence direction vector as well as a more comprehensive estimation framework for balancing the available parallelism with the overhead of communication in pipelined computations.

In the area of optimizing data distribution, we have developed a dynamic data partitioning algorithm as a layer built on top of the existing static partitioner [8]. A branch-and-bound technique is used to recursively decompose a program into a number of phases (each with a selected static distribution) after which redistribution costs are taken into account to determine which phases to use. Our data partitioning techniques make good use of performance estimates of

communication and computation in order to determine where to partition a program into subphases as well as to determine which phase sequence will result in the best performance.

To support the dynamic partitioning techniques, we have extended the static data partitioner, implemented in PARADIGM as part of a previous thesis [8], to function as a reentrant module in order to obtain static partitionings for individual regions of a program. We have extended the underlying communication and computation cost model estimation framework, using a virtual function interface similar to that used in the generic library interface, to make the cost estimation framework more architecture independent as well as easily extensible. We have also introduced the idea of a Hierarchical version of the Component Affinity Graph [16] (HCAG), which provides a representation for efficiently computing alignments of subregions of a program as well as performing interprocedural alignment analysis.

Finally, we have developed an interprocedural data-flow framework which provides a means to convert between a distribution-based representation (as specified by the data partitioner and required for compilation) and a redistribution-based form (as specified by HPF) while optimizing the amount of redistribution that must be performed. In addition to supporting the data partitioner, this framework also allows us to optimize dynamic HPF programs (using `REDISTRIBUTE` and `REALIGN` directives) as well as convert such programs into equivalent static versions through a process we refer to as Static Distribution Assignment (SDA) providing dynamic HPF support for existing subset HPF compilers.

We have evaluated the implementation of these optimizations in the PARADIGM compiler framework using several small kernels and example programs and measured their effect on the performance of the programs using an Intel Paragon and Thinking Machines CM-5.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a summary of related work that has been performed in the area of optimizing communication and data distribution. Chapter 3 presents several existing techniques for optimizing communication as well as the improvements we have made. Chapter 4 presents the technique currently used to obtain

static data distributions as well as the new techniques we have developed which select dynamic data distributions to further improve performance for applications which require different distributions over different phases of execution. Chapter 5 describes the interprocedural data-flow framework which converts distribution-based representations into an equivalent redistribution-based form while optimizing the amount of redistribution that must be performed. Chapter 6 presents the evaluation of these optimizations in the PARADIGM compiler framework using several small kernels and example programs and measuring their effect on the performance of the programs using an Intel Paragon and Thinking Machines CM-5. Finally, Chapter 7 presents the conclusions on this work, and discusses directions for future work in this area.

CHAPTER 2

RELATED WORK

In this chapter we will describe the related work in each of the three areas of focus in this thesis:

(1) Optimizing communication,

(2) Optimizing data distribution using the available communication optimizations, and

(3) Optimizing redistribution for dynamic data distributions.

2.1 Optimizing Communication

As the issues related to optimizing communication for distributed-memory multicomputers are very similar to the issues which arise when optimizing accesses to memory hierarchies in general, a considerable range of research has been performed in areas directly or indirectly related to the communication optimizations described in this thesis. A number of techniques have been developed in the areas of blocking [17–20] and tiling [21, 22] algorithms for improved locality cache performance, software controlled prefetching to reduce miss penalties for both uniprocessor [23–25] and multiprocessor systems [19, 26–28], and combining communication operations required to obtain non-local data in distributed-memory systems [13, 19, 29–36] to reduce the communication overhead.

In cases in which loops contain cross-iteration dependencies due to recurrences within the innermost loops, parallelism can be extracted by executing the loop in a pipelined fashion. It

is possible to automatically extract pipeline parallelism from such recurrences through the use of general techniques such as the the hyperplane method[1] [37–43] and unimodular transformations [22, 44, 45]. Such techniques have been widely used in the design of VLSI systolic arrays [42, 46, 47], the extraction of `doacross` parallelism [48, 49], as well as the development of software pipelining for distributed-memory machines [29, 32, 36, 50, 51].

Specifically for distributed-memory machines, Gallivan, Jalby, and Gannon [19] discussed some of the problems, in general terms, associated with automatically restructuring data and generating communication in order to improve performance for distributed-memory machines while still maintaining the correctness of the original program. Rogers and Pingali [29, 36] examined several optimizations that address the issues of overhead and synchronization in message transmission by comparing the results of the automatic techniques with hand-coded versions of several programs [29, 36]. Tseng [50, 51] described several transformations used in the AL compiler that relate to software pipelining for distributed-memory machines. In [30], Gerndt described the message coalescing and vectorization optimizations in detail as implemented as part of the SUPERB project [52]. Balasundaram, Fox, Kennedy, and Kremer [31] presented a message vectorization technique in relation to a performance estimation framework based on training sets. In [32], Hiranandani, Kennedy, and Tseng have evaluated a number of communication optimizations performed by the Fortran D [32] compiler and proposed a measure for controlling the granularity of a software pipeline. Amarasinghe and Lam [34] used data flow analysis techniques to determine where data was last written while optimizing communication in the SUIF compiler. Gong, Gupta, and Melhem [33] as well as Gupta, Schonberg, and Srinivasan [35] have also examined the use of data flow frameworks in order to eliminate redundant messages between separate loop nests in addition to optimizing communication within a single loop nest.

In this thesis we present our implementation of a number of previously proposed communication optimizations in the PARADIGM compiler, a more robust message vectorization algorithm based on the dependence direction vector, a more comprehensive estimation framework for balancing the available parallelism with the overhead of communication in pipelined com-

[1] Which has also been referred to as the *wavefront* method [37].

putations, as well as an evaluation of the presented optimizations on different distributed architectures.

2.2 Optimizing Data Distribution

2.2.1 Static partitioning

As previously stated, for distributed-memory multicomputers, the quality of the data partitioning for a given application is crucial to obtaining high performance. This task has traditionally been the user's responsibility, but in recent years much effort has been directed to automating the selection of data partitioning schemes. Several researchers have proposed static data partitioning systems that are able to select data distributions that remain in effect for the entire execution of an application. The two main problems which arise in selecting static distributions are determining how different arrays are aligned to each other and how the aligned arrays are distributed across the processors.

In [53], Mace examined the problem of determining the best memory storage patterns (shapes) for interleaved memories in vector processors showing that the optimal data partitioning problem is NP-complete. Knobe, Lukas, and Steele [54] developed alignment techniques for SIMD machines by constructing preferences between data references. A greedy heuristic is used to choose the highest weighted preferences within cycles of the preference graph. In [16], Li and Chen showed that the problem of array alignment is NP-complete and presented a heuristic solution based on bipartite graph matching. They also were the first to describe how to synthesize collective communication operations based on a pattern-matching approach on data references in the source program. Ramanujam and Sadayappan [55] developed a technique which selects data distributions in which a communication-free partitioning exists. They used a linear algebra framework to describe array access patterns within a parallel loop and determine when a communication-free data distribution exists. As a communication-free distribution usually does not exist for an entire program, their techniques mainly apply to individual loop nests. In the description of the AL compiler, Tseng [50, 51] presents a technique for determining the best

data distribution given which dimension of each array is to be distributed. Wholey [56] showed how performance estimation is a key in selecting good data distributions. As a hill climbing technique was used to select the best number of processors and dimensions to distribute, one main drawback of this technique is that it could get caught in local minima. Gupta and Banerjee [57] described a constraint-based algorithm to determine both the best configuration of an abstract multidimensional mesh topology along with how program data should be distributed on the mesh. Their alignment analysis is based on earlier work on multidimensional array alignment [16] extending their approach with estimates of the communication cost penalty for misaligned array dimensions. In [58], Chatterjee, Gilbert, Schreiber and Teng present a framework for determining the alignment of arrays in data-parallel languages such as Fortran 90. They decomposed the alignment problem into axis, stride, and offset subproblems and were able to obtain alignments more general than those provided by the owner-computes rule. Anderson and Lam [59] developed an iterative technique using a linear algebra framework, based on both computation and data decompositions. Their technique iterated until the relationship between the data and computation decompositions is valid. Bau, Koduklula, Kotlyar, Pingali, and Stodghill [60] have also developed a technique for determining the alignment of arrays based on a linear algebra framework. They solved the resulting alignment equations by reducing the problem to determining a basis for the null space of a matrix, but heuristics still must be used to determine which constraints to leave unsatisfied when the system is overconstrained.

2.2.2 Dynamic partitioning

The selection of distributions that dynamically change over the course of a program's execution adds another dimension to the data partitioning problem. As all of the difficulties involved in automatically selecting high quality static data distributions still exist in the selection of dynamic data distributions, many of the techniques used in the following approaches were based on algorithms or representations described previously in Section 2.2.1.

Hudak and Abraham have proposed a method for selecting redistribution points for Non-Uniform Memory Access (NUMA) shared-memory multiprocessors [61, 62]. Their technique

is based on locating significant control flow changes in the program and inserting remapping points [62]. Remapping points are inserted between loop nests with different nesting levels, around control loops, and between a loop that requires nearest neighbor communication (which is assigned a block partitioning) and a loop that is linearly varying (which is given a cyclic partitioning). Once remapping points have been added to the program structure, a merge phase is performed to remove any unnecessary data redistribution. Remapping points are removed between two phases with identical distributions as well as when one phase has an *unspecified* distribution (in which case it is assigned a distribution from an adjacent phase). This heuristic was shown to improve the performance of several programs as compared to a strict data-parallel implementation. For a program consisting of a matrix multiplication followed by an LU factorization, they observed a 26% improvement by applying this technique.

Chapman, Fahringer, and Zima have noted that there are many known good data distributions for important numerical problems that are frequently used [63]. They describe the design of a distribution tool that makes use of performance prediction methods when possible, but also heuristically uses empirical performance data when available. They have also developed cost estimates which model the work distribution, communication and data locality of a program while taking into account the features of the target architecture [64]. The combination of estimation techniques along with profile information will be used to guide their system in potentially selecting lists of distributions for different arrays that appear in the program. Currently, the performance prediction system has been integrated into a compiler based on Vienna Fortran [65], and research is under way on their partitioning system.

Anderson and Lam [59] have also proven that the dynamic decomposition problem is NP-hard by transforming the colored multiway cut problem (which is known to be NP-hard) into a subproblem of dynamic decomposition. They address the dynamic distribution problem by using a communication graph in which nodes are loop nests and edges represent the time for communication if the two loop nests involved in an edge are given different distributions. A greedy heuristic is used to combine nodes in such a way that the largest potential communication costs are eliminated first while maintaining sufficient parallelism. Their technique works from the bottom-up, examining highest cost edges first, combining two nodes if the cost of ex-

ecuting the combined nodes is less than the the sum of the original costs of the two nodes and the redistribution cost.

Work by Bixby, Kennedy and Kremer [66], by Kremer [67], and more recently by Garcia, Ayguadé, and Labarta [68] formulates the data partitioning problem in the form of a 0-1 integer programming problem. For each phase, a number of candidate partial data layouts are enumerated along with the estimated execution costs. The costs of the possible transitions among the candidate layouts are then computed forming a data layout graph. A static performance estimator is used to obtain the node and edge weights of the graph. Since each phase only specifies a partial data distribution, redistribution constraints can cross over multiple phases, thereby requiring the use of 0-1 integer programming. The number of possible data layouts for a given phase is exponential in the number of partitioned array dimensions resulting in potentially large search spaces. To benefit from advanced techniques in the field of integer programming, the resulting formulation is processed by a commercial integer programming tool. In some previous work [69], Kennedy and Kremer also examined a dynamic programming technique that could determine dynamic distributions in polynomial time if each candidate layout specified a mapping for every array in the program. The main drawback with both of these techniques is that the selection of phases must be made a priori to the formulation of the problem. This means that the size of the phases should be as small as possible so that the correct solution is found and that as many candidate layouts as possible should be specified in order to find the best dynamic data distribution. These factors contribute to the increase in the size of the search space.

Bixby, Kennedy, and Kremer [66] have also described an *operational definition* of a phase, which is defined as the outermost loop of a loop nest such that the corresponding iteration variable is used in a subscript expression of an array reference in the loop body. Even though this definition restricts phase boundaries to loop structures and does not allow for overlapping or nesting of phases, it can be seen that for the example in Section 4.3.1 this definition is sufficient to describe the two distinct phases of the computation.

More recently, Sheffler, Schreiber, Gilbert, and Pugh [70] have applied graph contraction methods to the dynamic alignment problem [71] to reduce the size of the problem space that must be examined. Several localized graph transformations are employed to reduce the size of

the dynamic alignment problem significantly while still preserving the optimal solution. The alignment-distribution graph (ADG) is based on a single static assignment data-flow form of a data-parallel language. Nodes in the ADG represent data-parallel computations while edges represent the flow of data. The ADG itself is formed in a divide-and-conquer approach using heuristics to approximately solve a combinatorial minimization problem at each step, taking into account both redistribution costs as well as all candidate distributions, to determine where to partition the program into subphases [72]. Candidates are formed by first identifying the extents, or iteration space, of all objects in the program resulting in a number of clusters. These clusters appear to result in groups very similar to that formed by the operational definition of a phase. A set of representative extents is constructed by selecting one vector from each cluster. This set is then used to determine the preferred distributions of each dimension typically resulting in a few tens of candidate distributions. When the alignments of the candidates for two connected nodes are different, communication costs are modeled by the worst-case data transfer between any two candidate distributions for nodes involved. After the graph contraction transformations have been applied to the ADG, the final data distribution is selected using several different heuristics. By first reducing the size of the problem, they are able to apply more expensive heuristics to obtain better overall solutions.

In this thesis, we present a dynamic data partitioning technique which we have developed as a layer built on top of an existing static partitioner [8]. A branch-and-bound technique is used to recursively decompose a program into a number of phases (each with a selected static distribution) by only considering the current communication constraints within a phase under examination. After decomposing the program into a hierarchy of phases, redistribution costs are then taken into account to determine which phases to use. Performance estimates of communication and computation are used to determine where to partition a program into subphases as well as to determine which phase sequence will result in the best performance.

2.3 Optimizing Data Redistribution

Interprocedural analysis is a useful program analysis technique that has been applied to many different applications [73]. In the area of parallel processing, it has been widely used for dependence analysis and program parallelization in the presence of function calls [74–77], detecting references to stale data for improving the performance of cache coherent architectures [78], as well as for compiling static Fortran D programs [79].

The work by Hall, Hiranandani, Kennedy, and Tseng [79] defined the term *reaching decompositions* for the Fortran D decompositions (or distributions, in HPF terminology) which reach a function call site. Their work describes extensions to the Fortran D compilation strategy [80] using the reaching decompositions for a given call site to compile Fortran D programs that contain function calls as well as to optimize the resulting implicit redistribution. Since explicit redistribution did not exist in the Fortran D language, their techniques are limited to only optimizing implicit redistribution performed at function boundaries and do not address all the situations which can arise in HPF.

More specifically, in the area of array redistribution, there has been much recent work in generating efficient communication to actually perform the redistribution [81–87]. Techniques that have been applied to this problem range from resolving the array element-to-element mapping at run time [85] to various approaches for analyzing the communication patterns before performing the communication to minimize the total number of messages sent [81, 82, 84] to strip-mining redistribution operations in order to overlap the communication with the last computation, which is generating the data to be redistributed [86, 87].

The work by Coelho and Ancourt [81] also describes an optimization for removing useless remappings specified by a programmer through explicit realign and redistribute operations. In comparison to the work in the Fortran D project, they are also concerned only with determining which distributions are generated from a set of redistributions, but instead focus only on explicit redistribution. They define a new representation called a redistribution graph in which nodes represent redistribution operations and edges represent the statements executed between redistribution operations. This representation, although correct in its formulation, does not seem to

fit well with any existing analysis already performed by optimizing compilers and also requires first summarizing all variables used or defined along an edge (all possible paths between successive redistribution operations) in order to optimize redistribution. As will be seen in Section 5.1, this representation is actually the dual of our representation which suffers from neither of these problems. Even though their approach currently only performs their analysis within a single function, they do suggest the possibility of an extension to their techniques which would allow them to also handle implicit remapping operations at function calls but do not describe an approach.

The definition of reaching distributions, however, is still a useful concept. We extend this definition to also include distributions which reach any point within a function in order to encompass both implicit and explicit redistributions thereby forming the basis of the work presented in this thesis. In addition to determining those distributions generated from a set of redistribution operations, this extended definition allows us to address a number of other applications in a unified framework.

2.4 Summary

In this chapter we have described the related work in the areas of optimizing communication, data distribution, and data redistribution for distributed-memory multicomputers. In Chapter 3, we will present several existing techniques for optimizing communication as well as the improvements we have made. In Chapter 4, we will describe the techniques we have developed for optimizing data distributions, and in Chapter 5, we will describe the interprocedural dataflow framework we have developed to optimize redistribution.

CHAPTER 3

OPTIMIZING COMMUNICATION

The start-up latency, or overhead, of communication for distributed-memory multicomputers tends to be two to three orders of magnitude greater than the per-byte transmission rate. For this reason, if multiple communication operations can be performed together, thereby reducing the frequency of communication operations in exchange for increasing the length of the message communicated, the overhead can be amortized over multiple messages reducing the overall cost of communication. This property forms the basis for all of the communication optimizations that will described later in this chapter.

The point-to-point transfer cost of a message of m bytes can be modeled as a linear function of the message length parameterized by the overhead and rate (see Table 3.1) for a specific architecture

$$transfer(m) \quad = \quad ovhd + rate \cdot m \tag{3.1}$$

In this cost model, we assume that any added contribution due to distance (or hops) between communicating processors is negligible in comparison to both the overhead and total transmission time. Since most modern interconnection networks utilize techniques such as wormhole routing over packet-switched interconnection networks with relatively low switch costs to minimize this effect, this is a valid assumption. For a machine in which the parameters depend on the length of the message, the model takes on different values for different ranges of message

Table 3.1: Communication model parameters

Machine	Message size	$ovhd$ (μs)	$rate$ (μs)	$\frac{ovhd}{rate}$
Intel Paragon	any	50	0.018	2777
Intel iPSC/860 [88]	$m \leq 100$	60	0.50	120
	$m > 100$	160	0.36	444
Intel iPSC/2 [88]	$m \leq 100$	350	0.20	1750
	$m > 100$	700	0.36	1944
IBM SP-2	any	58	0.032	1812
TMC CM-5 [88]	any	86	0.12	716

length. The model for a machine such as the iPSC/860 would be

$$transfer(m) = \begin{cases} 60 + 0.50m & (\text{if } m \leq 100) \\ 160 + 0.36m & (\text{if } m > 100) \end{cases}$$

To reduce the total amount of communication overhead, the communication optimizations examined in this chapter combine messages between different communication operations of unmodified overlapping array regions (message coalescing, Section 3.1); across a dimension of an array traversed over different iterations of a loop nest (message vectorization, Section 3.2); of different arrays communicated between the same source and destination processors (message aggregation, Section 3.3); and of successive pipelined messages which arise in the presence of inner loop recurrences (message pipelining, Section 3.4).

3.1 Message Coalescing

Separate communication for different references to the same data is unnecessary if the data has not been modified between uses [30]. When statically analyzing the access patterns, these redundant communication operations are detected and coalesced into a single message, allowing the data to be reused rather than communicated for every reference. For sections of arrays which are not disjoint, unions of their overlapping communication descriptors [11,12] ensure that each

unmodified data element is communicated only once. Such software-based caching is always beneficial since entire communication operations can be eliminated.

3.2 Message Vectorization

Non-local elements of an array that are indexed within a loop nest can also be *vectorized* into a single larger message instead of being communicated individually (see Figure 3.1) [19, 30]. The "itemwise" messages are combined, or *vectorized*, as they are lifted out of the enclosing loop nests to a selected level. Vectorization reduces the total number of communication operations, but at the cost of increasing the message length.[1] For this reason, vectorization should perform well on machines with high communication overheads. The vectorized data may also have to be packed into a contiguous buffer before communication (unless the array section to communicate is already contiguous [89]). For all of the machines we have examined (in Table 3.1), the combined overhead of all of the non-vectorized (single-element, inner loop) communication operations greatly outweighs the cost of packing/unpacking the data at the vectorization loop level.

Dependence analysis is used to determine the outermost loop at which the combining can be applied. The algorithm DEP-UPLEVEL shown in Figure 3.2 examines the direction vector of an incoming dependence arc for a given reference and determines the level which carries the dependence. It then returns this location as a number of levels up from the current nesting level.[2] The vectorization level is the lowest loop level which carries all of the the flow dependencies for a given reference which is computed by VECT-UPLEVEL as the minimum up level of all incoming flow dependencies for that reference.

The use of the vectorization level is shown in the communication placement algorithm in Figure 3.4 for completeness, but it is actually computed when first constructing the communication descriptors [11,12]. Both the construction and placement of the communication descriptors should use the same level. Because DEP-UPLEVEL examines the dependence direction vector

[1] Management of available memory may require that large regions of data be only partially vectorized [32].

[2] The "nesting level" is the number of loops which contain the reference while the "up level" is the number of loops to move up from the current level to reach the new nesting level.

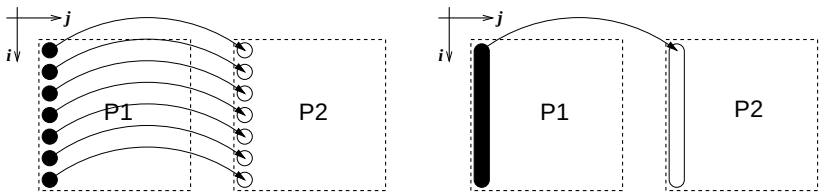


Figure 3.1: Message vectorization

```
VECT-UPLEVEL(ref)
 1  ▷ Start vect_level at topmost level
 2  vect_uplevel = NESTLEVEL(STMT(ref))
 3  ▷ Minimum dependence "up level" determines vect_uplevel
 4  for each edge in DEPINLIST(ref)
 5      do if DEPTYPE(edge) = FLOW
 6            then vect_uplevel ← MIN(vect_uplevel, DEP-UPLEVEL(edge))
 7  return (vect_uplevel)

DEP-UPLEVEL(edge)
 1  nest_level ← NESTLEVEL(STMT(DEPSINK(edge)))
 2  curr_level = −1          ▷ Dependence level not yet found
 3  ▷ Direction vectors stored from outer to inner loop order
 4  for i ← 0 to DEPDIRLEN(edge)
 5      do if [<] ∈ DEPDIR(edge)[i]
 6            then curr_level = i          ▷ Possible level with a "≤" direction
 7         if ([<] ∈ DEPDIR(edge)[i]) and ([=] ∉ DEPDIR(edge)[i])
 8            then curr_level = i          ▷ Definite level with a strict "<" direction
 9                 break
10  if curr_level < 0
11    then curr_level = i
12  if (curr_level < DEPDIRLEN(edge))
13     and ([<] ∈ DEPDIR(edge)[curr_level])
14    then curr_level++      ▷ Carried within loops with a "<" direction
15  uplevel ← nest_level − curr_level
16  return (uplevel)
```

Figure 3.2: Message vectorization algorithm
Macros used to access fields in structures are shown in CAPS.
Functions are indicated by SMALLCAPS.

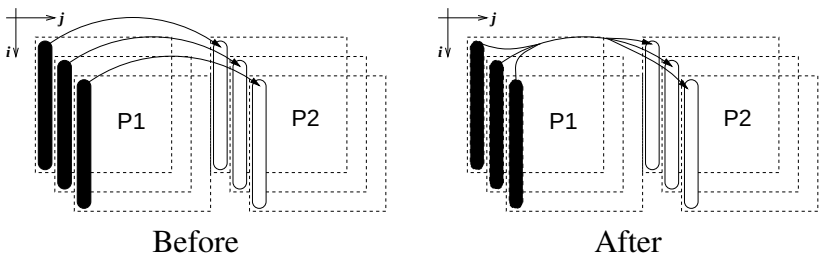


Figure 3.3: Message aggregation

(for both $<$ and $\leq$ dependencies) to determine the level which carries the dependence, this is more robust than other dependence-based message vectorization algorithms [30, 31, 36] and is even useful for selecting vectorization levels for partially vectorizable forward references in the presence of recurrences (as will be discussed in Section 3.4).

3.3 Message Aggregation

Multiple messages (corresponding to several array sections) to be communicated between the same source and destination can also be *aggregated* into a single larger message [30] as shown in Figure 3.3. The communication operations at a given point in a program are first sorted by their source and destination before generating the communication operations. This is also shown in the PLACE-COMM algorithm in Figure 3.4, but in the actual implementation the sorting is only performed once for each list of communication descriptors before code generation instead of as each operation is placed. Messages with identical destinations can then be collected into a single communication operation. The gain from aggregation is similar to vectorization in that multiple communication operations can be eliminated at the cost of increasing the message length.

Aggregation can be performed between communication operations of individual data references as well as vectorized communication operations (both of which will be examined in Section 6.1). With respect to buffering, if the data is already packed for the individual messages, then there is no extra time spent packing the aggregated messages. Packing multiple messages

```
PLACE-COMM(comm, ref)
 1  level ← NESTLEVEL(STMT(ref))
 2  ▷ Select the vectorization level for the reference
 3  if comm_vect
 4    then level ← level − VECT-UPLEVEL(ref)
 5  ▷ Add comm to communication descriptor list
 6  commprev ← NIL
 7  commdesc ← COMMLIST(level)
 8  while commdesc and
 9        (not comm_coal or DISJOINT(comm, commdesc))
10        do commprev ← commdesc
11           commdesc ← commdesc→next
12  ▷ Coalesce overlapping communication operations
13  if comm_coal and commdesc ≠ NIL
14    then UNION(commdesc, comm)
15    else if commprev ≠ NIL
16           then commprev→next ← comm
17           else COMMLIST(level) ← comm
18         comm→next ← commdesc
19  ▷ If aggregating communication, sort by source and destination
20  if comm_aggr
21    then SORT(COMMLIST(level), COMPARE(src, dest))

      comm_coal = communication coalescing flag
      comm_vect = communication vectorization flag
      comm_aggr = communication aggregation flag
```

Figure 3.4: Communication placement algorithm

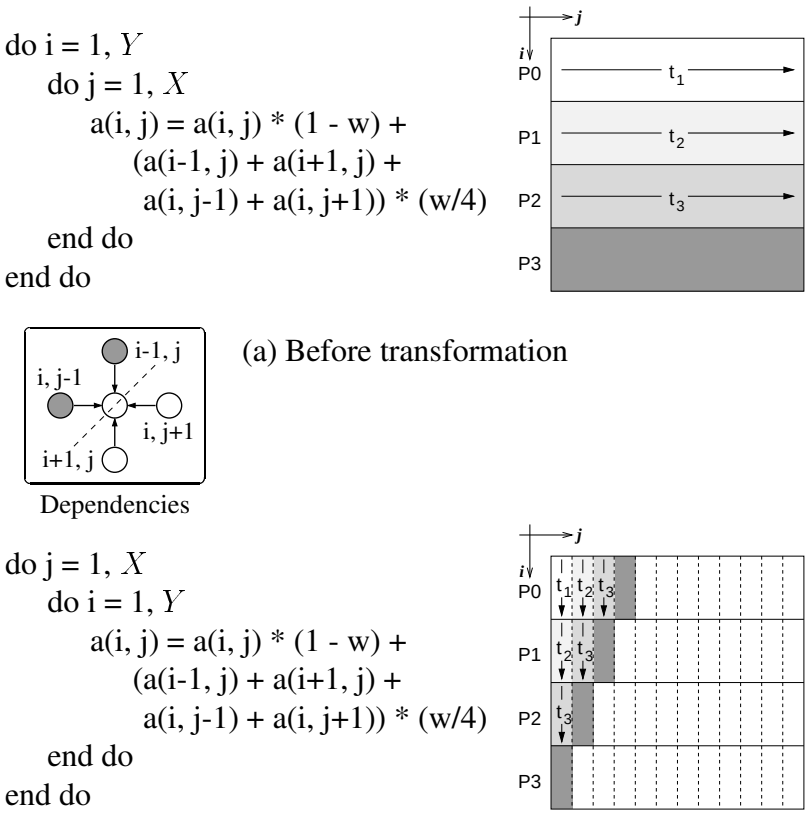
into the same buffer takes the same amount of time as it does to pack each individually into different buffers. Also, since it is likely that vectorized messages will require packing, the buffering associated with aggregating already vectorized communication is usually not a concern.

3.4 Message Pipelining

In loops in which there are no cross-iteration dependencies, parallelism is extracted by independently executing groups of iterations on separate processors. However, in cases in which there are cross-iteration dependencies due to recurrences within the innermost loops, it is not possible to immediately execute every iteration. When such recurrences are present in the program, the message vectorization algorithm (in Figure 3.2) will not be able to move all of the communication operations (more specifically, those associated with references involved in the inner loop recurrence dependencies) out of the innermost loops. The communication operations that remain within the innermost loops give rise to a message ordering which is pipelined across the processors which are partitioned across the recurrence [32, 36].

To illustrate how such situations can arise, a small code example that requires pipelining to extract parallelism is shown in Figure 3.5. Assuming that the data is distributed by rows, partitioning the code in Figure 3.5(a) will cause the first processor to perform the computation on every row it owns before sending the border row to the waiting processor, thereby serializing execution of the overall computation ($t_{1...3}$ indicate the processing order). In Figure 3.5(b), dependencies allow the loops to be interchanged resulting in the first processor to compute instead one partitioned column of elements before sending the border element of that column to the next processor. The next processor can now begin its computation much earlier (upon receipt of the border element). As this process continues, it gives rise to a pipelined execution ordering for the recurrence. As previously described in Chapter 2, such pipelining techniques have been widely used to solve recurrences on high-performance systems.

Ideally, if communication has zero overhead, such fine grain pipelining exposes the most parallelism in such computations since no processor will wait unnecessarily. Unfortunately, for distributed-memory systems, this is not completely true. By again considering the overhead of



(a) Before transformation

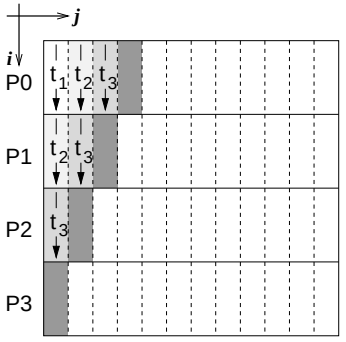
(b) After loop interchange

Figure 3.5: Code example requiring message pipelining

communication (in Table 3.1), the cost of performing numerous single element communications can be quite expensive. To address this problem, the total cost of communication can be reduced by increasing the amount of computation performed before communicating thereby balancing the resulting reduction in parallelism due to partial serialization with the gain achieved through amortizing the communication overhead. Reducing the total cost of communication by controlling the pipeline granularity has become known as *coarse grain pipelining* [32].

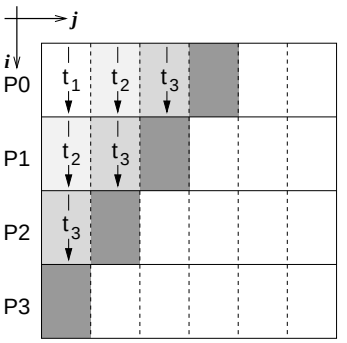
An example of the code that would be generated for message pipelining is shown in Figure 3.6. For fine grain pipelining, Figure 3.6(a), an element is communicated after each column of computation. In Figure 3.6(b) the serial j loop has been strip-mined to allow the computation of s columns before communicating. As the selection of an appropriate value of s is key to the performance of pipelined loops, this will be the focus of the remainder of this section.

```
do l = 1, L
    if (my$p> 0) send (my$p-1, ∝ X)
    if (my$p< p − 1) recv (my$p+1, ∝ X)
  do j = 1, X
      if (my$p> 0) recv (my$p-1, ∝ 1)
      do i = 1, min(Y-⌈Y/p⌉∗my$p, ⌈Y/p⌉)
          computation
      end do
      if (my$p< p-1) send (my$p+1, ∝ 1)
  end do
end do
```



(a) Fine grain pipelining

```
do l = 1, L
    if (my$p> 0) send (my$p-1, ∝ X)
    if (my$p< p-1) recv (my$p+1, ∝ X)
  do jj = 1, X, s
      jjmax = min(X, jj + s - 1)
      jjblk = jjmax - jj + 1
      if (my$p> 0) recv (my$p-1, ∝ jjblk)
      do j = jj, jjmax
          do i = 1, min(Y-⌈Y/p⌉∗my$p, ⌈Y/p⌉)
              computation
          end do
      end do
      if (my$p< p-1) send (my$p+1, ∝ jjblk)
  end do
end do
```



(b) Coarse grain pipelining

my$p	=	processor coordinate	jj	=	stripmining loop
p	=	total number of processors	jjmax	=	upper bound of stripmined loop
s	=	strip size	jjblk	=	actual loop blocking factor

Figure 3.6: Examples of message pipelining

3.4.1 Pipeline performance analysis

Note that in the code examples in Figure 3.6 an outer loop (l) has been added which encloses the pipelined loop nest. This type of pipeline is called a *two-level* pipelined loop as opposed to a *one-level* pipelined loop (previously shown in Figure 3.5). A two-level pipeline allows consecutive pipelines to be overlapped with each other, or chained, which can be seen more clearly in the model in Figure 3.7 between the completion of the first pipeline and the start of the second. The reason for using a two-level model is that a one-level pipeline model would be too conservative in that it would assume that there are barriers between successive pipelines (no chaining). The two-level model can capture the effect of a one-level model by setting the parameter L (the number of iterations of the enclosing loop) to one.

The generalized two-level pipeline model in Figure 3.7 corresponds directly to the code in Figure 3.6(b). In this model, the pipeline granularity is controlled by strip-mining the outer loop of the inner two-dimensional pipelined loop nest while chaining occurs between consecutive pipelines at the outermost enclosing loop. Definitions of all variables used in the following analysis are also provided in Figure 3.7.

Since the amount of available parallelism is reduced as the granularity, or strip size (s), is increased, this value must be carefully selected. If s is one, the pipeline is fine grained exposing the maximum parallelism. If s is equal to the bounds of the serial j loop (X), the pipeline is serialized (although chaining will still occur at the outermost level). Somewhere in between lies an optimal s that balances the cost of communication with the available parallelism. As s decreases, the amount of communication increases while the start-up cost decreases. As s increases, the amount of communication decreases while the start-up cost increases. To investigate this tradeoff more closely, an execution time estimate is developed from the framework in Figure 3.7 in order to analytically determine a strip size that minimizes both effects.

The first major phase of execution is the start-up time required to fill the pipeline. This is related to the number of processors as well as the strip size. From Figure 3.7, the start-up cost is equal to

$$\text{startup} = (p-1)(s \cdot \text{comp} + \text{comm})$$

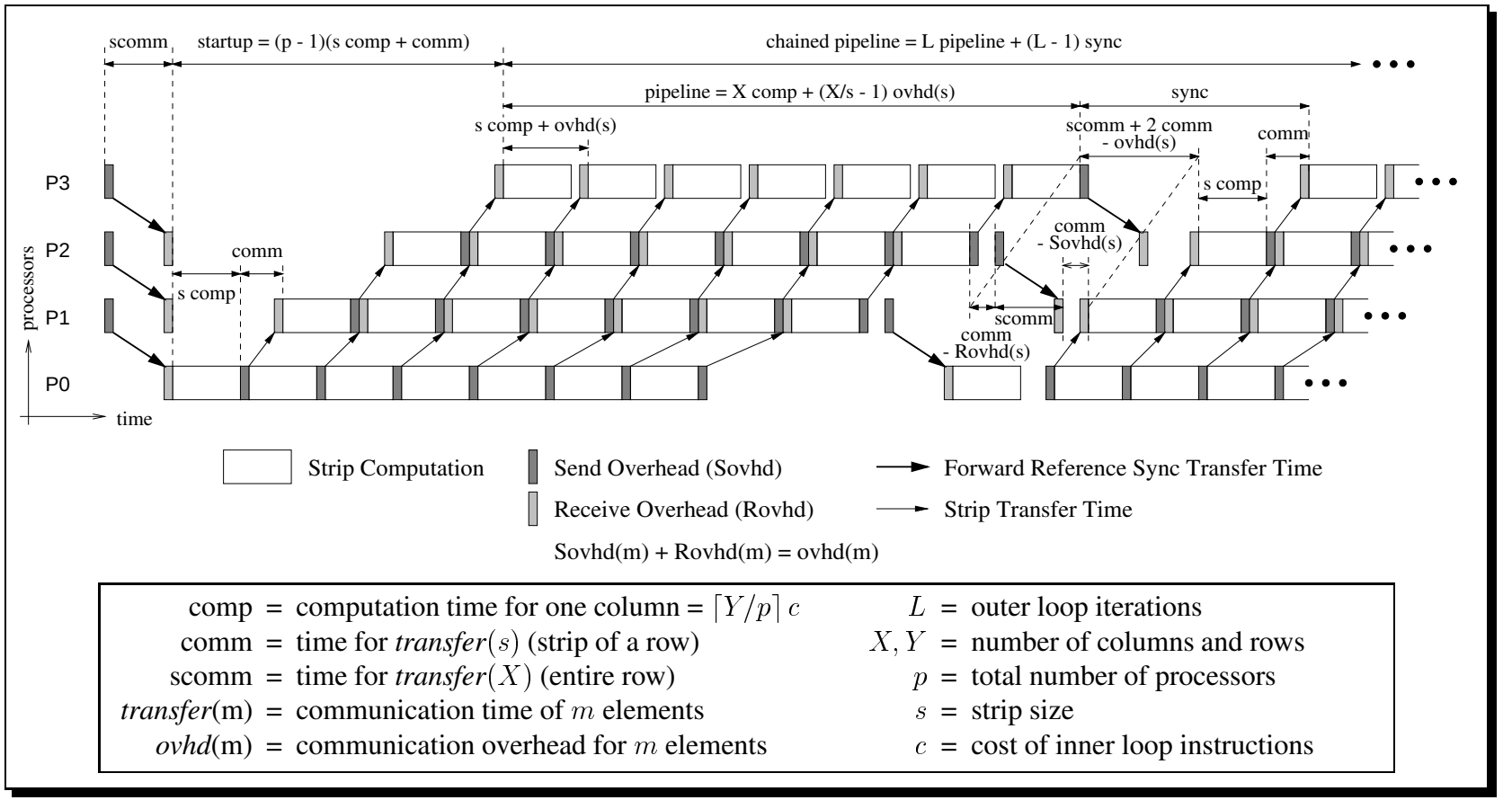


Figure 3.7: Estimation framework for coarse grain pipelining

The next portion of execution is the time spent in the pipeline. Ideally, with zero communication costs, this time should be equal to the amount of computation ($X \cdot \text{comp}$). However, because of the presence of communication, the time for each message communicated in the pipeline must also be taken into account. From Figure 3.7, the number of communication operations within the pipeline is $\left(\left\lceil \frac{X}{s} \right\rceil - 1\right)$ resulting in

$$\text{pipeline} = X \cdot \text{comp} + \left(\left\lceil \frac{X}{s} \right\rceil - 1\right) ovhd(s)$$

For loops with flow-dependencies caused by *forward* references (such as the a(i+1,j) and a(i,j+1) terms in the example in Figure 3.5), the execution of a sequence of pipelined loop nests will also generate communication incurring some additional synchronization costs to obtain the required data.[3] As the forward references access data that was computed in a previous pipeline, they can be partially vectorized and moved outside of the innermost loops. The vectorization level of the forward reference communication operations is located just inside the outermost enclosing loop in Figure 3.6 as determined by VECT-UPLEVEL.[4]

The "scomm" term is used to represent the amount of *synchronizing* communication. (If an entire row is communicated, then this cost is $transfer(X)$). Note that this will also be present in the start-up synchronization for the loop nest (see Figure 3.7.) As most parallel machines only support a single channel for memory transfer operations through the communication network, and since the pipeline will stall one stage waiting for a forward reference, this adds some extra delay to the "sync" term: Figure 3.7.

$$\text{sync} = (\text{scomm} + 2 \cdot \text{comm} - ovhd(s)) + (s \cdot \text{comp} + \text{comm})$$

Since L pipelines are chained together, the total execution time is therefore

$$\text{total cost} \;=\; \text{scomm} + \text{startup} + L \cdot \text{pipeline} + (L-1)\,\text{sync}$$

[3] In some applications there are no forward references and, therefore, no need to synchronize between outer loop iterations. In these cases each processor can proceed without any further synchronization.

[4] Note that the inner loop pipelined messages are placed one level above the level computed by VECT-UPLEVEL since the pipelining itself will actually preserve the dependencies carried by the true vectorization level.

$$= [LX + s(p + L - 2)] \left\lceil \frac{Y}{p} \right\rceil c + \quad (p + 3L - 4)\ transfer(s)$$
$$+ \left[L \left(\left\lceil \frac{X}{s} \right\rceil - 2 \right) + 1 \right]\ ovhd(s) + \quad L \cdot transfer(X) \tag{3.2}$$

By substituting the communication cost model previously presented in Equation (3.1) (with constant values for $ovhd$ and $rate$ for now) and b for the number of bytes that are communicated for each unit of the strip, the total cost becomes

$$\text{total cost } = [LX + s(p + L - 2)] \left\lceil \frac{Y}{p} \right\rceil c + \quad (p + 3L - 4)(ovhd + s \cdot b \cdot rate)$$
$$+ \left[L \left(\left\lceil \frac{X}{s} \right\rceil - 2 \right) + 1 \right]\ ovhd + \quad L(ovhd + X \cdot b \cdot rate)$$

The total cost can then be minimized with respect to the strip size to select the optimal granularity

$$\frac{\partial}{\partial s}\text{total cost } = (p + L - 2) \left\lceil \frac{Y}{p} \right\rceil c$$
$$+ (p + 3L - 4)b \cdot rate - L \cdot ovhd\ \frac{X}{s^2} = 0$$

Verification of the second derivative will show that this value of s is indeed a minimum. Solving for s results in

$$s = \sqrt{\frac{X \cdot L \cdot ovhd}{(p + L - 2)(b \cdot rate + \left\lceil \frac{Y}{p} \right\rceil c) + 2b(L - 1)rate}} \tag{3.3}$$

Since $ovhd$ and $rate$ are actually functions of s, Equation (3.3) is evaluated for each of the possible message size ranges in the communication model to obtain a value for s. If the program does not contain any flow dependencies caused by forward references, the "sync" and "scomm"

terms used in Equation (3.2) are zero resulting in the following expression for the minimum

$$s = \sqrt{\frac{X \cdot L \cdot ovhd}{(p-1)(b \cdot rate + \left\lceil \frac{Y}{p} \right\rceil c)}} \tag{3.4}$$

In comparison to this estimate, a simpler one-level strip size estimate developed in the Fortran D project [32] yields an estimate of s of the following form

$$s = \sqrt{\frac{p}{p-1}\frac{ovhd}{c}} \tag{3.5}$$

This estimate assumes that communication has a constant cost and that the array dimensions are square. Since it is based on a one-level pipeline model, it also assumes that chaining does not occur between consecutive pipelines. It is interesting to note that our two-level estimate (Equation (3.3)) can be reduced to the one-level estimate (Equation (3.5)) when several simplifying assumptions are made: square arrays ($X = Y$), approximate block sizes ($\left\lceil \frac{Y}{p} \right\rceil = \frac{Y}{p}$), constant communication cost ($rate = 0$), and no chaining ($L = 1$).

3.4.2 Serial computation cost estimation

Since the cost, c, of the inner loop computations appears in all of the final estimate expressions, it is necessary to estimate the computation costs. Performance estimation, for even serial codes, is a fairly hard problem that ideally should model the effects of the memory system, internal hardware instruction scheduling, as well as low-level compiler optimizations and is still an active area of research [90, 91]. To simplify the computation estimation problem, a basic instruction counting technique supporting simple control flow will be used [8].

Instruction counting can be performed at either the source level or assembly level. For each method, the costs of the basic operations for a given machine are expressed in terms of clock cycles. In the case of source-level estimation, the actual source code is examined and a determination is made on a line by line basis of the instructions needed to compile the code. These costs

must take into account any support instructions (address computation, register loads, stores, loop increment/decrement, etc.) performed for the actual computation. Assembly-level estimation is of course more accurate since there is no need to guess at the operations that the compiler may generate or the low-level optimizations that it might perform. However, it does require being able to perform pre-compilation of the source under examination.

In both cases, estimation of the cost of a fixed block of code (which may contain loops and other control flow structures) requires knowledge of the timing costs (cycle times obtained from microprocessor hardware manuals) of individual machine instructions in order to compute a dynamic cycle count. The extent of loop bounds (constant or variable) and the shape of the iteration space (due to functions appearing in the loop bounds) can be more easily determined at the source level while more exact information can be obtained at the assembly level. For our purposes, source level cost estimation will be used here.

3.5 Summary

In this chapter we have described several communication optimizations: message coalescing, message vectorization, message aggregation, and message pipelining. Experimental results of applying these optimizations will be presented in Chapter 6.

CHAPTER 4

OPTIMIZING DATA DISTRIBUTION

Optimizing the data distribution for a given application is a difficult task requiring careful examination of numerous tradeoffs. Since communication tends to be more expensive relative to local computation, a distribution should be selected to maintain high data locality for each processor. Excessive communication can easily offset any gains made through the use of available parallelism in the program. At the same time, the distribution should also evenly distribute the workload among the processors, making full use of the parallelism present in the computation. Since the programmer may not be (and should not have to be) aware of all the interactions between distribution decisions and communication optimizations which affect the overall performance of the program, this chapter addresses several techniques which we have developed to automate this process.

As many of the subproblems which must be solved to optimize the data distribution for a given program have been proven to be NP-complete [53, 67, 92], in the following sections we describe several effective heuristics employed to prune a considerable part of the search space while still reaching a good overall solution.

4.1 Static Performance Estimation

In order to optimize the data distribution for a program, it is necessary to first be able to estimate the performance for a given distribution. As the PARADIGM compiler has been designed to be largely machine independent, the performance of a program is estimated in terms

Table 4.1: Communication estimate costs models

Communication primitive	Base cost	Interconnection network topology: Tree	Hypercube	3-D mesh	2-D mesh	1-D array	Ring	Bus
Transfer(m)	$ovhd + rate * m$							
Shift(m,p)	Transfer(m) $*$	2						$2(p-1)$
Multicast(m,p)	Transfer(m) $*$	$\lceil \log_2 p \rceil$		$3(\lceil \sqrt[3]{p} \rceil - 1)$	$2(\lceil \sqrt{p} \rceil - 1)$	$(p-1)$	$\lceil (p-1)/2 \rceil$	1
AllMulticast(m,p)	$(p-1)$ $*$ Shift(m,p)							
Reduce(m,p)	Transfer(m) $*$	$\lceil \log_2 p \rceil$		$3(\lceil \sqrt[3]{p} \rceil - 1)$	$2(\lceil \sqrt{p} \rceil - 1)$	$(p-1)$	$\lceil (p-1)/2 \rceil$	$(p-1)$
AllReduce(m,p)	Reduce(m,p) $+$ Multicast(m,p)							
Scatter(m,p)	$(p-1)$ $*$ Transfer(m)							
Gather(m,p)	$(p-1)$ $*$ Transfer(m)							

of parameterized cost models [8,93]. For each target machine, a set of architectural parameters interfaces with the cost models effectively separating the static performance estimation framework from any one specific distributed-memory architecture.

As already mentioned in Section 3.4.2, the computation time is determined by first estimating the time for sequential execution based on a source-level dynamic count of the basic operations performed (memory load, store, index, integer and floating-point addition, multiplication, division, as well as integer modulo operation, and function calls). Parallel computation estimates are then obtained by further parameterizing the execution cost of a loop which is parallel by the number of processors over which it is partitioned.

The communication required in the program is then analyzed, taking into account several of the optimizations[1] presented in the previous chapter, to determine both the placement of communication as well as the size of the resulting message to be communicated. To estimate the resulting cost of the selected communication, the linear point-to-point transfer model, as previously given in Equation (3.1) on page 20, is used as a basis for the communication model.

The static performance estimator can also currently detect a number of different access patterns which give rise to higher level communication operations [8, 93] as shown in Table 4.1. In comparison to a basic transfer cost model, the cost of a high-level (or collective) commu-

[1]The static performance estimation framework currently does not model the overlap of communication and computation for pipelined loops resulting in conservative communication estimates for loops containing cross-iteration dependencies.

nication operation can be dependent on both the interconnect topology as well as the number of processors involved. To accurately represent the costs for each of these high-level communication operations, their cost models are ultimately based on the transfer cost model. As this approach was originally developed for hypercube based networks, we have extended it using a virtual function interface, similar to that used in the generic library interface, to make the cost estimation framework more architecture independent as well as easily extensible. Libraries such as MPI, which standardize distributed-memory programming using message passing, provide many of these common forms of collective communication. In the implementation of such a library, for a given parallel architecture, each communication operation must be written to efficiently utilize the topology in order to provide the performance shown in Table 4.1.

Note that the functions shown in Table 4.1 serve as best-case approximations of the performance of these communication operations. For architectures such as the 2-D (and 3-D) mesh, the costs also assume that the geometry of the physically allocated processors has an aspect ratio of 1:1(:1) independent of the logical arrangement of the abstract mesh. If more detailed modeling is required [94], the functions could be extended to accept the number of processors in each dimension of the abstract mesh to take into account the effect of the physical processor mapping. For multidimensional architectures in which there is more than one processor in a given dimension (i.e., a 2-D or 3-D mesh) this would improve the accuracy of the costs when processor subsets are involved in an operation, e.g., a broadcast along one dimension of a 2-D data distribution mapped onto a 2-D mesh is actually $(p-1)*\mathrm{Transfer}(m)$ not $2(\lceil\sqrt{p}\rceil-1)*\mathrm{Transfer}(m)$ whereas it is always $\lceil\log p\rceil * \mathrm{Transfer}(m)$ when mapped onto a hypercube.

With the exception of the architecture-specific costs (the basic computation operation costs and the communication parameters $ovhd$ and $rate$), these cost models allow the partitioning techniques to be relatively machine independent.

4.2 Static Distribution Selection

In the PARADIGM compiler, static data partitioning decisions are made in a number of distinct phases [8, 57]. Often, there is a tradeoff between minimizing interprocessor communica-

tion and exploiting all available parallelism; the communication and the computational costs imposed by the underlying architecture must both be taken into account. Below are brief descriptions of each of the major phases performed during the static data partitioning pass.

4.2.1 Alignment

The alignment pass identifies the array dimensions that should be mapped to the same processor mesh dimension. The alignment preferences between two arrays can be between different pairings of dimensions (interdimensional or axis alignment) as well as by an offset or stride within a given pair of dimensions (intradimensional or stride alignment). Currently, only interdimensional alignment is performed in the partitioning pass.

4.2.2 Block/cyclic distribution

Once array alignment has been performed, the distribution pass determines whether each array dimension should be distributed in a blocked or cyclic manner. Array dimensions are first classified by their communication requirements. If the communication in a mesh dimension is recognized as a nearest-neighbor pattern, it indicates the need for a blocked distribution. For dimensions that are only partially traversed (less than a certain threshold), a cyclic distribution may be more desirable for load balancing. Using alignment information from the previous phase, the array dimensions that cross-reference each other are assigned the same kind of partitioning to ensure the intended alignment.

4.2.3 Block size selection

When a cyclic distribution is chosen, the compiler is able to make further adjustments on the block size giving rise to block-cyclic partitionings. Since only a cyclic distribution is chosen in the previous phase, in order to improve the load balance for partially traversed array dimensions, a closer examination of the communication costs must be performed. This analysis is sometimes needed when arrays are used to simulate recordlike structures (not supported di-

rectly in FORTRAN 77) or when lower-dimensional arrays play the role of higher-dimensional arrays.

4.2.4 Mesh configuration

After all the distribution parameters have been determined, the cost estimates are functions of only the number of processors in each mesh dimension. For each set of aligned array dimensions, the compiler determines if there are any parallel operations performed. If no parallelism exists in a given dimension, it is collapsed onto a single processor. If there is only one dimension that has not been collapsed, all processors are assigned to this dimension. In the case of multiple dimensions of parallelism, the compiler determines the best arrangement of processors by evaluating the cost expression to estimate execution time for each feasible configuration.

4.3 Dynamic Distribution Selection

For complex programs we have seen that static data distributions may be insufficient to obtain acceptable performance. Static distributions suffer in that they cannot reflect changes in a program's data access behavior. When conflicting data requirements are present, static partitionings tend to be compromises between a number of preferred distributions. Instead of requiring a single data distribution for the entire execution, program data could also be redistributed dynamically for different *phases*[2] of the program. Such dynamic partitionings can yield higher performance than for a static partitioning when the redistribution is more efficient than the communication pattern required by the statically partitioned computation.

4.3.1 Motivation for dynamic distributions

Figure 4.1 shows the basic computation performed in a two-dimensional Fast Fourier Transform (FFT). To execute this program in parallel on a machine with distributed memory, the main

[2] A *phase* can be described simply as a sequence of statements in a program over which a given distribution is unchanged.

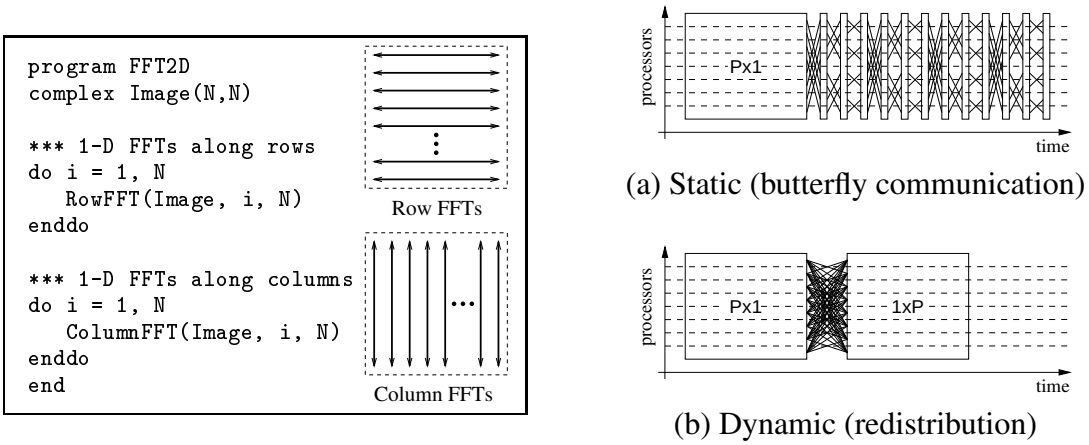


Figure 4.1: Two-dimensional Fast Fourier Transform

data array, `Image`, is partitioned across the available processors. By examining the data accesses that will occur during execution, it can be seen that, for the first half of the program, data is manipulated along the rows of the array. For the rest of the execution, data is manipulated along the columns. Depending on how data is distributed among the processors, several different patterns of communication could be generated. The goal of automatic data partitioning is to select the distribution that will result in the highest level of performance.

If the array were distributed by rows, every processor could independently compute the FFTs for each row that involved local data. After the rows had been processed, the processors would now have to communicate to perform the column FFTs as the columns have been partitioned across the processors. Conversely, if a column distribution were selected, communication would be required to compute the row FFTs while the column FFTs could be computed independently. Such static partitionings, as shown in Figure 4.1(a), suffer in that they cannot reflect changes in a program's data access behavior. When conflicting data requirements are present, static partitionings tend to be compromises between a number of preferred distributions.

Instead of requiring a single data distribution for the entire execution, program data could also be redistributed dynamically for different phases of the program. For this example, assume the program is split into two separate phases; a row distribution is selected for the first phase and a column distribution for the second (as shown in Figure 4.1(b)). By redistributing the data between the two phases, none of the one-dimensional FFT operations would require

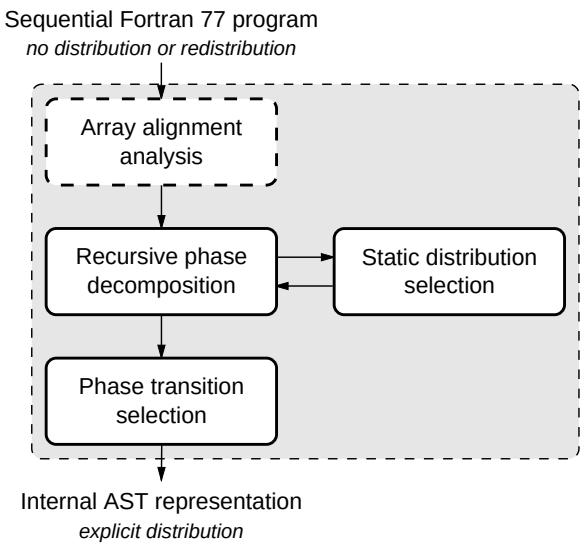
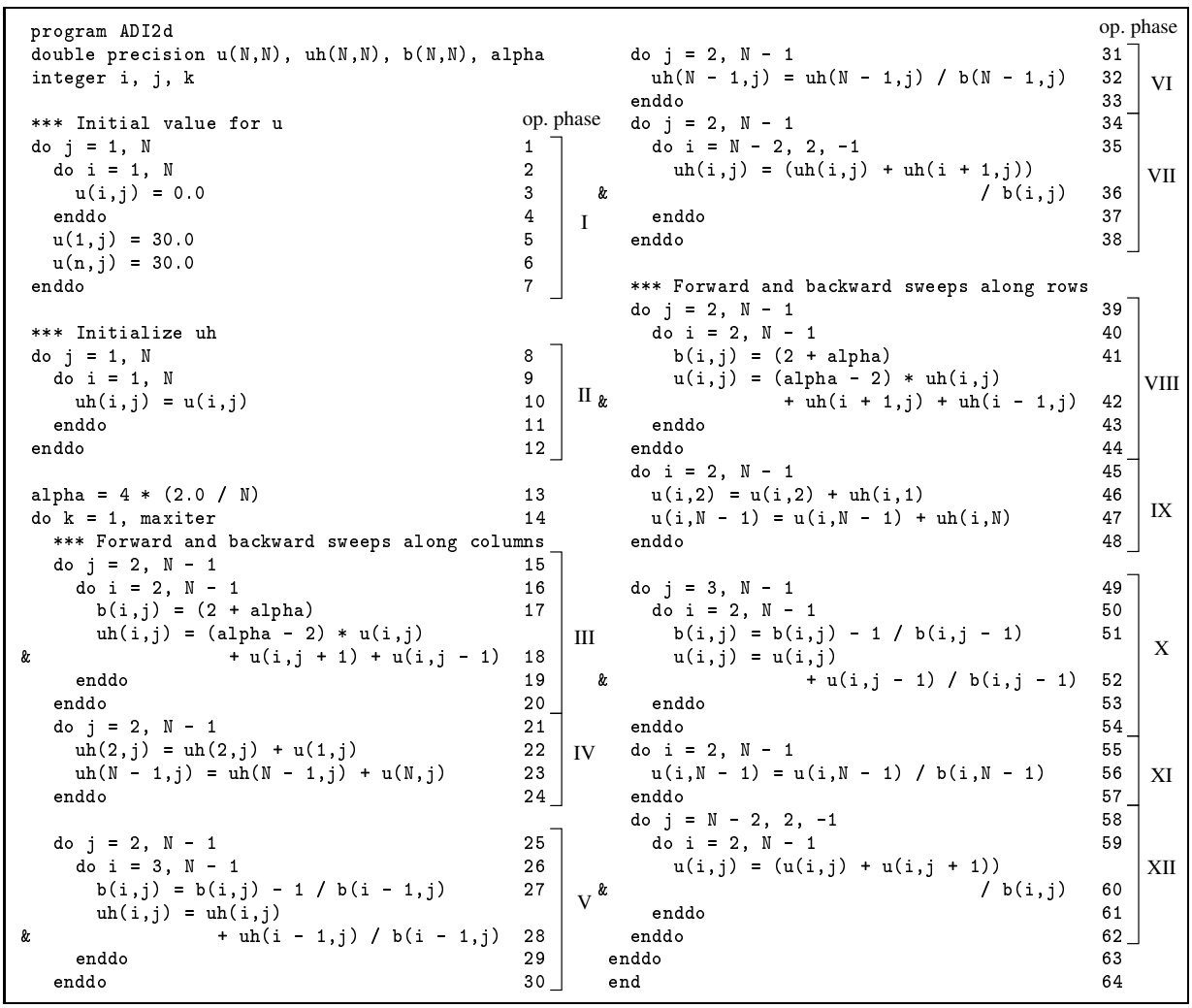


Figure 4.2: Overview of the dynamic distribution approach

communication. Such dynamic partitionings can yield higher performance than a static partitioning when the redistribution is more efficient than the communication pattern required by the statically partitioned computation.

4.3.2 Overview of the dynamic distribution approach

The approach we have developed to automatically select dynamic distributions, shown in Figure 4.2, consists of two main steps which can be preceded by an initial analysis of array alignments. First, in Section 4.3.3, we will describe how to recursively decompose the program into a hierarchy of candidate phases obtained using existing static distribution techniques. Then, in Section 4.3.4 we will describe how to select the most efficient sequence of phases and phase transitions taking into account the cost of redistributing the data between the different phases. To avoid performing redundant work within the static partitioner, it would also be possible to decouple the analysis of array alignment preferences and the selection of the actual alignments by using an appropriate data structure which will described in Section 4.3.5. This approach allows us to build upon the static partitioning techniques previously described in Section 4.2.



```
program ADI2d
double precision u(N,N), uh(N,N), b(N,N), alpha
integer i, j, k

*** Initial value for u                                  op. phase
do j = 1, N                                              1
  do i = 1, N                                            2
    u(i,j) = 0.0                                         3
  enddo                                                  4
  u(1,j) = 30.0                                          5
  u(n,j) = 30.0                                          6
enddo                                                    7

*** Initialize uh
do j = 1, N                                              8
  do i = 1, N                                            9
    uh(i,j) = u(i,j)                                     10
  enddo                                                  11
enddo                                                    12

alpha = 4 * (2.0 / N)                                    13
do k = 1, maxiter                                        14
  *** Forward and backward sweeps along columns
  do j = 2, N - 1                                        15
    do i = 2, N - 1                                      16
      b(i,j) = (2 + alpha)                               17
      uh(i,j) = (alpha - 2) * u(i,j)
&                 + u(i,j + 1) + u(i,j - 1)              18
    enddo                                                19
  enddo                                                  20
  do j = 2, N - 1                                        21
    uh(2,j) = uh(2,j) + u(1,j)                           22
    uh(N - 1,j) = uh(N - 1,j) + u(N,j)                   23
  enddo                                                  24

  do j = 2, N - 1                                        25
    do i = 3, N - 1                                      26
      b(i,j) = b(i,j) - 1 / b(i - 1,j)                   27
      uh(i,j) = uh(i,j)
&                 + uh(i - 1,j) / b(i - 1,j)             28
    enddo                                                29
  enddo                                                  30
```

I

II

III

IV

V

```
                                                         op. phase
  do j = 2, N - 1                                        31
    uh(N - 1,j) = uh(N - 1,j) / b(N - 1,j)               32
  enddo                                                  33
  do j = 2, N - 1                                        34
    do i = N - 2, 2, -1                                  35
      uh(i,j) = (uh(i,j) + uh(i + 1,j))
&                                 / b(i,j)               36
    enddo                                                37
  enddo                                                  38

  *** Forward and backward sweeps along rows
  do j = 2, N - 1                                        39
    do i = 2, N - 1                                      40
      b(i,j) = (2 + alpha)                               41
      u(i,j) = (alpha - 2) * uh(i,j)
&                 + uh(i + 1,j) + uh(i - 1,j)            42
    enddo                                                43
  enddo                                                  44
  do i = 2, N - 1                                        45
    u(i,2) = u(i,2) + uh(i,1)                            46
    u(i,N - 1) = u(i,N - 1) + uh(i,N)                    47
  enddo                                                  48

  do j = 3, N - 1                                        49
    do i = 2, N - 1                                      50
      b(i,j) = b(i,j) - 1 / b(i,j - 1)                   51
      u(i,j) = u(i,j)
&                 + u(i,j - 1) / b(i,j - 1)              52
    enddo                                                53
  enddo                                                  54
  do i = 2, N - 1                                        55
    u(i,N - 1) = u(i,N - 1) / b(i,N - 1)                 56
  enddo                                                  57
  do j = N - 2, 2, -1                                    58
    do i = 2, N - 1                                      59
      u(i,j) = (u(i,j) + u(i,j + 1))
&                                 / b(i,j)               60
    enddo                                                61
  enddo                                                  62
enddo                                                    63
end                                                      64
```

VI

VII

VIII

IX

X

XI

XII

Figure 4.3: 2-D Alternating Direction Implicit iterative method (ADI2D) (shown with operational phases)

To help illustrate the dynamic partitioning technique, an example program will be used. In Figure 4.3, a two-dimensional Alternating Direction Implicit iterative method[3] (ADI2D) is shown, which computes the solution of an elliptic partial differential equation known as Poisson's equation [95]. Poisson's equation can be used to describe the dissipation of heat away from a surface with a fixed temperature as well as to compute the free-space potential created by a surface with an electrical charge.

[3]To simplify later analysis of performance measurements, the program shown performs an arbitrary number of iterations as opposed to periodically checking for convergence of the solution.

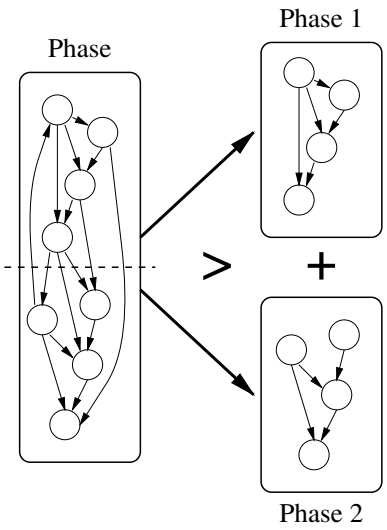


Figure 4.4: Phase decomposition

For the program in Figure 4.3, a static data distribution will incur a significant amount of communication for over half of the program's execution. For illustrative purposes only, the operational definition of phases previously described in Section 2.2 identifies twelve different "phases" in the program. These phases exposed by the operational definition need not be known for our technique (and, in general, are potentially too restrictive) but they will be used here for comparison as well as to facilitate the discussion.

4.3.3 Phase decomposition

Initially, the entire program is viewed as a single phase for which a static distribution is determined. At this point, the immediate goal is to determine if and where it would be beneficial to split the program into two separate phases such that the sum of the execution times of the resulting phases is less than the original (as illustrated in Figure 4.4). Using the selected distribution, a *communication graph* is constructed to examine the cost of communication in relation to the flow of data within the program.

We define a *communication graph* as the flow information from the dependence graph (generated by Parafrase-2 [6]) weighted by the cost of communication. The nodes of the communication graph correspond to individual statements while the edges correspond to flow dependen-

cies that exist between the statements. As a heuristic, the cost of communication performed for a given reference in a statement is initially assigned to (*reflected* back along) every incoming dependence edge corresponding to the reference involved. Since flow information is used to construct the communication graph, the weights on the edges serve to expose communication costs that exist between producer/consumer relationships within a program. Also since we restrict the granularity of phase partitioning to the statement level, single node cycles in the flow dependence graph are not included in the communication graph.

After the initial costs have been assigned to the communication graph, they are scaled according to the number of outgoing edges to which each reference was assigned. The scaling conserves the total cost of a communication operation for a given reference, ref, at the consumer, j, by assigning portions to each producer, i, proportional to the dynamic execution count of the given producer, i, divided by the dynamic execution counts of all producers. Note that the scaling factors are computed separately for producers which are predecessor or successors of the consumer as shown in Equation (4.2). Once the individual edge costs have been scaled to conserve the total communication cost, they are propagated back toward the start of the program (through all edges to producers which are predecessors) while still conserving the propagated cost as shown in Equation (4.3). Also, to further differentiate between producers at different nesting levels, all scaling factors are also scaled by the ratio of the nesting levels as shown in Equation (4.1).

Scaling initial costs:

$$lratio(i,j) = \frac{nestlevel(i)+1}{nestlevel(j)+1} \tag{4.1}$$

$$W(i,j,ref) = \frac{dyncount(i)}{\sum_{\substack{i<j \ P \in in\_pred(j,ref) \\ i>j \ P \in in\_succ(j,ref)}} dyncount(P)} \cdot lratio(i,j) \cdot comm(i,ref) \tag{4.2}$$

Propagating costs:

$$W(i,j,ref) = W(i,j,ref) + \frac{dyncount(i)}{\sum_{P \in in\_pred(j,*)} dyncount(P)} \cdot lratio(i,j) \sum_{P \in out(j)} W(j,P,*) \tag{4.3}$$

In the actual implementation, this is accomplished in two passes over the communication graph after assigning the initial communication costs: the first to compute all the scaling factors and the second to propagate costs back toward the start of the program.

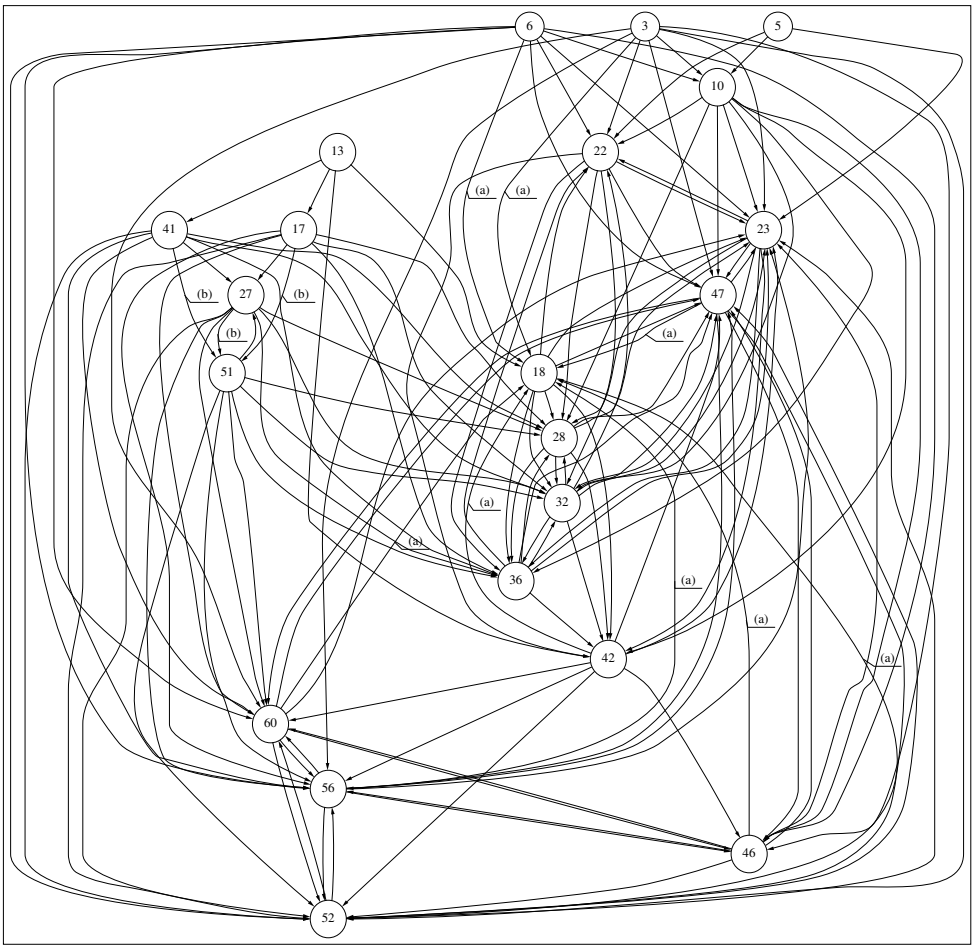
In Figure 4.5, the communication graph is shown for ADI2D with some of the edges labeled with the expressions automatically generated by the static cost estimator (using a problem size of 512×512 and `maxiter` set to 100). For reference, the communication models for an Intel Paragon and a Thinking Machines CM-5, corresponding to the communication primitives used in the cost expressions, are shown in Table 4.2. Conditionals appearing in the cost expressions represent costs that will be incurred based on specific distribution decisions (e.g., $P_2 > 1$ is true if the second mesh dimension is assigned more than one processor).

Once the communication graph has been constructed, a split point is determined by computing a maximal cut of the communication graph. The maximal cut removes the largest communication constraints from a given phase to potentially allow better individual distributions to be selected for the two resulting split phases. Since we also want to ensure that the cut divides the program at exactly one point to ensure only two subphases are generated for the recursion, only cuts between two successive statements will be considered. Since the ordering of the nodes is related to the linear ordering of statements in a program, this guarantees that the nodes on one side of the cut will always all precede or all follow the node most closely involved in the cut. The following algorithm is used to determine which cut to use to split a given phase.

For simplicity of this discussion, assume for now that there is at most only one edge between any two nodes. For multiple references to the same array, the edge weight can be considered to be the sum of all communication operations for that array. Also, to better describe the algorithm, view the communication graph $G = (V, E)$ in the form of an adjacency matrix (with source vertices on rows and destination vertices on columns).

(1) For each statement S_i $\{i \in [1, (|V| - 1)]\}$ compute the cut of the graph between statements S_i and S_{i+1} by summing all the edges in the submatrices specified by $[S_1, S_i] \times [S_{i+1}, S_{|V|}]$ and $[S_{i+1}, S_{|V|}] \times [S_1, S_i]$

(2) While computing the cost of each cut also keep track of the current maximum cut.



(a) $100 * (P_2 > 1) *$ Shift(510)
(b) $3100 *$ Transfer(510)

Figure 4.5: Communication graph and some example initial edge costs for ADI2D (Statement numbers correspond to Figure 4.3)

Table 4.2: Communication primitives

	Intel Paragon	TMC CM-5
Transfer(m)	$50 + 0.018m$	$23+0.12m \quad m \leq 16$ $86+0.12m \quad m > 16$
Shift(m)	$2 *$ Transfer(m)	

(3) If there is more than one cut with the same maximum value, choose from this set the cut that separates the statements at the highest nesting level. If there is more than one cut with the same highest nesting level, record the earliest and latest maximum cuts with that nesting level (forming a cut window).

(4) Split the phase using the selected cut.

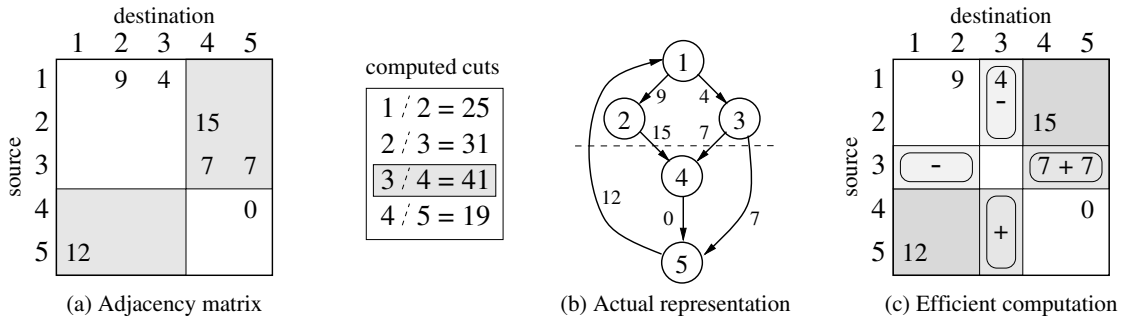


Figure 4.6: Example graph illustrating the computation of a cut

In Figure 4.6, the computation of the maximal cut on a smaller example graph with arbitrary weights is shown. The maximal cut is found to be between vertices 3 and 4 with a cost of 41. This is shown both in the form of the sum of the two adjacency submatrices in Figure 4.6(a), and graphically as a cut on the actual representation in Figure 4.6(b).

In Figure 4.6(c), the cut is again illustrated using an adjacency matrix, but the computation is shown using a more efficient implementation which only adds and subtracts the differences between two successive cuts using a running cut total while searching for the maximum cut in sequence. This implementation also provides much better locality than the full submatrix summary when analyzing the actual sparse representation since the differences between two successive cuts can be easily obtained by traversing the incoming and outgoing edge lists (which correspond to columns and rows in the adjacency matrix respectively) of the node immediately preceding the cut. This takes $\mathcal{O}(E)$ time on the actual representation, only visiting each edge twice – once to add it and once to subtract it.

A new distribution is selected for each of the resulting phases inheriting any unspecified distributions (due to an array not appearing in a subphase) from the parent phase. The process is

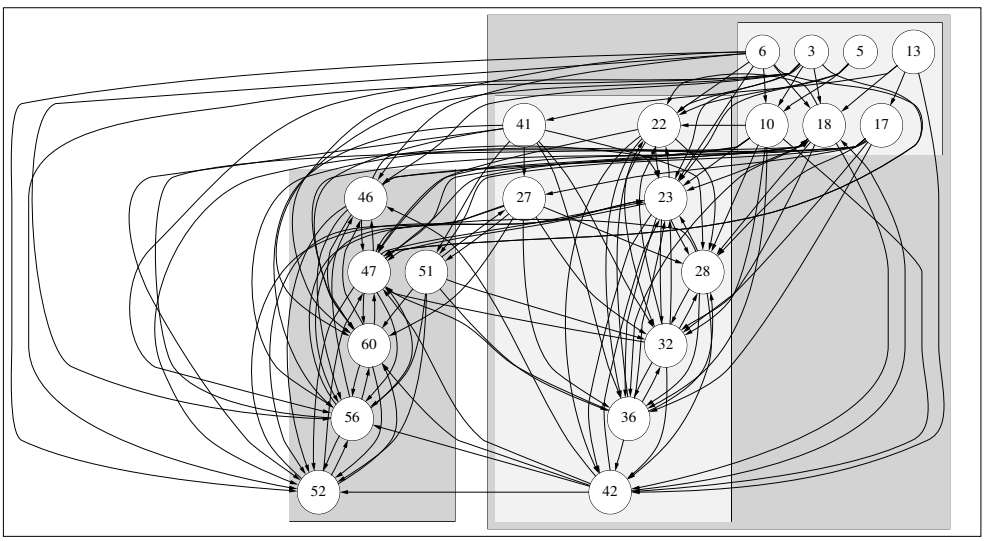


Figure 4.7: Partitioned communication graph for ADI2D
(Statement numbers correspond to Figure 4.3.)

then continued recursively using the costs from the newly selected distributions corresponding to each subphase. As shown in Figure 4.4, each level of the recursion is carried out in branch and bound fashion such that a phase is split only if the sum of the estimated execution times of the two resulting phases shows an improvement over the original.[4] In Figure 4.7, the partitioned communication graph is shown for ADI2D after the phase decomposition is completed.

As mentioned in the cut algorithm, it is also possible to find several cuts which all have the same maximum value and nesting level forming a window over which the cut can be performed. This can occur since not all statements generate communication resulting in either edges with zero cost or regions over which the propagated costs conserve edge flow, both of which will maintain a constant cut value. To handle cut windows, the phase should be split into two subphases such that the lower subphase uses the earliest cut point and the upper subphase uses the latest, resulting in overlapping phases. After new distributions are selected for each overlapping subphase, the total cost of executing the overlapped region in each subphase is examined. The overlap is then assigned to the subphase that resulted in the lowest execution time for this region. If they are equivalent, the overlapping region can be equivalently assigned to either subphase.

[4] A further optimization can also be applied to bound the size of the smallest phase that can be split by requiring its estimated execution time to be greater than a "minimum cost" of redistribution.

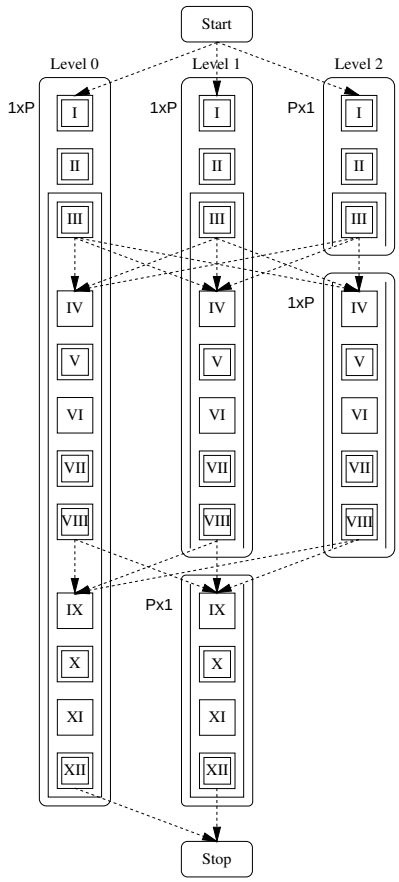
Currently, this technique is not yet implemented for cut windows. We instead always select the earliest cut point in a window for the partitioning.

To be able to bound the depth of the recursion without ignoring important phases and distributions, the static partitioner must also obey the following property. A partitioning technique is said to be *monotonic* if it selects the best available partition for a segment of code such that (aside from the cost of redistribution) the time to execute a code segment with a selected distribution is less than or equal to the time to execute the same segment with a distribution that is selected after another code segment is appended to the first. In practice, this condition is satisfied by the static partitioning algorithm that we are using. This can be attributed to the fact that conflicts between distribution preferences are not broken arbitrarily, but are resolved based on the costs imposed by the target architecture [57].

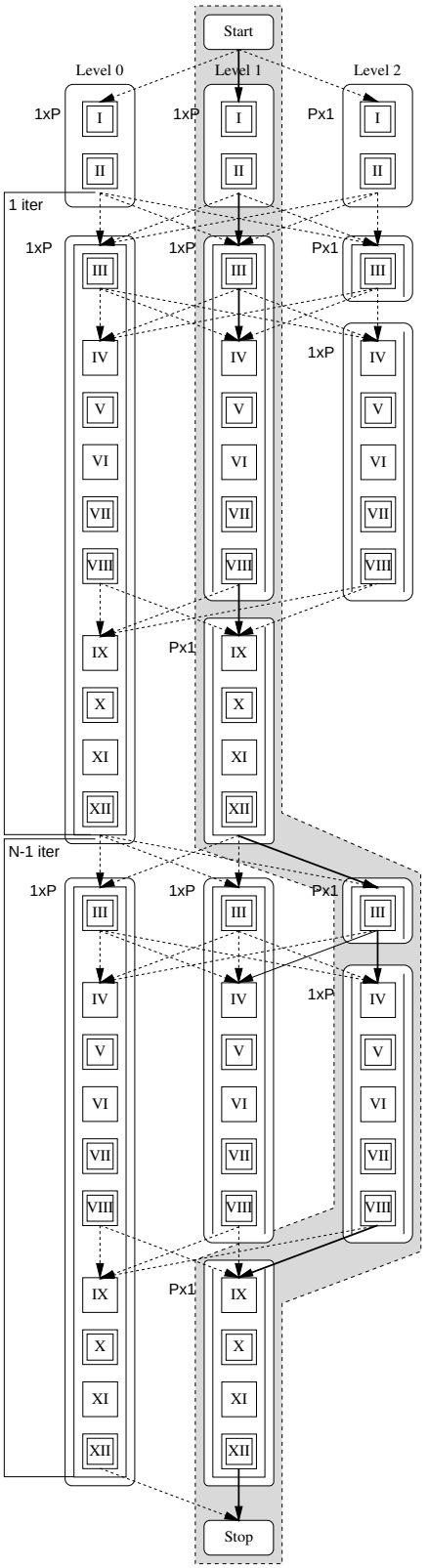
It is also interesting to note that if a cut occurs within a loop body, and loop distribution can be performed, the amount of redistribution can be greatly reduced by lifting it out of the distributed loop body and performing it in between the two sections of the loop. Also, if dependencies allow statements to be reordered, statements may be able to move across a cut boundary without affecting the cost of the cut while possibly reducing the amount of data to be redistributed. Both of these optimizations can be used to reduce the cost of redistribution but neither will be examined in this thesis.

4.3.4 Phase and phase transition selection

After the program has been recursively decomposed into a hierarchy of phases, a Phase Transition Graph (PTG) is constructed. Nodes in the PTG are phases resulting from the decomposition while edges represent possible redistribution between phases as shown in Figure 4.8(a). Since it is possible that using lower level phases may require transitioning through distributions found at higher levels (to keep the overall redistribution costs to a minimum), the phase transition graph is first sectioned across phases at the granularity of the lowest level of the phase



(a) Initial phase transition graph



(b) After performing loop peeling

Figure 4.8: Phase transition graph for ADI2D

decomposition[5] Redistribution costs are then estimated [84] for each edge and are weighted by the execution count of the surrounding code.

If a redistribution edge occurs within a loop structure, additional redistribution may be induced due to the control flow of the loop. To account for a potential "reverse" redistribution which can occur on the back edge of the iteration, the phase transition graph is partitioned around the loop body and the first iteration of loop containing the phase transition is peeled off and the phases of the first iteration of the body re-inserted in the phase transition graph as shown in Figure 4.8(b). Redistribution within the peeled iteration is only executed once while that within the remaining loop iterations is now executed $(N-1)$ times, where N is the number of iterations in the loop. The redistribution, which may occur between the first peeled iteration and the remaining iterations, is also multipled by $(N-1)$ in order to model when the back edge causes redistribution (i.e., when the last phase of the peeled iteration has a different distribution than the first phase of the remaining one).

Once costs have been assigned to all redistribution edges, the best sequence of phases and phase transitions is selected by computing the shortest path on the phase transition graph. This is accomplished in $\mathcal{O}(V^2)$ time (where V is now the number of vertices in the phase transition graph) using Dijkstra's single source shortest path algorithm [96].

After the shortest path has been computed, the loop peeling performed on the PTG can be seen to have been necessary to obtain the best solution if the peeled iteration has a different transition sequence than the remaining iterations. Even if the peeled iteration does have different transitions, not actually performing loop peeling on the actual code will only incur at most one additional redistribution stage upon entry to the loop nest. This will not overly affect performance if the execution of the entire loop nest takes significantly longer than a single redistribution operation, which is usually the case especially if the redistribution considered within the loop was actually accepted when computing the shortest path.

Using the cost models for an Intel Paragon and a Thinking Machines CM-5, the distributions and estimated execution times reported by the static partitioner for the resulting phases

[5] Sectioned phases that have identical distributions within the same horizontal section of the PTG are actually now redundant and can be removed, if desired, without affecting the quality of the final solution.

Table 4.3: Detected phases and estimated execution times (sec) for ADI2D (Performance estimates correspond to 32 processors.)

Op. Phases(s)	Distribution	Intel Paragon	TMC CM-5	
I-XII	*,BLOCK 1×32	22.151461	39.496276	Level 0
I-VIII	*,BLOCK 1×32	1.403644	2.345815	Level 1
IX-XII	BLOCK,* 32×1	0.602592	0.941550	
I-III	BLOCK,* 32×1	0.376036	0.587556	Level 2
IV-VIII	*,BLOCK 1×32	0.977952	1.528050	

described as ranges of operational phases are shown in Table 4.3. The performance parameters of the two machines are similar enough that the static partitioning actually selects the same distribution at each phase for each machine. The times estimated for the static partitioning are slightly higher than those actually observed, resulting from a conservative assumption regarding pipelines[6] made by the static cost estimator [8], but they still exhibit similar enough performance trends to be used as estimates. For both machines, the cost of performing redistribution is low enough in comparison to the estimated performance gains that a dynamic distribution scheme is selected, as shown by the shaded area in Figure 4.8(b).

Pseudo-code for the dynamic partitioning algorithm is presented in Figure 4.9 to briefly summarize both the phase decomposition and phase transition selection procedures as described. As distributions for a given phase are represented as a set[7] of variables, each of which having an associated distribution, a masking union set operation is used to inherit unspecified distributions ($dist \uplus dist_i$). A given variable's distribution in the $dist$ set will be replaced if it also has a distribution in the $dist_i$ set thus allowing any unspecified distributions in subphase i ($dist_i$) to be inherited from its parent ($dist$).

Since the use of inheritance during the phase decomposition process implicitly maintains the coupling between individual array distributions, redistribution at any stage will only affect

[6] Initially, a BLOCK, BLOCK distribution was selected by the static partitioner for (only) the first step of the phase decomposition. As the static performance estimation framework does not currently take into account any overlap between communication and computation for pipelined computations we decided that this decision was due to the conservative performance estimate. For the analysis presented for ADI2D, we bypassed this problem by temporarily restricting the partitioner to only consider 1-D distributions.

[7] As will be more rigorously defined later in Section 5.1.3.

```
PARTITION(program)
 1  cutlist ← ∅
 2  dist ← STATIC-PARTITIONING(program)
 3  phases ← DECOMPOSE-PHASE(program, dist, cutlist)
 4  ptg ← SELECT-REDISTRIBUTION(phases, cutlist)
 5  Assign distributions based on shortest phase recorded in ptg

DECOMPOSE-PHASE(phase, dist, cutlist)
 1  Add phase to list of recognized phases
 2  Construct the communication graph for the phase
 3  cut ← MAX-CUT(phase)
 4  if VALUE(cut) = 0     ▷ No communication in phase
 5    then return
 6  Relocate cut to highest nesting level of identical cuts
 7  phase_1, phase_2 ←(cut)— phase
 8  ▷ Note: if cut is a window, phase_1 and phase_2 will overlap
 9  dist_1 ← STATIC-PARTITIONING(phase_1)
10  dist_2 ← STATIC-PARTITIONING(phase_2)
11  ▷ Inherit any unspecified distributions from parent
12  dist_1 ← dist ⟗ dist_1
13  dist_2 ← dist ⟗ dist_2
14  if (cost(phase_1) + cost(phase_2)) < cost(phase)
15    then ▷ If cut is a window, phase_1 and phase_2 overlap
16         if LAST_STMTNUM(phase_1) > FIRST_STMTNUM(phase_2)
17           then RESOLVE-OVERLAP(cut, phase_1, phase_2)
18         LIST-INSERT(cut, cutlist)
19         phase→left = DECOMPOSE-PHASE(phase_1, dist_1, cutlist)
20         phase→right = DECOMPOSE-PHASE(phase_2, dist_2, cutlist)
21    else phase→left = NULL
22         phase→right = NULL
23  return (phase)

SELECT-REDISTRIBUTION(phases, cutlist)
 1  if cutlist = ∅
 2    then return
 3  ptg ← CONSTRUCT-PTG(phases, cutlist)
 4  Divide ptg horizontally at the recursion lowest level
 5  for each loop in phases
 6      do if loop contains a cut at its nesting level
 7           then Divide ptg at loop boundaries
 8                PEEL(loop, ptg)
 9  Estimate the interphase redistribution costs for ptg
10  Compute the shortest phase transition path on ptg
11  return (ptg)
```

Figure 4.9: Pseudo-code for the partitioning algorithm

the next stage. This can be contrasted to the technique proposed by Bixby, Kennedy, and Kremer [66] which first selects a number of partial candidate distributions for each phase specified by the operational definition. Since their phase boundaries are chosen in the absence of flow information, redistribution can affect stages at any distance from the current stage. This causes the redistribution costs to become binary functions depending on whether or not a specific path is taken, therefore, necessitating the need for 0-1 integer programming. In [67] they do agree, however, that 0-1 integer programming is not necessary when all phases specify complete distributions (such as true in our case). In their work, this occurs only as a special case in which they specify complete phases from the innermost to outermost levels of a loop nest. For this situation they show how the solution can be obtained using a hierarchy of single source shortest path problems in a bottom-up fashion (as opposed to solving only one shortest path problem after performing a top-down phase decomposition as in our approach).

Up until now, we have not described how to handle control flow other than for loop constructs. More general flow (caused by conditionals or branch operations) can be viewed as separate paths of execution with different frequencies of execution. The same techniques that have been used for scheduling assembly level instructions by selecting traces of interest [97] or forming larger blocks from sequences of basic blocks [98] in order to optimize the most frequently taken paths can also be applied to the phase transition. Once a single trace has been selected (using profiling or other criteria) its phases are obtained using the phase selection algorithm previously described but ignoring all code off the main trace. Once phases have been selected, all off-trace paths can be optimized separately by first setting their stop and start nodes to the distributions of the phases selected for the points at which they exit and re-enter the main trace. Each off-trace path can then be assigned phases by applying the phase selection algorithm to each path individually. Although this specific technique is not currently implemented in the compiler, but will be addressed in future work, other researchers have also been considering it as a feasible solution for selecting phase transitions in the presence of general control flow [99].

4.3.5 Hierarchical Component Affinity Graph (HCAG)

To further extend the partitioning algorithms to perform interprocedural analysis as well as facilitate the computation of static alignments and communication cost estimates within a specified region of the program, we have also developed a hierarchical form of the extended component affinity graph (CAG) [16] as currently used in the static partitioner [57]. Since alignment preferences are only affected by the relationship of the subscript expressions between the left- and right-hand sides of an assignment statement, this information does not change with respect to the current distribution under consideration. Rather than computing an entire affinity graph for each new phase revealed during the partitioning process, the HCAG can be computed once for the entire program and reused throughout the analysis. This is achieved by precomputing intermediate alignment graphs and storing them for later use within the levels created by compound nodes (e.g., 'loop' or 'if' statements) in the program's abstract syntax tree (AST).

The HCAG is constructed in a single recursive traversal of the AST performing these operations:

(1) **pre-order:** record individual alignment preferences for the statement

(2) **in-order:** store the current state of the affinity graph and then set it to NULL (for compound nodes)

(3) **post-order:** merge the statement's alignment preferences with the current state of the affinity graph.

The graph that is stored at the root of the AST corresponds to the CAG for the entire program whereas each compound statement header will contain the CAG summarizing all statements within the corresponding subtree. Now, all that is required to obtain the CAG for a given phase is to simply merge the HCAG components from only the topmost level of the region of the AST containing the given phase. This saves a considerable amount of time which would otherwise be required to obtain the CAG when examining each new phase.

Furthermore, by also recursively traversing on the call graph for function and subroutine calls encountered during the traversal, the HCAG can also be used to perform interprocedu-

ral alignment analysis. Upon encountering the first call site for a given function, the HCAG is computed for the function as previously described and stored retaining all parameters symbolically in the alignment cost expressions. At all following call sites, the stored HCAG is simply retrieved and the current values of the parameters substituted in the cost expressions. This technique ignores any interprocedural side effects but, unless the access patterns are highly side-effect dependent, it will still accurately model access patterns across function calls.

In Figure 4.10, the HCAG is shown for ADI2D. For clarity, only a few of the alignment costs are labeled and only the statements corresponding to loop headers are shown. If all statements were shown in the AST, statements contained within a loop body would appear as a connected sequence of right children attached as the left child of the loop header. In the AST, the body of a compound statement is attached as the left child while the right child follows statements in sequence. In the figure, intermediate affinity graphs are illustrated using a small inset at each level of the HCAG while the different nesting levels of the HCAG are indicated by shaded outlines. The affinity graphs represent each dimension of an array as a separate node while alignment preferences (with an associated cost which is incurred if the alignment is not obeyed) connect different dimensions of the arrays.

One thing to notice for ADI2D is how most of the intermediate affinity graphs are identical at the two innermost levels of the HCAG. This is due to the fact that the innermost two loops in ADI2D are almost always perfectly nested causing the outer loop to not contribute any further alignment information than that already obtained from within the inner loop. The structures of the top two levels of this HCAG are also identical since the preferences of the two intermediate graphs merged to form the topmost level were already present in both subgraphs. It is important to note that even though the structure is the same for the top two levels, the alignment costs are actually different. Even though there aren't any alignment conflicts in this example, one last observation to make is how useful the HCAG can be for analyzing code in which an alignment conflict may exist. If a conflict exists when considering a large portion of the program but disappears when smaller regions are examined, appropriate alignments can be efficiently obtained for each of the competing regions by using the HCAG in combination with the techniques described in the previous section.

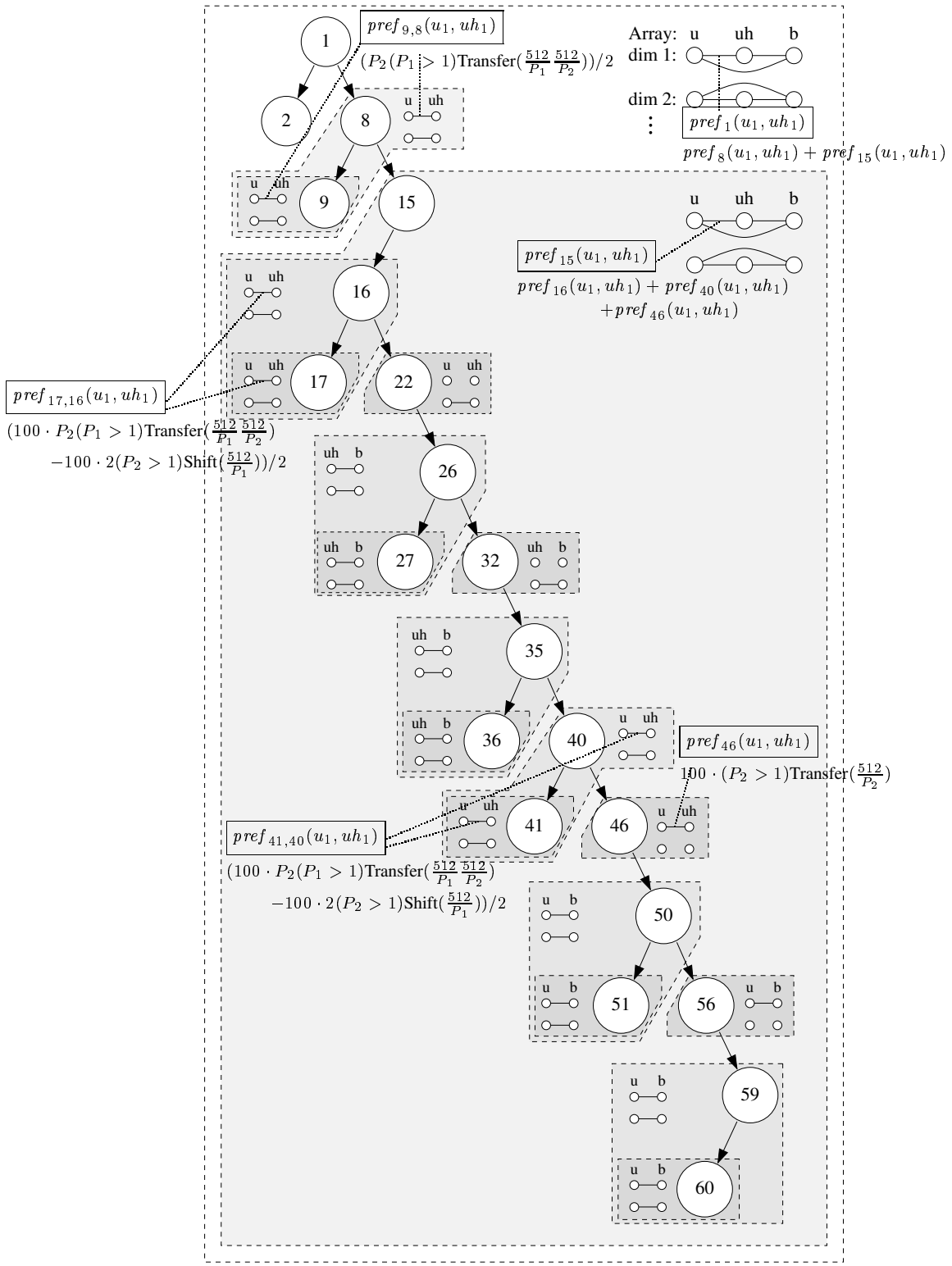


Figure 4.10: Hierarchical Component Affinity Graph (HCAG) for ADI2D
(Statement numbers correspond to Figure 4.3)

As there are still other implementation issues within the static partitioner which remain to be resolved before it can perform interprocedural partitioning, the HCAG has not been implemented in the compiler. One possibility in particular, is to integrate this data structure with the Hierarchical Task Graph (HTG) [100] already constructed by Parafrase-2. As this would give us the opportunity to study techniques for automatically selecting both data and task distributions in a unified framework, this is beyond the scope of this thesis and will be the focus of future work in this area.

4.4 Summary

In this chapter we have described the techniques we have developed for optimizing data distributions for distributed-memory multicomputers. First, the program is recursively decomposed into a hierarchy of candidate phases. Then, taking into account the cost of redistributing the data between the different phases, the most efficient sequence of phases and phase transitions is selected. By basing these techniques upon the existing algorithms previously implemented in PARADIGM to obtain static data distributions, the synergy of the combined approach allows us to select either static or dynamic data distributions in unified framework. An experimental evaluation of the resulting data distributions will be presented later in Chapter 6.

CHAPTER 5

OPTIMIZING DATA REDISTRIBUTION

In this chapter, we present an interprocedural data-flow framework which provides a means to convert between a distribution-based representation (as specified by the data partitioner and required for compilation) and redistribution-based form (as specified by HPF) in order to output the selected distributions in the form of an HPF program. Initially, this was all that we set out to do, but quickly discovered that it was also possible to use the same framework in reverse to convert a redistribution-based representation (an HPF program) into a distribution-based form. Not only is this necessary step in order to compile dynamic HPF programs, but as will be seen in this chapter, it is also possible to optimize the amount of redistribution that must be performed during this transformation by removing unnecessary redistribution specified by the programmer as well as that due to the HPF semantics of function calls. As will described in more detail in Section 5.2.3, this framework also allows us to convert certain dynamic HPF programs into equivalent static versions through a process we refer to as Static Distribution Assignment (SDA) providing dynamic HPF support for existing subset HPF compilers.

In HPF (as previously mentioned in Chapter 1), dynamic distributions can be described explicitly using executable redistribution directives (`REDISTRIBUTE` or `REALIGN`, which specify where new distributions become active) or implicitly by calling functions (which require different data distributions than the calling function). To actually compile an HPF program into an efficient form, however, both the redistribution operations as well as the possible distributions for the individual blocks of code must be known. Since the HPF `REDISTRIBUTE` and `REALIGN` directives only specify redistribution information, both the intra- and interprocedural control

flow through a program must be examined to decide exactly which redistribution operations were last performed in order to determine which distributions are active at any given point in the program.

For interprocedural control flow (function calls), the distribution of the actual arguments which appear in a call to a function will not necessarily match the distribution of the dummy arguments within the function. In HPF, several different forms of distribution directives for dummy arguments are provided [2]. The distribution directive for a dummy argument may be:

prescriptive - The distribution *prescribes* the mapping of the dummy argument. Passing an actual argument with a different distribution will require redistributing the argument.

transcriptive - The distribution of the dummy argument is inherited or *transcribed* from the actual argument.

descriptive - The distribution *describes* the mapping of the dummy argument with the claim that no redistribution will take place. If an interface is present in the calling function, then this becomes a prescriptive directive.

Both prescriptive as well as descriptive arguments (with a provided interface) can have the same effect as explicit `REDISTRIBUTE` or `REALIGN` directives whereas transcriptive arguments provide a mechanism for writing functions which can accept any type of distribution. As purely descriptive directives (without an interface) do not require any additional support from an HPF compiler, they will not be discussed further in this thesis.

The basic motivation for the framework presented in this thesis is best summarized in the following excerpt from the HPF language specification [5]:

> *An overriding principle is that any mapping or remapping of arguments is not visible to the caller. This is true whether such remapping is implicit (in order to conform to prescriptive directives, which may themselves be explicit or implicit) or explicit (specified by `REALIGN` or `REDISTRIBUTE` directives). When the subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. It is not possible for a subprogram to change the mapping of any object in a manner visible to its caller, not even by means of `REALIGN` and `REDISTRIBUTE`.*

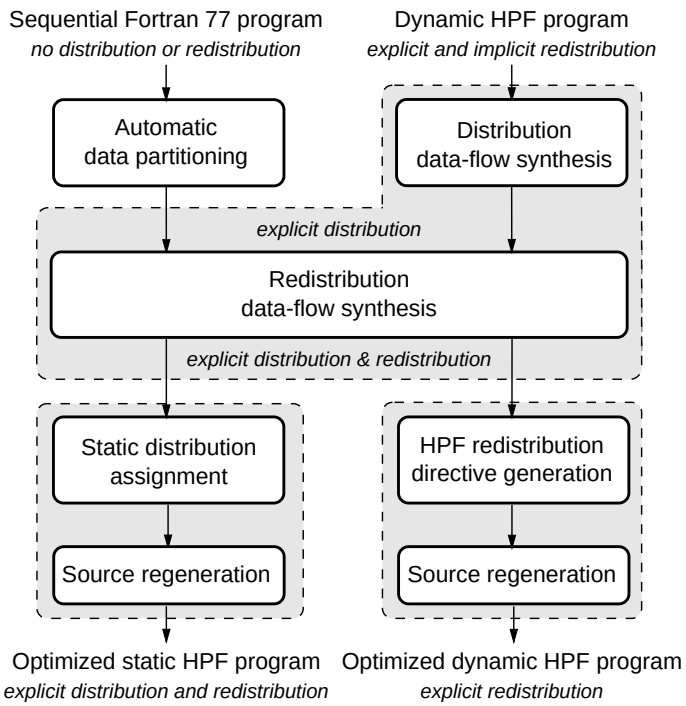


Figure 5.1: Overview of the array redistribution data-flow analysis framework (shaded areas indicate components described in this chapter)

To provide the correct semantics in the presence of implicit and explicit redistributions (or remappings) and generate efficient code, it is necessary for the compiler to perform interprocedural array data-flow analysis. The compiler should also generate efficient code avoiding the use of either a simple copy-in/copy-out strategy or a complete redistribution of all arguments upon every entry and exit of a function. As will be shown, it is possible to optimize such interprocedural redistribution simultaneously with intraprocedural redistribution operations within the framework presented in this thesis.

5.1 Data Redistribution Analysis

An overall view of the array redistribution data-flow analysis framework we have developed is shown in Figure 5.1. In addition to serving as a back end to the automatic data partitioning system [101] (described in Chapter 4), the framework is also capable of analyzing (and opti-

mizing) existing HPF programs providing a mechanism to generate fully explicit dynamic HPF programs [102].

The intermediate form of a program within the framework is one in which the distribution of every array at every point in the program as well as the redistribution required to move from one point to the next are explicitly known. The different paths through the framework involve passes which process the available distribution information in order to obtain the missing information required to move from one representation to another.

In HPF, dynamic distributions are described by specifying the transitions between different distributions (through explicit redistribution directives or implicit redistribution at function boundaries). After converting dynamic HPF programs (which are well behaved – those in which every use or definition of an array only has one reaching redistribution) to a fully static version, through a process we call static distribution assignment (SDA), which will be explained later in this thesis, both the redistribution and distribution are explicitly specified. With this framework, it is also possible to convert arbitrary HPF programs into an optimized HPF program containing only explicit redistribution directives and descriptive function arguments. The data partitioner, on the other hand, explicitly assigns different distributions to individual blocks of code serving as an automated mechanism for converting sequential Fortran programs into efficient HPF programs. In this case, the framework can be used to synthesize explicit interprocedural redistribution operations in order to preserve the meaning of what the data partitioner intended while using HPF semantics.

As will be shown in the remainder of this thesis, by analyzing the different program representations using this framework, it is possible to automatically:

(1) determine which distributions hold over specific sections of a program when given redistribution directives

(2) optimize both the inter- and intraprocedural transitions between dynamic distributions while still maintaining the original semantics of the HPF program when given distributions for each point in the program

(3) determine when the distribution pattern specified by an HPF program causes a given array to be assigned multiple distributions due to different redistribution operations on multiple paths within a function or as a result of parameter aliasing (resulting in a non-conforming HPF program)

(4) convert (well-behaved) dynamic HPF programs into equivalent static forms through a process we refer to as static distribution assignment (SDA), which can be used to extend the capabilities of existing subset HPF compilers.

The core of this analysis is built upon two separate interprocedural data-flow problems which perform:

- distribution synthesis (Section 5.2.1), and
- redistribution synthesis (Section 5.2.2).

These two data-flow problems are both based upon the problem of determining both the inter- and intraprocedural reaching distributions for a program. Before giving further details of how all of these transformations are accomplished through the use of these two data-flow problems, we will first describe the idea of reaching distributions and the basic representations we use to perform this analysis.

5.1.1 Computing reaching distributions

The problem of determining which distributions reach any given point taking into account control flow in the program is very similar to the computation of reaching definitions (a forward data-flow problem):

> *A definition d is said to reach a point p if there is a path in the control flow graph (CFG) from the point immediately following d to p, such that d is not* killed *along that path by another definition* [73].

In classic compilation theory a control flow graph consists of nodes (basic blocks) representing uninterrupted sequences of statements and edges representing the flow of control between basic blocks. For determining reaching distributions, an additional restriction must be added to

this definition. Not only should each block B be viewed as a sequence of statements with flow only entering at the beginning and leaving at the end, but the data distribution for the arrays defined or used within the block is also not allowed to change. In comparison to the original definition of a basic block, this imposes tighter restrictions on the extents of a block. Using this definition of a block in place of a basic block results in what we refer to as the distribution flow graph (DFG).

Using this view of a block in a DFG and by viewing array distributions as definitions, the same data-flow framework used for computing reaching definitions [73] can now be used to obtain the reaching distributions by defining the following sets for each block in a function:

- DIST(B) - set of distributions present when executing block B
- REDIST(B) - set of redistributions performed upon entering block B
- GEN(B) - set of distributions generated by executing block B
- KILL(B) - set of distributions killed by executing block B
- IN(B) - set of distributions that exist upon entering block B
- OUT(B) - set of distributions that exist upon leaving block B
- DEF(B), USE(B) - set of variables defined or used in block B

It is important to note that GEN and KILL are specified as the distributions generated or killed by *executing* block B as opposed to entering (redistribution at the head of the block) or exiting (redistribution at the tail of the block) in order to allow both forms of redistribution. GEN and KILL are initialized by DIST or REDIST (depending on the current applications as described in Section 5.2) and may be used to keep track of redistributions that occur on entry (e.g., HPF redistribute directives or functions with prescriptive distributions) or exit (e.g., calls to functions which internally change a distribution before returning). To perform interprocedural analysis, the function itself also has IN and OUT sets, which contain the distributions present upon entry and summarize the distributions for all possible exits.

Once the sets have been defined, the following data-flow equations are iteratively computed for each block until the solution OUT(B) converges for every block B (where PRED(B) are

the nodes which immediately precede B in the flow of the program):

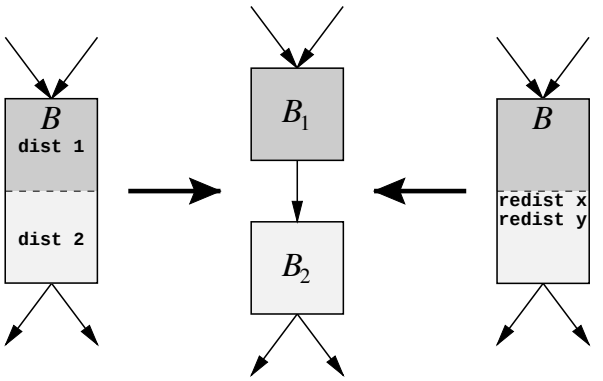
$$\mathrm{IN}(B) = \bigcup_{P \in \mathrm{PRED}(B)} \mathrm{OUT}(P) \tag{5.1}$$

$$\mathrm{OUT}(B) = \mathrm{GEN}(B) \bigcup (\mathrm{IN}(B) - \mathrm{KILL}(B)) \tag{5.2}$$

Since the confluence operator is a union, both IN and OUT never decrease in size and the algorithm will eventually halt. By processing the blocks in the flow graph in a depth-first order, the number of iterations performed will roughly correspond to the level of the deepest nested statement, which tends to be a fairly small number on real programs [73].

As can be seen from Eqs. (5.1) and (5.2), the DEF and USE sets are actually not used to compute reaching distributions, but will have other uses for optimizing redistribution which will be explained in more detail in Section 5.2.2).

5.1.2 Constructing the distribution flow graph



(a) Distribution split (b) Redistribution split

Figure 5.2: Splitting CFG nodes to obtain DFG nodes

Since the definition of the DFG is based upon the CFG, the CFG can be easily transformed into a DFG by splitting basic blocks at points at which a distribution changes as shown in Figure 5.2. This can be due to an explicit change in distribution, as specified by the automatic data partitioner, or by an actual HPF redistribution directive. If the change in distribution is due to a sequence redistribution directives, the overall effect is assigned to the block in which they

are contained; otherwise, a separate block is created whenever executable operations are interspersed between the directives.

When splitting a basic block, the existing control flow information can be updated locally to avoid computing the entire distribution flow graph from scratch. For reasons which simplify its implementation, SPLIT-NODE$(B, stmt)$ will actually create B_2 while modifying B to become B_1. The statements now included in B_2 still have pointers to the original flow graph node in which they are contained which must be updated, but nothing must be done to the statements included in B_1. This avoids handling the special case of creating a new head when splitting the first basic block in a function (since the head never changes when B is modified to become B_1).

A separate function SPLIT-NODE-AFTER is also provided to split a DFG node after a given statement. Both versions SPLIT-NODE and SPLIT-NODE-AFTER only split the node if there is actually a statement before or after (respectively) the statement of interest.

5.1.3 Representing distribution sets

To represent a distribution set in a manner that would provide efficient set and comparison operations, the bulk of the distribution information associated with a given variable is stored in its symbol table entry as a distribution table, and bit vectors are used within the sets to specify distributions which were currently active for a given variable. Since a separate symbol table entry is created for each variable within a given scope, this provides a clean interface for accepting distributions from the HPF front end [7] as any distribution present within a function due to redistribution can be added to the table corresponding to the current scope of the variable while inserting the directives into the abstract syntax tree.

As shown in Figure 5.3, the actual distribution sets are maintained as linked lists with a separate node representing each variable with a bit vector (corresponding to the entries in the distribution table for that variable) to indicate which distributions are currently active for the variable. To maintain sufficient efficiency while still retaining the simplicity of a list, the list is always maintained in sorted order by the address of the variable's entry in the symbol table in order to facilitate comparison and set operations between sets. This allows us to implement operations

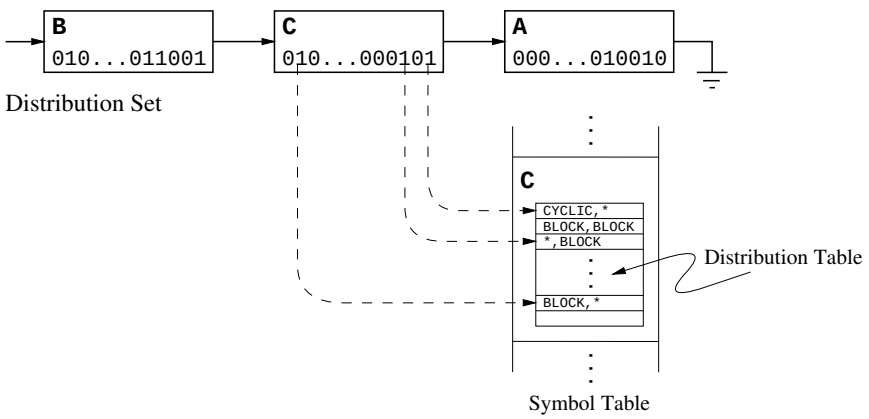


Figure 5.3: Distribution set using bit vectors

on two sets by merging them in only $\mathcal{O}(n)$ comparison or combining operations (where n is the length of the list). The operation could also involve a comparison or combination of two constant length bit vectors[1] to maintain the distribution information with $\mathcal{O}(1)$ complexity.

Since these sets are now actually sets of variables each containing a set representing their active distributions, SET_{var} will be used to specify the variables present in a given distribution set SET. For example, the notation $\overline{\text{SET}_{var}}$ can be used to indicate the inverse of the distributions *for each* variable contained within the set as opposed to an inverse over the universe of all active variables (which would be indicated as $\overline{\text{SET}}$).

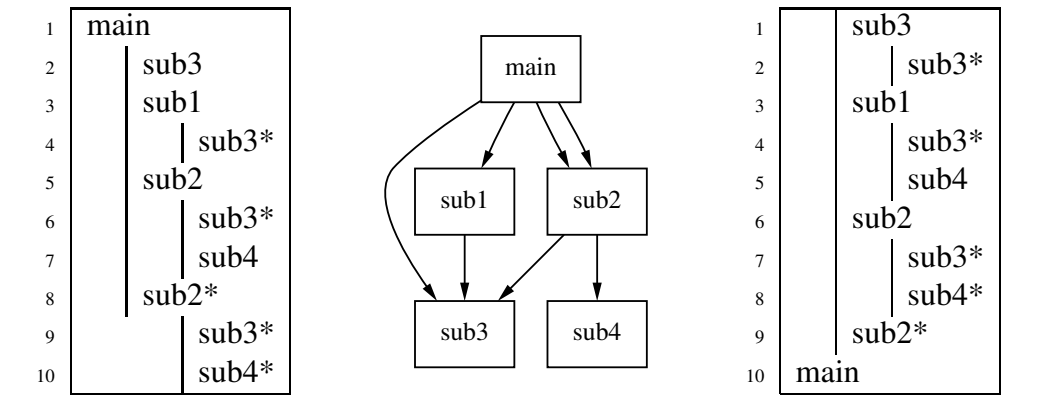
In addition to providing full union, intersection, and difference operations ($\bigcup$, $\bigcap$, $-$) which operate on both levels of the set representation (between the variable symbols in the sets as well as between the bit vectors of identical symbols) masking versions of these operations ($\uplus$, $\circledcirc$, $\div$) are also provided which operate at only the symbol level. In the case of a masking union ($a \uplus b$), a union is performed at the symbol level such that any distributions for a variable appearing in set a will be *replaced* by distributions in set b. This allows new distributions in b to be added to a set while replacing any existing distributions in a. Masking intersections ($a \circledcirc b$) and differences ($a \div b$) act somewhat differently in that the variables in set a are either selected or removed (respectively) by their appearance in set b. These two operations are useful for implementing existence operations (e.g., $a_{var}|b_{var} = a \circledcirc b$, $a_{var}|\overline{b_{var}} = a \div b$).

[1] Bit vector manipulation routines can also be written to maintain bit vectors with a variable length [103], but to simplify the initial implementation they are fixed to 32 bits (therefore allowing 32 different distributions to be present for any given symbol).

5.2 Interprocedural Data Redistribution Analysis

Since the semantics of HPF require that all objects accessible to the caller after the call are distributed exactly as they were before the call, it is possible to first completely examine the context of a call before considering any distribution side effects due to the call. It may seem strange to say that there can be side effects when we just said that the semantics of HPF preclude it. To clarify this statement, such side effects *are* allowed to exist, but only to the extent that they are not *apparent* outside of the call. As long as the view specified by the programmer is maintained, the compiler is allowed do whatever it can to optimize both the inter- and intraprocedural redistributions so long as the resulting distributions used at any given point in the program are not changed.

Referring back to Figure 5.1 the core of this analysis is built upon two separate interprocedural data-flow problems which perform distribution synthesis and redistribution synthesis. These two data-flow problems are based upon the problem of determining both the inter- and intraprocedural reaching distributions for a program. An example call graph is shown in Figure 5.4 to help illustrate the flow of these two phases of the interprocedural analysis.



(a) Top-down (pre-order) for calling context

(b) Bottom-up (post-order) for side-effects

Figure 5.4: Example call graph and depth-first traversal order

If distribution information is not present (i.e., HPF input), distribution synthesis is first performed in a top-down manner over a program's call graph to compute which distributions are

present at every point within a given function. By establishing the distributions that are present at each call site, the input distributions are obtained for each of the functions which it calls as well as causes a function to be cloned if a new set of input distributions is used. Redistribution synthesis is then applied in a bottom-up manner over the call graph to analyze where the distributions are actually used and synthesizing the redistribution required within a function.

Since this analysis is interested in the effects between an individual caller/callee pair, and not in summarizing the effects from all callers before examining a callee, it is not necessary to perform a topological traversal for the top-down and bottom-up passes over the call graph. In this case, it is actually more intuitive to perform a depth-first pre-order traversal of the call graph (shown in Figure 5.4(a)) to fully analyze a given function before proceeding to analyze any of the functions it calls and to perform a dept-first post-order traversal (shown in Figure 5.4(b)) to fully analyze all called functions before analyzing the caller.

One other point to emphasize is that these interprocedural techniques can be much more efficient than analyzing a fully inlined version of the same program since it is possible to prune the traversal at the point a previous solution is found for a function in the same calling context. In Figure 5.4, asterisks indicate points at which a function is being examined after having already been examined previously. If the calling context is the same as the one used previously, the traversal can be pruned at this point reusing information recorded from the previous context. Depending on how much reuse occurs, this factor can greatly reduce the amount of time the compiler spends analyzing a program in comparison to a fully inlined approach.

To summarize the process shown in Figure 5.1, the distribution synthesis pass

(1) determines the reaching distributions for every point in the function according to the intraprocedural control flow, and

(2) determines which input distributions will be present at each call site (calling context) cloning functions when necessary

while the redistribution synthesis pass

(1) restricts the distributions to those used at each point in the program, and

(2) optimizes redistributions by examining both the intraprocedural flow of distributions and interprocedural side effects of each function call in the caller's context.

After both determining the reaching distributions and optimizing the required redistribution, the internal representation can then be converted into either a static or dynamic HPF program with fully explicit redistribution.

The algorithms for performing distribution synthesis will be presented in Section 5.2.1 while redistribution synthesis will be presented in Section 5.2.2. The static HPF conversion, known as static distribution assignment (SDA), is covered in detail in Section 5.2.3. Since the dynamic HPF conversion only entails generating redistribution directives based on the contents of the REDIST sets, it will not be discussed further in this section.

5.2.1 Distribution synthesis

When analyzing HPF programs, it is necessary to first perform distribution synthesis in order to determine which distributions are present at every point in a program. Since HPF semantics specify that any redistribution (implicit or explicit) due to a function call is not visible to the caller, each function can be examined independently of the functions it calls. Only the input distributions for a given function and the explicit redistribution it performs have to be considered to obtain the reaching distributions for a function.

Given an HPF program, nodes (or blocks) in its DFG are delimited by the redistribution operations which appear in the form of HPF `REDISTRIBUTE` or `REALIGN` directives. As shown in Figure 5.5, the redistribution operations assigned to a block B represent the redistribution that will be performed when entering the block on any input path (indicated by the set REDIST(B)) as opposed to specifying the redistribution performed for each incoming path (REDIST(B, B_1) or REDIST(B, B_2) in the figure).

If the set GEN(B) is viewed as the distributions which are generated and KILL(B) as the distributions which are killed upon entering the block, this problem can now be cast directly into the reaching distribution data-flow framework by making the following assignments:

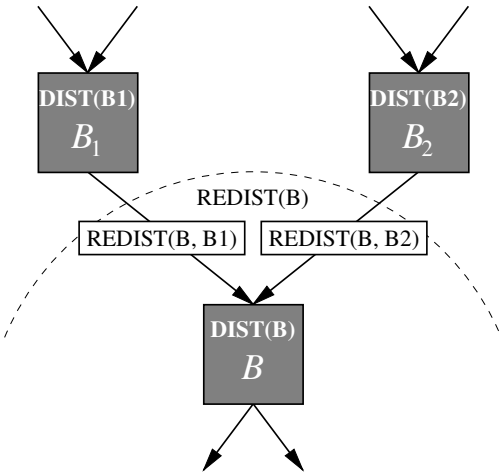


Figure 5.5: Distribution synthesis
(converting redistributions to distributions)

Data-flow initialization:

$$\begin{array}{rclrcl} \text{REDIST}(B) & = & \text{from directives} & \text{DIST}(B) & = & \emptyset \\ \text{GEN}(B) & = & \text{REDIST}(B) & \text{KILL}(B) & = & \overline{\text{REDIST}_{var}(B)} \\ \text{OUT}(B) & = & \text{REDIST}(B) & \text{IN}(B) & = & \emptyset \end{array}$$

Data-flow solution:

$$\text{DIST(B)} = \text{OUT}(B)$$

The full interprocedural algorithm for converting redistribution into distribution information is shown in Figure 5.6. This algorithm recurses at line 20 in R2D-FUNC to perform the top-down pruning traversal previously shown in Figure 5.4(a) computing the data-flow solution for each function at each step in the traversal. Before examining calls made within a function, IN($function$) is computed at line 19 in R2D-FUNC for both the parameters as well as all of the globals used within the function call.[2] The distributions of all of the actual parameters are initially assumed to be inherited and are masked with any prescriptive distributions that may exist for the function. One final point to make is that even though functions are cloned during the top-down traversal, they are actually recorded at line 22 as each call site is examined after the traversal of the given function is complete.

[2]The `sumfcn` pass in Parafrase-2 [6] is first run to summarize all global references made within every function.

```
REDIST2DIST(program)
 1  function ← MAIN_FUNCTION(program)
 2  R2D-FUNC(function, STATIC_DIST(function))

R2D-FUNC(function, indist)
 1  if a solution already exists for indist
 2    then return function
 3    else function ← CLONE(function)
 4  ▷ Obtain dataflow solution for function
 5  R2D-INIT(function, indist)
 6  repeat
 7        for each node in FLOW_GRAPH(function)
 8            do update IN/OUT data-flow equations for node
 9    until convergence
10  ▷ Record DIST solution
11  for each node in FLOW_GRAPH(function)
12      do DIST(node) ← OUT(node)
13  OUT(function) ← DIST(STOP_NODE(function))
14
15  ▷ Perform top-down traversal over call graph
16  for each call in CALL_LIST(function)
17      do indist ← (actual parameters mapped to dummy parameters
18                              + required globals) from OUT(NODE(call))
19         indist ← indist ⋓ PRESCRIBED_DIST(FUNCTION(call))
20         clone ← R2D-FUNC(call, indist)
21         if FUNCTION(call) ≠ clone
22           then FUNCTION(call) ← clone     ▷ Record call to clone
23  return function

R2D-INIT(function, indist)
 1  reset IN/OUT(function)
 2  IN(function) ← indist
 3  for each node in FLOW_GRAPH(function)
 4      do reset REDIST/DIST(node)
 5  R2D-HPF-REDIST-INIT(function)      ▷ Add HPF redistribution
 6  for each node in FLOW_GRAPH(function)
 7      do GEN(node) ← REDIST(node)
 8         KILL(node) ← INVERTVAR(GEN(node))
 9         IN(node) ← ∅
10         OUT(node) ← GEN(node)

R2D-HPF-REDIST-INIT(function)
 1  for each HPF_DIR(stmt) in function
 2      do node ← FLOW_GRAPH(stmt)
 3         if REDIST occurs after a normal statement in block
 4           then node ← SPLIT-NODE(node, stmt)
 5         REDIST(node) ← REDIST(node) ⋓ (redist specified by HPF directive)
```

Figure 5.6: Interprocedural analysis for distribution synthesis

It is important to note that R2D-HPF-INIT processes executable HPF directives in the order in which they appear in the program and adds them to any existing REDIST sets using a masking union operation. This captures the sequential ordering of the HPF directives, which is important if the programmer should (accidentally) redistribute the same array multiple times in a sequence of redistribution directives. Only the last distribution for a given sequence should be recorded in the REDIST set for the associated block. The overall effect of a sequence of consecutive redistribution directives is therefore assigned to the entry of the block. If the redistribution operation occurs after a normal statement in the block, however, a new DFG node is created.

According to the the HPF standard, a `REALIGN` operation only affects the array being realigned while a `REDISTRIBUTE` operation should redistribute all arrays currently aligned to the given array being redistributed (in order to preserve any previous specified alignments). In the current implementation, R2D-HPF-INIT only records redistribution information for the array immediately involved in a `REDISTRIBUTE` operation. This results in only redistributing the array involved in the directive and not all of the alignees of the target to which it is aligned. In the future, the implementation could be easily extended to support the full HPF interpretation of `REDISTRIBUTE` by simply recording the same redistribution information for all alignees for the target of the array involved in the operation.

5.2.2 Redistribution synthesis

After the distributions have been determined for each point in the program, the redistribution can be optimized. Instead of using either a simple copy-in/copy-out strategy or a complete redistribution of all arguments upon every entry and exit of a function, any implicit redistribution around function calls can be reduced to only that which is actually required to preserve the HPF semantics. Any unnecessary redistribution operations (implicitly specified by HPF semantics or explicitly specified by a programmer) that result in distributions which would not otherwise be used before another redistribution operation occurs are completely removed in this pass.

Blocks are now delimited by changes in the distribution set. As shown in Figure 5.7, the set of reaching distributions previously computed for a block B represent the distributions which

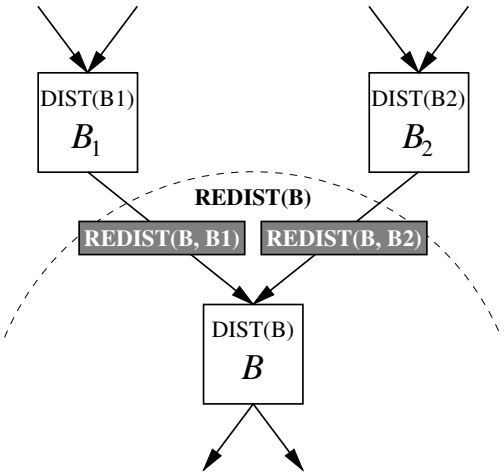


Figure 5.7: Redistribution synthesis
(converting distribution to redistribution)

are in effect when executing that block (indicated by the set DIST$(()B)$). By first restricting the DIST(B) sets to the variables defined or used within block B, a redistribution operation will only be performed between two blocks if there is an intervening definition or use of that variable before the next change in distribution.

This also has the effect of generating redistribution when it is needed (demand-driven, or lazy, redistribution), in order to ensure that the distribution is actually used in a different distribution before performing a redistribution. Although it will not be examined here it would also be possible to take a lazy redistribution solution and determine the earliest possible time that the redistribution could be performed (eager redistribution) in order to redistribute an array when a distribution is no longer in use. The area between the eager and lazy redistribution points forms a window over which the operation can be performed to obtain the same effect. As will be shown later, it would be advantageous to position multiple redistribution operations in overlapping windows to the same point in the program in order to aggregate the communication thereby reducing the amount of communication overhead (as previously described in Section 3.3). As the lazy redistribution point is found using a forward data-flow (reaching distributions) problem, it would be possible to find the eager redistribution point by performing some additional bookkeeping to record the last use of a variable as the reaching distributions are propagated along the flow graph; however, such a technique is not currently implemented

in PARADIGM. In comparison to other approaches, interval analysis has also been used to determine eager/lazy points for code placement, but at the expense of a somewhat more complex formulation [104].

If the set GEN(B) is viewed as the distributions which are generated and KILL(B) as the distributions which are killed upon leaving block B, this problem can now be cast directly into the reaching distribution data-flow framework by making the following initializations:

data-flow initialization:

$$\begin{array}{rclcrcl} \text{REDIST}(B) & = & \emptyset & & \text{DIST}(B) & = & \text{DIST}(B) \cap (\text{DEF}(B) \cup \text{USE}(B)) \\ \text{GEN}(B) & = & \text{DIST}(B) & & \text{KILL}(B) & = & \overline{\text{DIST}_{var}(B)} \\ \text{IN}(B) & = & \emptyset & & \text{OUT}(B) & = & \text{DIST}(B) \end{array}$$

data-flow solution:

$$\begin{aligned} \text{REDIST}(B,P) &= \text{DIST}_{var}(B) \mid (\text{OUT}_{var}(P) - \text{DIST}_{var}(B)) \neq \emptyset \ \ (\forall P \in \text{PRED}(B)) \\ \text{REDIST}(B) &= \bigcup_{P \in \text{PRED}(B)} \text{REDIST}(B,P) \\ &= \text{DIST}_{var}(B) \mid (\text{IN}_{var}(B) - \text{DIST}_{var}(B)) \neq \emptyset \end{aligned}$$

We define GEN(B) and KILL(B) as the distributions which are generated or killed upon leaving block B due to the fact that we have chosen to use a caller redistributes model. As will be seen later, this exposes many interprocedural optimization opportunities and is also necessary to support function calls which may require redistribution on both their entry and exit. Since REDIST(B) is determined from both the DIST and IN sets, DIST(B) represents the distributions needed for executing block B, while the GEN, KILL sets will be used to represent the exit distribution (which may or may not match DIST).

The full interprocedural algorithm for converting distribution to redistribution information is shown in Figure 5.8. It should be noted that this algorithm traverses the expanded call graph that was formed by any cloning performed during distribution synthesis. This algorithm recurses at line 6 in D2R-FUNC to perform the bottom-up pruning traversal previously shown in Figure 5.4(b) computing the data-flow solution for each function at each step in the traversal. After all call sites have been examined for a function, both OUT($call$) and IN($call$) are examined in D2R-CALL-INIT (shown in Figure 5.9) to determine if there was an implicit change in

```
DIST2REDIST(program)
 1  for each function in program     ▷ Initialize all functions as not yet visited
 2      do VISITED(function) ← FALSE
 3  function ← MAIN_FUNCTION(program)
 4  D2R-FUNC(function)

D2R-FUNC(function)
 1  if VISITED(function)     ▷ Check for previous DIST2REDIST solution
 2    then return
 3    else  VISITED(function) ← TRUE
 4  ▷ Perform bottom-up traversal over call graph
 5  for each call in CALL_LIST(function)
 6      do D2R-FUNC(call)
 7
 8  ▷ Obtain dataflow solution for function
 9  D2R-INIT(function)
10  repeat
11          for each node in FLOW_GRAPH(function)
12              do update IN/OUT data-flow equations for node
13    until convergence
14  ▷ Record REDIST(node)
15  for each node in FLOW_GRAPH(function)
16      do REDIST(node) ← DIST(node)|(IN(node) − DIST(node)) ≠ ∅
17  outdist ← parameters and globals from STOP_NODE(function)
18  OUT(function) ← outdist

D2R-INIT(function)
 1  for each node in FLOW_GRAPH(function)
 2      do reset REDIST,GEN/KILL/IN/OUT(node)
 3  for each node in FLOW_GRAPH(function)
 4      do ▷ Restrict distributions to DEF/USE for node
 5          DIST(node) ← DIST(node) ⋂ DEFUSE(node)
 6          GEN(node) ← DIST(node)      ▷ Initialize GEN set
 7  DIST-GROW-INVARIANT(function)     ▷ Record invariants
 8  D2R-INIT-CALL(function)           ▷ Record IN/OUT sets for each call
 9  for each node in FLOW_GRAPH(function)
10      do KILL(node) ← INVERTVAR(GEN(node))
11          IN(node) ← ∅
12          OUT(node) ← GEN(node)
```

Figure 5.8: Interprocedural analysis for redistribution synthesis

```
D2R-CALL-INIT(function)
 1  for each call in CALL_LIST(function)
 2      do indist ← (dummy parameters mapped to actuals
 3                   + globals) from IN(FUNCTION(call))
 4         outdist ← (dummy parameters mapped to actuals
 5                   + globals) from OUT(FUNCTION(call))
 6         node ← NODE(call)
 7         stmt ← STMT(call)
 8         if outdist ∉ DIST(node)
 9           then newnode ← SPLIT-NODE-AFTER-STMT(node, stmt)
10                GEN(node) ← GEN(node) ⊎ outdist
11                GEN(newnode) ← DIST(newnode)
12         if indist ∉ DIST(node)
13           then if call occurs after a normal statement in block
14                  then node ← SPLIT-NODE(node, stmt)
15                DIST(node) ← DIST(node) ⊎ indist

DIST-GROW-INVARIANT(function)
 1  for each node in function
 2      do TMPDIST(node) ← DIST(node)
 3  for each stmt in function
 4      do DIST-GROW-STMT(stmt)
 5  for each node in function
 6      do invdist = TMPDIST(node) | only one distribution per variable
 7         if invdist ≠ ∅
 8           then ▷ Add to existing DIST and GEN sets for node
 9                DIST(node) ← DIST(node) ⊎ invdist
10                GEN(node) ← GEN(node) ⊎ DIST(node)

DIST-GROW-STMT(stmt)
1  node ← FLOW_GRAPH(stmt)
2  dist ← DIST(node)        ▷ distributions present at node
3  gendist ← GEN(node)      ▷ distributions generated at node
4  if dist ≠ ∅    ▷ Add distributions to parent statments
5    then while stmt ← PARENT(stmt)
6               do node ← FLOW_GRAPH(stmt)
7                  TMPDIST(node) ← TMPDIST(node) ⋃ dist
8                  TMPDIST(node) ← TMPDIST(node) ⋃ gen
```

Figure 5.9: Initialization for redistribution synthesis

distribution at the call site. If so, the node containing the call site is appropriately split[3] and the GEN and DIST sets for the node are masked by the OUT and IN distributions sets, respectively, as required for the call.

5.2.2.1 Optimizing invariant distributions

Besides performing redistribution only when necessary, it is also desirable to only perform necessary redistribution as infrequently as possible. An algorithm for growing semantically invariant distribution regions[4] before synthesizing the redistribution operations is shown in Figure 5.9. This algorithm records all distributions that do not change within a nested statement (loop or if structures) on the parent statement (or header) for that structure. This has the effect of moving redistribution operations which result in the invariant distribution out of nested structures as far as possible. An example of this optimization will be illustrated later in Section 5.3.4.

As a side effect, loops which are considered to contain an invariant distribution no longer propagate previous distributions for the invariant arrays. Since redistribution is moved out of the loop, this means that for the (extremely rare) special case of a loop invariant distribution (which was not originally present outside of the loop) contained within a undetectable zero trip loop, only the invariant distribution from within the loop body is propagated even though the loop nest was never executed. As this is only due to the way invariant distributions are handled, the dataflow handles non-invariant distributions as expected for zero trip loops (an extra redistribution check may be generated after the loop execution).

One last point to note is that the algorithm presented in Figure 5.9 is conservative since it only pulls out distributions that are completely invariant, not distributions that are invariant with respect to the start and end of an iteration but change distributions intermediately. Although, not

[3] If a node is split at a call that is contained within a larger expression and which refers to any array that changed distribution during the call, the compiler must split the expression into multiple statements at the call site using temporary variables to obtain the partial results and preserve the order of execution, or perform redistribution at the end of function before returning which may require more cloning based on the required return distribution.

[4] Invariant redistribution with respect to the semantics of HPF can technically become non-invariant when return distributions from function calls within a loop nest are allow to temporarily exist in the caller's scope (and therefore are recorded before considering any interprocedural side effects due to function calls with different return distributions). Such regions can be still treated as invariant, with the proper support as will be addressed in Section 5.3.1, since this is the view HPF provides to the programmer.

shown here, this would require computing a separate "nested" DEF/USE set over the loop body for each loop header, performing a union of the (unrestricted) reaching distributions for only the entry and exit nodes of the loop, and then restricting this resulting set by the nested DEF/USE set.

5.2.2.2 Multiple active distributions

Even though it is not specifically stated as such in the HPF standard, we will consider an HPF program in which every use or definition of an array has only one active distribution to be well-behaved. Since PARADIGM cannot currently compile programs which contain references with multiple active distributions, this property is currently detected by examining the reaching distribution sets for every node (limited by DEF/USE) within a function. A warning is issued if any set contains multiple distributions for a given variable stating that the program is not well-behaved. An example of this situation will be illustrated later in Section 5.3.4.

In the presence of function calls, it is also possible to access an array through two or more more paths when parameter aliasing is present. If there is an attempt to redistribute one of the aliased symbols, the different aliases now have different distributions even though they actually refer to the same array. This form of multiple active distributions is actually considered to be non-conforming in HPF [2] as it can result in consistency problems if the same array were allowed to occupy two different distributions. As it may be difficult for the programmers to make this determination, this can be automatically detected by determining if the reaching distribution set contains different distributions for any aliased arrays.[5] An algorithm is presented in Figure 5.10 which examines the reaching distribution sets for every node within a function for this condition.

5.2.3 Static Distribution Assignment (SDA)

To utilize the available memory on a given parallel machine as efficiently as possible, only the distributions that are active at any given point in the program should actually be allocated

[5]The `param_alias` pass in Parafrase-2 [6] is first run to compute the alias sets for every function call.

```
ILLEGAL-ALIASING-DIST(distset)
 1  illegal ← FALSE
 2  srcdist ← distset
 3  srcsym ← DYN_VAR(srcdist)
 4  while srcdist
 5        do tgtdist ← DYN_NEXT(srcdist)
 6           tgtsym ← DYN_VAR(tgtdist)
 7           while tgtdist
 8                 do ▷ Check for aliasing present between srcsym and tgtsym
 9                    if (srcsym and tgtsym in same alias set) and
10                          not EQUIVALENT-DISTS(srcdist, tgtdist)
11                       then illegal ← TRUE
12                    tgtdist ← DYN_NEXT(tgtdist)
13           srcdist ← DYN_NEXT(srcdist)
14  return illegal

EQUIVALENT-DISTS(srcdist, tgtdist)
 1  equivalent ← TRUE
 2  srcsym ← DYN_VAR(srcdist)
 3  tgtsym ← DYN_VAR(tgtdist)
 4  ▷ Both should have same number of unique distributions
 5  if DYN_NUMDIST(srcdist) ≠ DYN_NUMDIST(tgtdist)
 6     then equivalent ← FALSE
 7  ▷ All distributions should be present for both symbols
 8  for each dist in srcdist
 9      do if S_DYN_DIST(srcsym,dist) ∉ S_DYN_DIST(tgtsym, *)
10            then equivalent ← FALSE
11  return equivalent
```

Figure 5.10: Algorithms for detecting non-conforming HPF programs

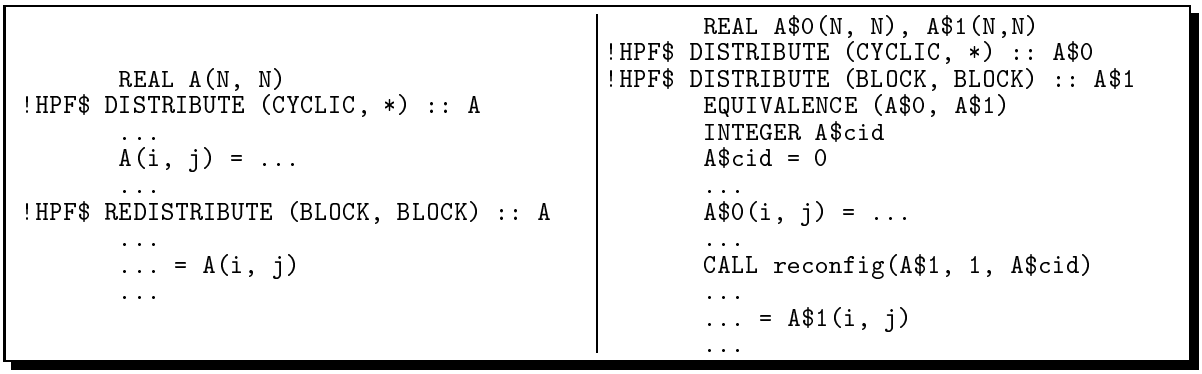
space. It is interesting to note that as long as a given array is distributed among the same total number of processors, the actual space required to store one section of the partitioned array is the same no matter how many array dimensions are distributed.[6] By using this observation, it is possible to statically allocate the minimum amount of memory by associating all possible distributions of a given array to the same area of memory.

Static Distribution Assignment (SDA) (inspired indirectly by the Static Single Assignment (SSA) form [105]) is a process we have developed in which the names of array variables are duplicated and renamed statically based on the active distributions represented in the corresponding DIST sets. As names are generated, they are assigned a static distribution (by which we mean this new name will not change distribution during the course of the program) corresponding to the currently active dynamic distribution for the original array. Redistribution now takes the form of moving data from a statically distributed source array to another statically distributed destination array (as opposed to rearranging the data within a single array).

To statically achieve the minimum amount of memory allocation required, all of the renamed duplicates of a given array are declared to be "equivalent." The `EQUIVALENCE` statement in Fortran 77 allows this to be performed at the source level in a somewhat similar manner as assigning two array pointers to the same allocated memory as is possible in C or Fortran 90, Redistribution directives are also now replaced with actual calls to a redistribution library.

Because the different static names for an array share the same memory, this implies that the communication operations used to implement the redistribution should read all of the source data before writing to the target. In the worst case, an entire copy of a partitioned array can be buffered at the destination processor before it is actually received and moved into the destination array. However, as soon as more than two different distributions are present for a given array, the `EQUIVALENCE` begins to pay off, even in the worst case, in comparison to separately allocating each different distribution. If the performance of buffered communication is insufficient for a given machine (due to the extra buffer copy), non-buffered communication could be used

[6] Taking into account distributions in which the number of processors allocated to a given array dimension does not evenly divide the size of the dimension, it can be equivalently said that there is a given amount of memory which can store all possible distributions with very little excess.



```
      REAL A(N, N)
!HPF$ DISTRIBUTE (CYCLIC, *) :: A
      ...
      A(i, j) = ...
      ...
!HPF$ REDISTRIBUTE (BLOCK, BLOCK) :: A
      ...
      ... = A(i, j)
      ...
```

(a) Before SDA

```
      REAL A$0(N, N), A$1(N,N)
!HPF$ DISTRIBUTE (CYCLIC, *) :: A$0
!HPF$ DISTRIBUTE (BLOCK, BLOCK) :: A$1
      EQUIVALENCE (A$0, A$1)
      INTEGER A$cid
      A$cid = 0
      ...
      A$0(i, j) = ...
      ...
      CALL reconfig(A$1, 1, A$cid)
      ...
      ... = A$1(i, j)
      ...
```

(b) After SDA

Figure 5.11: Example of static distribution assignment

instead thereby precluding the use of `EQUIVALENCE` (unless some form of explicit buffering is performed by the redistribution library itself).

In Figure 5.11, a small example is shown to illustrate this technique. In this example, a redistribution operation on `A` causes it to be referenced using two different distributions. A separate name is statically generated for each distribution of `A`, and the redistribution directive is replaced with a call to a run-time redistribution library [84]. The array accesses in the program can now be compiled by PARADIGM using techniques developed for programs which only contain static distributions [12, 89] by simply ignoring the communication side effects of the redistribution call. Although it is not needed for this example, a separate ID variable, `A$cid`, is also maintained to indicate the current configuration of the corresponding array. When different distributions reach a redistribution point through different control-flow paths, this ID variable is necessary to determine the source distribution of an array. In this example, the ID variable is also assigned statically generated constants which correspond to the different distributions. It is also possible to use instead symbolic constants that index into a version of the compiler's symbol table, containing the actual distribution for each configuration, made available at run time. Such a run-time symbol table, allows the compiler to generate interprocedural SDA code

by adding ID variables into a function's parameter list for each distributed array to properly handle parameter mapping across function boundaries.[7]

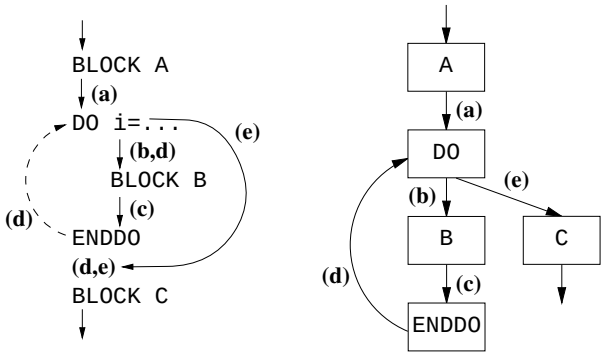
If more than one distribution is active at any given point in the program, the program is considered to be *not well-behaved*, and the array variable can not be directly assigned a static distribution. Furthermore, in order to even generate code for multiple active distributions, it must be able to handle all combinations of active distributions for the referenced variables. If code specialization is performed (which is currently not performed by PARADIGM), it is possible to generate a large number (the product of the number of distributions for each array in the code block) of versions for a given block of code. It should be noted that this situation can only occur when accepting HPF programs that were written by hand using the `REDISTRIBUTE` or `REALIGN` directives. This problem will not arise when dealing with HPF programs that were automatically generated by the automatic data partitioning techniques previously described. In this case, the data partitioner assigns full distribution descriptions to individual blocks of code as opposed to indirectly specifying them through redistribution operations.

In certain circumstances, however, it may be possible to perform code transformations to make an HPF program well-behaved. For instance, a loop that contained multiple active distributions on the entry to its body due only to a distribution from the loop back edge (caused by redistribution within the loop) that wasn't present on the loop entry would not be well-behaved. If the first iteration of that loop were peeled off, the entire loop body would now have a single active distribution for each variable and the initial redistribution into this state would be performed outside of the loop. This and other code transformations which help reduce the number of distributions reaching any given node will be the focus of further work in this area.

5.3 Implementation

In this section we will discuss some of the implementation details related to the data-flow framework and present results of applying these techniques on several examples.

[7] We are currently in the process of developing the run-time system to fully support the interprocedural SDA code which the compiler already generates.



(a) Source code (b) Flow graph

Figure 5.12: Example loop shown with flow transitions

5.3.1 Source level data flow

The data-flow equations (Eqs. (5.1) and (5.2)) are computed for all blocks in the same manner except for one special consideration that must be made for `DO` loop headers. Since the resulting information will be used in a source-to-source platform, care must be taken to ensure that all flow paths are realizable at the source level. In Figure 5.12, the source code for a loop and its corresponding flow graph are shown illustrating where the transitions **(a)–(e)** between the basic blocks[8] actually appear in the source code. As the transition along the loop backflow edge **(d)** does not have an explicit location in the source code, this edge requires special treatment. By directing the flow from edge **(d)** into the OUT set of the loop header, instead of its IN set, this backedge flow is not observed as an input to the `DO` loop header, but correctly appears at the entry of both the loop body, as well as at the entry of the code following the loop (e.g., block B). The points indicated as **(b,d)** and **(d,e)** in Figure 5.12(a) therefore correspond to the locations at which the effects of the backflow edge will appear combined with the effects of edges **(b)** and **(e)** in the regenerated source code. Even though this example illustrates the flow for a `DO` loop, this approach extends to applying data-flow solutions at the source level for all types of iterative loops.[9]

[8] The `ENDDO` statement is normally included as part of the loop body, but it is shown separately here to emphasize flow between the loop body and the end of the loop.

[9] In contrast, loops formed using unstructured constructs (i.e., `GOTO` operations that branch back to a label) do not require any special treatment since the target label and the associated statement can be separated forming a region in which the back transition **(d)** can be placed.

Invariant distributions recorded in the GEN on loop headers, which are no longer truly invariant due to optimizations removing intermediate redistributions around function calls within a loop body, raise a subtle point with respect to this treatment. As the distributions along the loop backflow must take precedence over loop invariants when iterating upon the loop body, this ordering is maintained by masking the GEN set by the loop backflow before adding it to the OUT set for the flow edge **(b)** from the header into the loop body. However, as loop invariant distributions take precedence over loop backflow upon exiting the loop body, the reverse is done, masking the OUT set by the GEN set flow for the loop exit edge **(e)**.

5.3.2 Virtual clones

Because there is considerable state associated with the body of a function, it is much more efficient to record the cloned data as a separate context for the function and delay the actual cloning until later, a process which we refer to as *virtual cloning*.

For this work, this requires maintaining cloned copies of only the REDIST, DIST, IN, OUT sets on each DFG node within a function as well as the IN, and OUT sets for the function itself. To support virtual clones in general, the specification of a function now becomes a pair (function symbol table entry, clone number). References to functions (i.e., call sites) also record which clone is to be used for the call in the current virtual clone context.

By creating a "virtual" clone, all of the other state associated with the function (e.g., control-flow and dependence graphs) can be reused during interprocedural analysis within each virtual clone's context. When the actual source is to be regenerated, only the abstract syntax tree (AST) is actually cloned making specific modifications based on the information recorded as virtual clones.

5.3.3 Array reshaping

One area that is not fully addressed by the current implementation is array reshaping, which can occur through function parameters at call boundaries by linearizing a multi-dimensional array or passing a slice of an array (with or without an offset). As array reshaping does not always

result in a valid HPF distribution in general, it might be necessary to label invalid distributions as irregular, and only map distributions for the cases in which reshaping results in a valid HPF distribution. In either case, array reshaping can be easily handled by the presented framework, but further work is still required to extend the current implementation to support distribution mapping when mapping the actual parameters to the dummy parameters while taking into account the possibility of array reshaping.

5.3.4 Examples

In Figure 5.13(a), a synthetic HPF program is presented which performs a number of different tests (described as comments in the input code) of the optimizations performed by the framework. In this program, one array, `x`, is redistributed both explicitly using HPF directives and implicitly through function calls using several different interfaces. Two of the functions, `func1` and `func2`, have prescriptive interfaces which may or may not require redistribution (depending on the current configuration of the input array). The first function differs from the second in that it also redistributes the array such that it returns with a different distribution than which it was called. The last function, `func3`, differs from the first two in that it has an (implicit) transcriptive interface. Calls to this function will cause it to inherit the current distribution of the actual parameters.

Several things can be noted when examining the optimized HPF shown[10] in Figure 5.13(b). First of all, the necessary redistribution operations required to perform the implicit redistribution at the function call boundaries have been made explicit in the program. Here, the interprocedural analysis has completely removed any redundant redistribution by relaxing the HPF semantics allowing distributions caused by function side effects to exist so long as they do not affect the original meaning of the program. For the transcriptive function, `func3`, the framework has generated two separate clones, `func3$0` and `func3$1`, corresponding to two different active distributions at a total of three different calling contexts.

[10] The HPF output, generated by PARADIGM, has been slightly simplified in the following figures to improve their clarity. Unnecessary alignment directives have been removed and consecutive redistribution operations have been collapsed into one.

```
      PROGRAM test
      INTEGER x(10,10)
c     *** For tests involving statement padding
      INTEGER a
!HPF$ PROCESSORS :: square(2,2)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, BLOCK) :: x
c     *** Use of initial distribution
      x(1,1) = 1
c     *** Testing loop invariant redistribution
      DO i = 1,10
         DO j = 1,10
            a = 0
!HPF$        REDISTRIBUTE (BLOCK, CYCLIC) :: x
            x(i,j) = 1
            a = 0
         ENDDO
      ENDDO
      a = 0
c     *** Testing unnecessary redistribution
!HPF$  REDISTRIBUTE (BLOCK, CYCLIC) :: x
      if (x(i,j) .gt. 1) then
c        *** Testing redistribution in a conditional
!HPF$    REDISTRIBUTE (BLOCK, BLOCK) :: x
         x(i,j) = 2
         call func3(x,n)
      else
         x(i,j) = 3
      endif
c     *** Uses with multiple reaching distributions
      x(1,1) = 2
      call func1(x,n)
      DO i = 1,10
         DO j = 1,10
            x(j,i) = 2
         ENDDO
      ENDDO
!HPF$  REDISTRIBUTE (CYCLIC(3), CYCLIC) :: x
c     *** Testing chaining of function arguments
      call func1(x,n)
      call func2(x,n)
      call func1(x,n)
c     *** Testing loop invariant due to return
      DO i = 1,10
         DO j = 1,10
c           *** Test cloning for transcriptive functions
            call func3(x,n)
         ENDDO
      ENDDO
      a = 1
c     *** Testing unused distribution
!HPF$  REDISTRIBUTE (BLOCK, CYCLIC) :: x
      a = 0
c     *** Testing "semantically killed" distribution
!HPF$  REDISTRIBUTE (CYCLIC(3), CYCLIC) :: x
      call func3(x,n)
      END

      integer function func1(a,n)
c     *** Prescriptive function with different return
      integer n, a(n, n)
!HPF$  DYNAMIC, DISTRIBUTE (BLOCK, CYCLIC) :: a
      a(1,1) = 1
!HPF$  REDISTRIBUTE (BLOCK, CYCLIC) :: a
      a(1,2) = 1
!HPF$  REDISTRIBUTE (CYCLIC, CYCLIC) :: a
      a(1,3) = 1
      end

      integer function func2(y,n)
c     *** Prescriptive function with identical return
      integer n, y(n,n)
!HPF$  DYNAMIC, DISTRIBUTE (CYCLIC, CYCLIC) :: y
      y(1,1) = 2
      end

      integer function func3(z,n)
c     *** (implicitly) Transcriptive function
      integer n, z(n,n)
      z(1,1) = 3
      end
```

(a) Before optimization

```
      PROGRAM test
      INTEGER x(10,10)
      INTEGER a, n
!HPF$ PROCESSORS :: square(2,2)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, BLOCK) :: x
      x(1,1) = 1
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      DO i = 1,10
        DO j = 1,10
          a = 0
          x(i,j) = 1
          a = 0
        END DO
      END DO
      a = 0
      IF (x(i,j) .GT. 1) THEN
!HPF$ REDISTRIBUTE (BLOCK, BLOCK) ONTO square :: x
        x(i,j) = 2
        CALL func3$0(x,n)
      ELSE
        x(i,j) = 3
      END IF
c     *** WARNING: too many dists (2) for x
      x(1,1) = 2
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      CALL func1(x,n)
      DO i = 1,10
        DO j = 1,10
c         *** WARNING: too many dists (2) for x
          x(j,i) = 2
        END DO
      END DO
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      CALL func1(x,n)
      CALL func2(x,n)
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      CALL func1(x,n)
!HPF$ REDISTRIBUTE (CYCLIC(3), CYCLIC) ONTO square :: x
      DO i = 1,10
        DO j = 1,10
          CALL func3$1(x,n)
        END DO
      END DO
      a = 1
      a = 0
      CALL func3$1(x,n)
      END

      INTEGER FUNCTION func1(a,n)
      INTEGER n, a(n,n)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, CYCLIC) :: a
      a(1,1) = 1
      a(1,2) = 1
!HPF$ REDISTRIBUTE (CYCLIC, CYCLIC) ONTO square :: a
      a(1,3) = 1
      END

      INTEGER FUNCTION func2(y,n)
      INTEGER n, y(n,n)
!HPF$ DYNAMIC, DISTRIBUTE(CYCLIC,CYCLIC) ONTO square :: y
      y(1,1) = 2
      END

      INTEGER FUNCTION func3$1(n,z)
      INTEGER n, z(n,n)
!HPF$ DISTRIBUTE(CYCLIC(3),CYCLIC) ONTO square :: z
      z(1,1) = 3
      END

      INTEGER FUNCTION func3$0(n,z)
      INTEGER n, z(n,n)
!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO square :: z
      z(1,1) = 3
      END
```

(b) After optimization

Figure 5.13: Synthetic example for interprocedural redistribution optimization

Two warnings were also generated by the compiler, inserted by hand as comments in Figure 5.13(b), indicating that there were (semantically) multiple reaching distributions to two uses of `x` in the program. The first use actually does have two reaching distributions due to a conditional with redistribution performed on only one path. The second use, however, occurs after a call to a prescriptive function, `func1`, which implicitly redistributes the array to conform to its interface. Because a redistribution operation can only generate a single distribution, `x` can actually have only one reaching distribution, but semantically it still has two – hence the second warning.

As there are several other optimizations performed on this example, which have been described previously, we will not describe them in more detail here, but the reader is directed to the comment descriptions in the code for further information.

An example of a more realistic program is the ADI2D program (previously used to illustrate the data distribution optimizations in Chapter 4) shown in Figure 5.14 both before and after analyzing it using this framework. To make the program more challenging for the framework to analyze, the main computation in the program is broken into separate functions. In this program, two functions (`col_solve`, and `row_solve`) are called – one using transcriptive arguments so that it will inherit their distributions from the calling routine and the other using prescriptive arguments. In addition to the implicit redistribution that occurs when calling `row_solve`, each function also performs redistribution during its execution, potentially changing the input distribution and, if so, returning a different distribution than that with which it was called.

Examining the transformed output in Figure 5.14(b), the redistribution originally performed upon entering the loop in the main program has been lifted out of the nest as it was found to be invariant, being performed instead, only once outside the entire loop body. Another important point to note is that even though all redistribution operations are now explicit, there are no redistribution operations performed around the two call sites in the main program. Upon first glance, this does not seem correct since the second function `row_solve` had a prescriptive distribution that was different from the active distribution in the calling context of the main program. There is not any redistribution performed before calling `row_solve` because the first function `col_solve` actually redistributed the arrays from their initial state, leaving them in the

```
      program ADI2d
      parameter (N = 512, maxiter = 100)
      double precision u(N,N), uh(N,N), b(N,N), alpha
!hpf$ processors :: linear(32)
!hpf$ distribute (*, block) onto linear :: u, uh, b

c     *** Initialization removed
c     ...
      alpha = 4 * (2.0 / N)
      do k = 1, maxiter
!hpf$    redistribute (block, *) :: u, uh, b
         call col_solve(u, uh, b, N, alpha)
         call row_solve(u, uh, b, N, alpha)
      enddo
      end

      subroutine col_solve(u, uh, b, N, alpha)
      double precision u(N,N), uh(N,N), b(N,N), alpha
!hpf$ inherit :: u, uh, b
c        *** forward and backward sweeps along columns
         do j = 2, N - 1
            do i = 2, N - 1
               b(i,j) = (2 + alpha)
               uh(i,j) = (alpha - 2) * u(i,j)
     &                           + u(i,j+1) + u(i,j-1)
            enddo
         enddo
!hpf$    redistribute (*, block) :: u, uh, b
         do j = 2, N - 1
            uh(2,j) = uh(2,j) + u(1,j)
            uh(N-1,j) = uh(N-1,j) + u(N,j)
         enddo

         do j = 2, N - 1
            do i = 3, N-1
               b(i,j) = b(i,j) - 1 / b(i-1,j)
               uh(i,j) = uh(i,j) + uh(i-1,j) / b(i-1,j)
            enddo
         enddo
         do j = 2, N - 1
            uh(N-1,j) = uh(N-1,j) / b(N-1,j)
         enddo
         do j = 2, N - 1
            do i = N-2, 2, -1
               uh(i,j) = (uh(i,j) + uh(i+1,j)) / b(i,j)
            enddo
         enddo
      end

      subroutine row_solve(u, uh, b, N, alpha)
      double precision u(N,N), uh(N,N), b(N,N), alpha
!hpf$ distribute (*, block) :: u, uh, b
c        *** forward and backward sweeps along rows
         do j = 2, N - 1
            do i = 2, N - 1
               b(i,j) = (2 + alpha)
               u(i,j) = (alpha - 2) * uh(i,j)
     &                           + uh(i+1,j) + uh(i-1,j)
            enddo
         enddo
!hpf$    redistribute (block, *) :: u, uh, b
         do i = 2, N - 1
            u(i,2) = u(i,2) + uh(i,1)
            u(i,N-1) = u(i,N-1) + uh(i,N)
         enddo

         do j = 3, N-1
            do i = 2, N - 1
               b(i,j) = b(i,j) - 1 / b(i,j-1)
               u(i,j) = u(i,j) + u(i,j-1) / b(i,j-1)
            enddo
         enddo
         do i = 2, N - 1
            u(i,N-1) = u(i,N-1) / b(i,N-1)
         enddo
         do j = N-2, 2, -1
            do i = 2, N - 1
               u(i,j) = (u(i,j) + u(i,j+1)) / b(i,j)
            enddo
         enddo
      end
```

(a) Before optimization

```
      PROGRAM adi2d
      PARAMETER(n = 512, maxiter = 100)
      DOUBLE PRECISION u(n,n), uh(n,n), b(n,n)
      DOUBLE PRECISION alpha
!HPF$ PROCESSORS :: linear(32)
!HPF$ DISTRIBUTE (*, BLOCK) ONTO linear :: u, uh, b
c     *** Initialization removed
c     ...
      alpha = 4 * (2.0 / n)
!HPF$ REDISTRIBUTE (BLOCK, *) ONTO linear :: u, uh, b
      DO k = 1,maxiter
        CALL col_solve(u,uh,b,n,alpha)
        CALL row_solve(u,uh,b,n,alpha)
      END DO
      uh(1,1) = u(1,1)
      END

      SUBROUTINE col_solve(u,uh,b,n,alpha)
      DOUBLE PRECISION u(n,n), uh(n,n), b(n,n)
      DOUBLE PRECISION alpha
!HPF$ PROCESSORS :: linear(32)
!HPF$ DISTRIBUTE(BLOCK,*) ONTO linear :: u, uh, b
      DO j = 2,n - 1
        DO i = 2,n - 1
          b(i,j) = (2 + alpha)
          uh(i,j) = (alpha - 2) * u(i,j)
     &                    + u(i,j + 1) + u(i,j - 1)
        END DO
      END DO
!HPF$ REDISTRIBUTE (*, BLOCK) ONTO linear :: u, uh
      DO j = 2,n - 1
        uh(2,j) = uh(2,j) + u(1,j)
        uh(n - 1,j) = uh(n - 1,j) + u(n,j)
      END DO
!HPF$ REDISTRIBUTE (*, BLOCK) ONTO linear :: b
      DO j = 2,n - 1
        DO i = 3,n - 1
          b(i,j) = b(i,j) - 1 / b(i - 1,j)
          uh(i,j) = uh(i,j) + uh(i - 1,j) / b(i - 1,j)
        END DO
      END DO
      DO j = 2,n - 1
        uh(n - 1,j) = uh(n - 1,j) / b(n - 1,j)
      END DO
      DO j = 2,n - 1
        DO i = n - 2,2, -1
          uh(i,j) = (uh(i,j) + uh(i + 1,j)) / b(i,j)
        END DO
      END DO
      END

      SUBROUTINE row_solve(u,uh,b,n,alpha)
      DOUBLE PRECISION u(n,n), uh(n,n), b(n,n)
      DOUBLE PRECISION alpha
!HPF$ PROCESSORS :: linear(32)
!HPF$ DISTRIBUTE (*, BLOCK) ONTO linear :: u, uh, b
      DO j = 2,n - 1
        DO i = 2,n - 1
          b(i,j) = (2 + alpha)
          u(i,j) = (alpha - 2) * uh(i,j)
     &                    + uh(i + 1,j) + uh(i - 1,j)
        END DO
      END DO
!HPF$ REDISTRIBUTE (BLOCK, *) ONTO linear :: u, uh
      DO i = 2,n - 1
        u(i,2) = u(i,2) + uh(i,1)
        u(i,n - 1) = u(i,n - 1) + uh(i,n)
      END DO
!HPF$ REDISTRIBUTE (BLOCK, *) ONTO linear :: b
      DO j = 3,n - 1
        DO i = 2,n - 1
          b(i,j) = b(i,j) - 1 / b(i,j - 1)
          u(i,j) = u(i,j) + u(i,j - 1) / b(i,j - 1)
        END DO
      END DO
      DO i = 2,n - 1
        u(i,n - 1) = u(i,n - 1) / b(i,n - 1)
      END DO
      DO j = n - 2,2, -1
        DO i = 2,n - 1
          u(i,j) = (u(i,j) + u(i,j + 1)) / b(i,j)
        END DO
      END DO
      END
```

(b) After optimization

Figure 5.14: 2-D ADI with interprocedural redistribution optimization

state which the prescriptive interface for `row_solve` required for its arguments. As compared to a full copy-in/copy-out approach, the redistribution framework has maintained HPF semantics while still completely eliminating a pair of redistribution operations – redistributing from (*, BLOCK) to (BLOCK, *) after returning from `col_solve` and redistributing from (BLOCK, *) to (*, BLOCK) before calling `row_solve`.

It is also interesting to note the placement of the redistribution performed within the two functions `col_solve` and `row_solve`. Even though the redistribution of three arrays was originally specified as occurring at a specific point in the function, the redistribution of two of the arrays, `u` and `uh`, is performed at the original point while the redistribution of `b` is performed following an intervening loop nest. Here we are observing how the data-flow framework is locating redistribution operations just before the array is used in the new configuration (just in time, or lazy, redistribution). As previously discussed in Section 5.2.2, there is actually a window between the eager and lazy redistribution points over which the operation can be performed to obtain the same effect. If the overlapping of the eager/lazy windows of these three operations were considered, they could instead be collected at a single point, allowing their communication to be aggregated. As there is a tradeoff between the amount of buffering required to perform the communication and the amount of communication overhead eliminated due to the aggregation, this will be examined further in future work.

To further examine the optimization of loop invariant redistribution operations, another example is shown in Figure 5.15. In this example, a redistribution operation on `A` is performed at the deepest level of a nested loop. If there are no uses of `A` before the occurrence of the redistribution (and no further redistribution performed in the remainder of the loop), then `A` will always have a (BLOCK, BLOCK) distribution within the loop body. This situation is detected by the framework and the redistribution operation is re-synthesized to occur outside of the entire loop nest. It could be argued that even when it appeared within the loop, the underlying redistribution library could be written to be smart enough to only perform the redistribution when it is necessary (i.e., only on the first iteration) so that we have not really optimized away $\mathtt{N}^2$ redistribution operations. Even in this case, this optimization has still completely eliminated ($\mathtt{N}^2$-1)

```
!HPF$ DISTRIBUTE (CYCLIC, *) :: A
      ...
      DO i = 1,N
        DO j = 1,N
          ...
!HPF$ REDISTRIBUTE (BLOCK, BLOCK) :: A
          A(i,j) = ...
          ...
        ENDDO
      ENDDO
```

(a) Before optimization

```
!HPF$ DISTRIBUTE (CYCLIC, *) :: A
       ...
!HPF$ REDISTRIBUTE (BLOCK, BLOCK) :: A
      DO i = 1,N
        DO j = 1,N
          ...
          A(i,j) = ...
          ...
        END DO
      END DO
```

(b) After optimization

Figure 5.15: Example of loop invariant redistribution

check operations that would have been performed at run time to determine if the redistribution was required.

5.4 Summary

In this chapter, we have presented an interprocedural data-flow framework that provides a means to convert between a distribution-based representation (as specified by the data partitioner) and redistribution-based form (as specified by HPF) while optimizing the amount of redistribution that must be performed. In addition to supporting the data partitioner, this framework also allows us to optimize dynamic HPF programs (using `REDISTRIBUTE` and `REALIGN` directives) as well as convert such programs into equivalent static versions through a process we refer to as Static Distribution Assignment (SDA) providing dynamic HPF support for existing subset HPF compilers. Both the capability of generating redistribution-based codes as well as the SDA transformation will be instrumental in the evaluation of data distributions, which are presented in Chapter 6.

CHAPTER 6

EXPERIMENTAL RESULTS

In the three previous chapters, we described the algorithms for optimizing communication, data distribution and redistribution that we have implemented in the PARADIGM compiler framework. In this chapter we demonstrate these techniques using a number of benchmark kernels and larger example programs. The effect of the optimizations on the performance of the test programs is evaluated using an Intel Paragon and Thinking Machines CM-5.

6.1 Optimizing Communication

A group of small scientific programs are used to examine the performance of the presented communication optimizations. These programs include individual Fortran 77 kernels and subroutines which range in size from roughly 20 to 50 lines of code and exhibit stencil, or nearest-neighbor, access patterns. Since nearest-neighbor patterns will generate significant amounts of communication that can severely degrade performance if they are not properly optimized, they are good candidates for evaluating the automatic message combining and placement optimizations while their size is small enough to be able to reason about the decisions the compiler makes during the compilation process. The selected program fragments, in increasing order of complexity, include:

- Jacobi's Iterative Method [95]
- ADI Integration, Livermore kernel 8 (ADI) [106]

Table 6.1: Data partitioning for initial tests

	Array Dimensions	Data Distribution
Jacobi	1000×1000	BLOCK, BLOCK
ADI	$4 \times 10,000 \times 2$	$*$, BLOCK, $*$
EXPL	$10,000 \times 7$	BLOCK, $*$
EFLUX	5000×34	BLOCK, $*$

- 2-D Explicit Hydrodynamics, Livermore kernel 18 (EXPL) [106]
- Euler Fluxes (EFLUX, from FLO52 in the Perfect Club) [107]

Array dimensions are statically specified and loop bounds are determined at compile time when possible. Both the mesh configuration as well as the data partitioning of the arrays are automatically selected by the compiler. The sizes of the major arrays are shown in Table 6.1 along with the distributions chosen for each of the programs. Other than a few annotations which were added to EFLUX to specify that several scalar variables could be privatized.[1] The evaluation of the optimizations is performed using 16 processors of an Intel Paragon and a Thinking Machines CM-5. Traces are also taken on the Paragon using the Portable Instrumented Communication Library (PICL) [15].

PICL traces are analyzed using the ParaGraph [109] visualization tool to examine the effect of each optimization. The ParaGraph "Space time diagram" is used to visualize the execution profile to examine the frequency and amount of communication that take place. The space time diagram displays processors along the vertical axis and time along the horizontal. Continuous horizontal lines indicate uninterrupted execution while a break in a line indicates that a processor has blocked awaiting communication. Communication operations are indicated as vertical lines between the cooperating processors' execution profiles. Snapshot views of the execution of EXPL on an Intel iPSC/2 compiled with each of the selected optimizations are shown in each figure while performance data is presented for all test programs (see Figures 6.1 to 6.4).

[1] For EFLUX, privatization [108] is used for all optimizations except for run-time resolution. Scalar expansion was performed for the evaluation of run-time resolution in order to improve the determination of ownership at run-time no other directives were added to the programs.

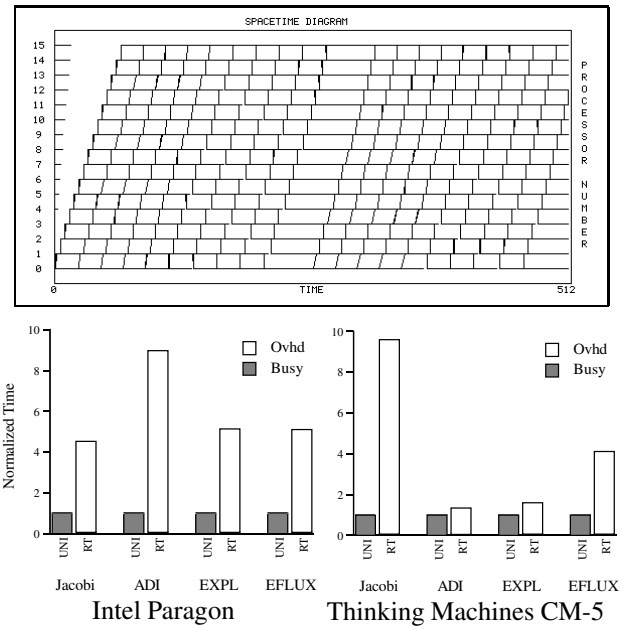


Figure 6.1: Run-time resolution

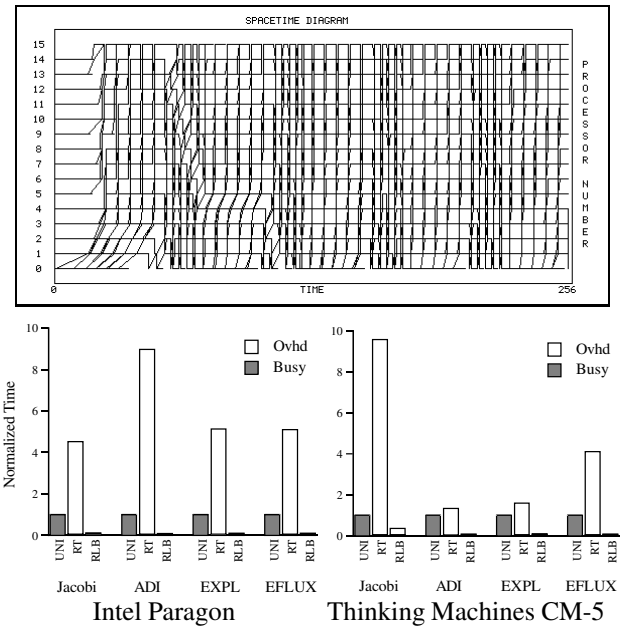


Figure 6.2: Reduced loop bounds and coalescing

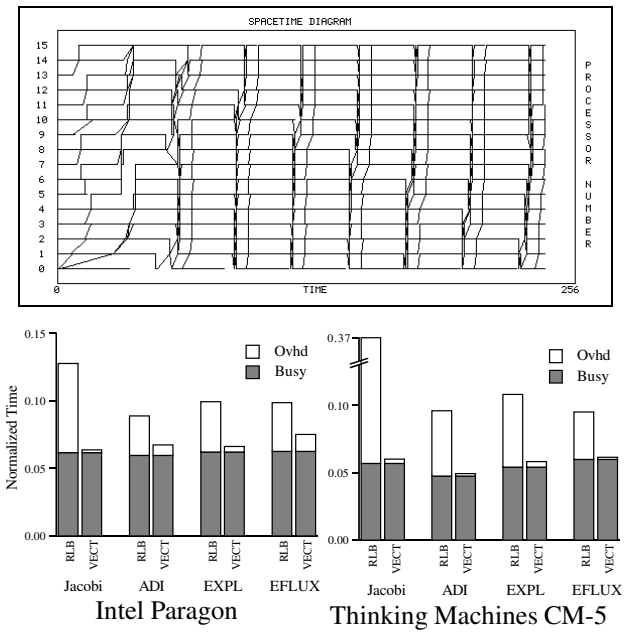


Figure 6.3: Message vectorization

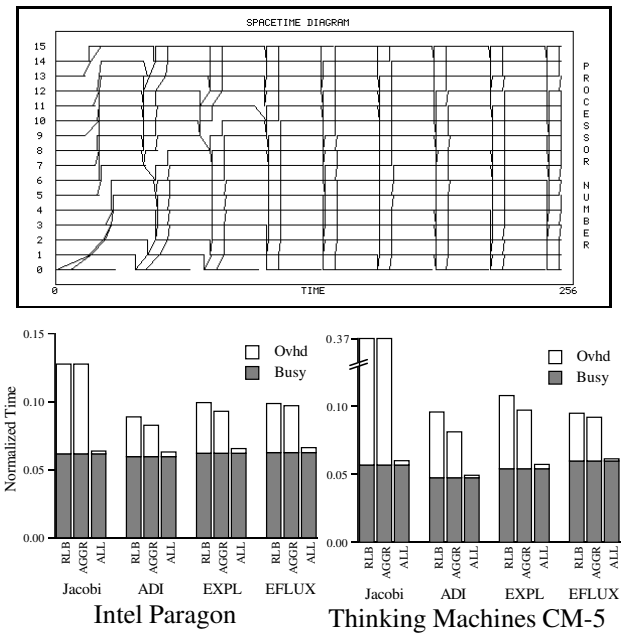


Figure 6.4: Message aggregation

UNI	Uniprocessor Execution	VECT	Message Vectorization
RT	Run-time Resolution	AGGR	Message Aggregation
RLB	Reduced Loop Bounds & Message Coalescing	ALL	All Optimizations applied

Traces taken from Explicit Hydrodynamics (Livermore kernel 18) on an Intel Paragon
(each time unit represents 1 ms, except for Figure 6.1 in which each is 10 ms)

For comparison purposes, the reported execution times have been normalized to the serial execution of the corresponding program (i.e., ideal speedup will equal $\frac{1}{16} = 0.0625$) and are further separated into two quantities:

- Busy – amount of time spent on *useful* computation (where *useful* refers only to the code which carries out the actual computation)
- Overhead – time spent executing code related to computation partitioning and communication

The relative effectiveness of each optimization can therefore be determined by examining the amount of overhead eliminated as the optimizations are incrementally applied.

Although several of the supported communication libraries provide tracing options, such profiling will only measure the portion of the overhead which is actually spent in communication operations. Other related costs (such as buffer packing and unpacking and the evaluation of communication masks) would be assigned to the busy time possibly giving a false impression of how well each optimization is performing. To avoid this problem, the busy time is instead computed by taking a partitioned version of each test program and removing all of the code related to communication and buffer management. The communication buffers are then initialized with appropriate data to keep the computation valid, and the elapsed time for the partitioned computation alone is measured. Even though the results of the partitioned computation alone will not match the actual solution computed by the full parallel program, the *useful* operation count will be identical, therefore, yielding a tight lower bound on how much improvement can be expected from the optimizations. Overhead is now simply the time that the parallel run takes in excess of the measured busy time.

6.1.1 Results of run-time resolution

A direct application of the owner computes rule without any further optimization leads to *run-time resolution* [9]. Since each processor must execute the entire iteration space (to determine if any other processor needs locally owned data), different instances of a communication operation for a given reference are effectively serialized across the processors in a mesh dimen-

sion (see the trace in Figure 6.1). Multiple communication operations appear to be pipelined, but the time spent between successive messages can be quite large.

Examining the execution time for each program, it can be seen that traversing the entire iteration space to compute ownership results in large amounts of overhead. The communication performed for run-time resolution is also very inefficient since it is comprised of a large number of single element (sometimes even redundant) messages resulting in high communication overhead. The net result is a reduction in performance when compared to the original program's serial performance.

6.1.2 Results of message coalescing and loop bounds reduction

By applying loop bounds reduction and statically generating communication operations [12, 89], the serialization present in the baseline resolution cases can be eliminated. The statically generated communication operations are effectively collapsed in time as compared to the serialized communication present in run-time resolution (see the trace in Figure 6.2). By statically generating communication operations, the loop bounds can be statically partitioned instead of requiring every processor to check for the possibility of communication for each reference over the entire iteration space at run time.

The overhead is dramatically reduced as all ownership and communication are now statically determined. Note, however, that the size of the messages is still identical to that in run-time resolution (single elements), but redundant messages have been eliminated by application of the coalescing optimization. It is still apparent that there is an excessive amount of small messages being communicated since communication overhead is still a dominant factor in the execution time (as seen in Figure 6.3).

6.1.3 Results of message vectorization

It is also possible to vectorize the communication operations after loop bounds reduction and message coalescing have been applied. Using dependence information to determine when vectorization is applicable, communication operations can be lifted out of the innermost loops

thereby reducing the communication frequency, as shown in the trace in Figure 6.3. Recall that this also increases the size of the messages, but since there is no change in the transmission rate for larger messages on either the Paragon or the CM-5, this results is a large gain in performance as many communication operations have been replaced by a single operation. As a secondary effect, it is also possible that the target node compiler can do a better job of optimizing the core computation since the communication operations have been moved as far out of the innermost loops as possible separating it from the bulk of the computation.

For ADI, it is actually possible to vectorize all communication completely out of the entire loop nest. Because there is no synchronization due to communication within the loop nest, the reduction of the array bounds in the absence of communication resulted in super-linear speedup for the CM-5. This can be attributed to a gain in cache performance due to the reduction in the size of the working set as a result of the data partitioning.

6.1.4 Results of message aggregation

Aggregation also reduces the frequency of communication by grouping multiple communication operations, increasing the size of the resulting message. (See the trace in Figure 6.4.) Applying aggregation after loop bounds reduction and message coalescing, a performance gain is seen for ADI, EXPL, and EFLUX. There was no improvement using aggregation on Jacobi's method since it communicated only one message for each source/destination pair.

By applying aggregation after vectorization, the communication overhead is further reduced as seen in the "thinning" of communication when comparing the traces in Figures 6.3 and 6.4. Since groups of messages to be communicated at the same point in the program are combined, the performance improvement is related to the number of different array sections that must be communicated. The scope of improvement is much smaller than vectorization for the test programs since the number of communication operations at any given point (anywhere from one to three) is much smaller in comparison to the iteration counts over which the messages were vectorized (related to the size of the array bounds along processor boundaries, see Table 6.1). If the properties were reversed, (a large number of arrays were to be communicated at a given point

but were not vectorizable, such as is possible with recurrences), it is possible that aggregation could have a larger impact on program performance than vectorization for some programs.

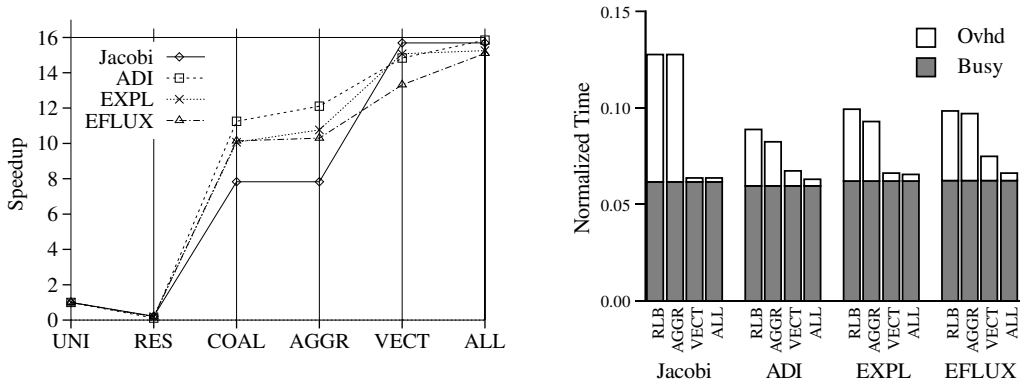
6.1.5 Results of all non-pipelined optimizations

Figure 6.5 shows the relative speedup and the amount of overhead while Table 6.2 reports the efficiencies measured for each optimization. Eliminating the overhead of traversing the entire iteration space to compute ownership, the combination of static communication generation and loop bounds reduction attained roughly half of the total available performance for most of the programs. By applying message vectorization it was possible to reduce most of the remaining communication overhead resulting in near-ideal performance while message aggregation was beneficial for those programs that contained references to a number of different arrays.

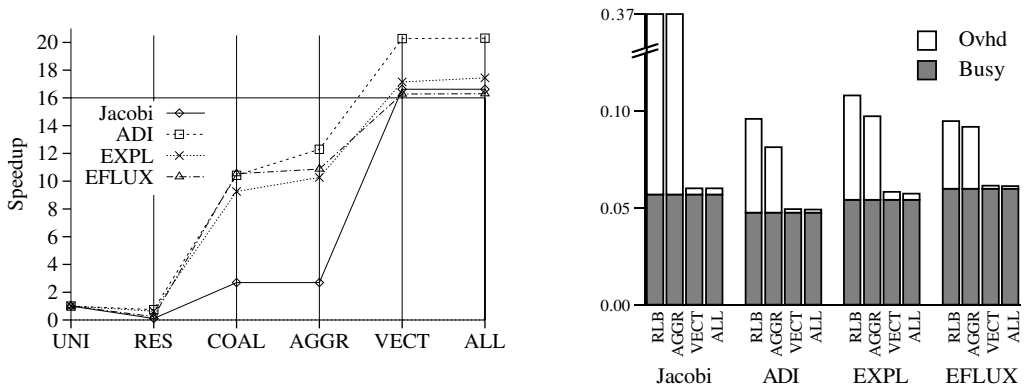
It is also interesting to compare the performance of the Paragon with results previously obtained for the Intel iPSC/2 (second generation hypercube) and the iPSC/860 (third generation hypercube) [13]. The iPSC/2 has a 16MHz i386/387, the iPSC/860 has a 40MHz i860, and the Paragon has a 50MHz i860 with a second i860 for handling communication operations. The two hypercubes have identical interconnection networks with similar communication bandwidths, but have dramatically differing computational capabilities. The Paragon has similar computation performance compared to the iPSC/860 but with a much higher communication bandwidth. When comparing the iPSC/860 to the iPSC/2, the change in communication performance is about a factor of 5 while the change in computation is about a factor of 100. When comparing the Paragon to the iPSC/860, the change in communication is about a factor of 20 for large messages while the computation capabilities improve by only roughly 25%. For this reason, on the iPSC/860 the ratio of the cost of communication to computation is fairly high compared to the iPSC/2 or the Paragon. Comparing Figure 6.5(a) to the previous results [13], the Paragon and the iPSC/2 exhibit similar ratios between the busy and overhead times while the iPSC/860 tends to exhibit higher percentages of overhead (which can be expected when examining the above factors). From these observations, it is apparent that performing communication optimizations becomes more critical when the computational capabilities are increased more quickly than the

Table 6.2: Program efficiencies vs. optimization

		RT	RLB	AGGR	VECT	ALL
Paragon	Jacobi	1.4%	48.9%	48.9%	98.1%	98.1%
	ADI	0.7%	70.4%	75.7%	92.7%	99.1%
	EXPL	1.2%	62.9%	67.3%	94.2%	95.3%
	EFLUX	1.2%	63.4%	64.4%	83.3%	94.3%
CM-5	Jacobi	0.7%	16.8%	16.8%	103.9%	103.9%
	ADI	4.7%	65.2%	76.9%	126.7%	126.9%
	EXPL	3.9%	57.9%	64.2%	107.2%	109.0%
	EFLUX	1.5%	65.8%	68.0%	101.7%	101.9%



(a) Intel Paragon (16 processors)



(b) Thinking Machines CM-5 (16 processors)

Figure 6.5: Comparison of combined optimizations

communication (iPSC/2 $\rightarrow$ iPSC/860) or the communication capabilities decrease in comparison to computation (Paragon $\rightarrow$ iPSC/860).

To examine their scalability, each of the fully optimized test programs were also executed with varying numbers of processors. In Table 6.3, each program's mesh configuration is shown as selected by the automatic partitioning pass while the speedup curves for each run on the Paragon and the CM-5 are shown in Figure 6.6. The super-linear speedup is again observed for ADI on the CM-5 and can be attributed to the cache effect previously described in Section 6.1.3.

The performance of the test programs was also compared to an existing data-parallel compiler available on the CM-5 by converting them into a data-parallel language known as Connection Machine Fortran (CMF 2.1-2).[2] The reduction in performance of the programs compiled with CMF can be attributed to the fact that it uses an SIMD (Single Instruction Multiple Data) model of execution for compilation (carried over from the CM-2). Simulating an SIMD execution model on a MIMD multicomputer (such as the CM-5) does not require every operation to be executed in full synchronization, but entails synchronizing the processors between blocks of computation at points of communication. This comparison demonstrates how, for a MIMD style architecture such as the CM-5, more performance can be gained through the use of a less synchronous model of computation (SPMD) with the application of these optimizations.

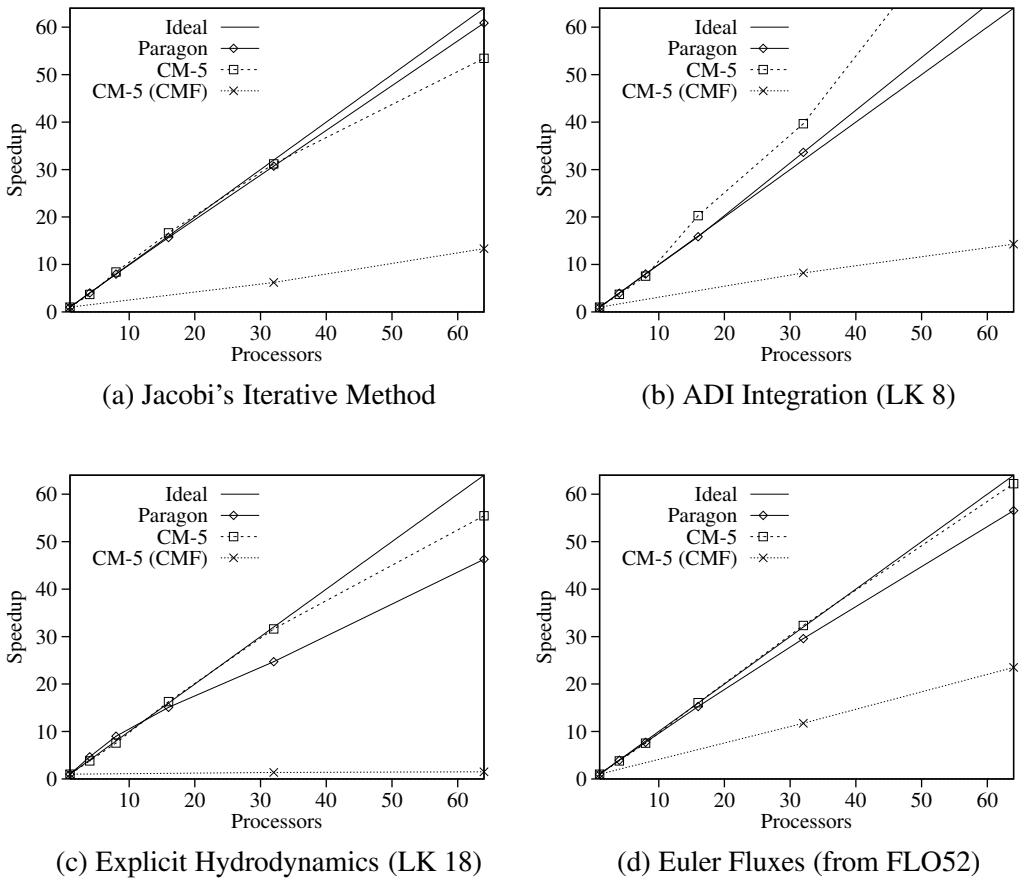
6.1.6 Results of message pipelining

To evaluate the quality of the strip size estimate developed in Section 3.4, a group of programs which require pipelining to extract parallelism are selected. These programs include individual Fortran 77 kernels and subroutines which range in size from roughly 15 to 60 lines of code (the amount of computation performed within the inner loop increasing with the complexity of the programs) and all exhibit inner-loop cross-iteration dependencies which give rise to pipelined communication. Also, since the pipelining transformation has not yet been completely integrated into the compiler, their size is small enough to be able to examine the loop partitioning performed by PARADIGM and insert the pipelined communication operations by

[2]The CM-5 vector units were not utilized for either the CMF or optimized message-passing runs since the C and Fortran node compilers could not generate vector code for the message passing programs.

Table 6.3: Mesh configurations

Test	Procs.	Mesh	Test	Procs.	Mesh
ADI EXPL EFLUX	1	1×1	Jacobi	1	1×1
	2	2×1		2	2×1
	4	4×1		4	4×1
	8	8×1		8	4×2
	16	16×1		16	4×4
	32	32×1		32	8×4
	64	64×1		64	8×8



(a) Jacobi's Iterative Method

(b) ADI Integration (LK 8)

(c) Explicit Hydrodynamics (LK 18)

(d) Euler Fluxes (from FLO52)

Figure 6.6: Performance comparison

Table 6.4: Data partitioning for pipeline tests

	L	Array Dimensions	Data Distribution
SOR	100	512×512	BLOCK, $*$
IMPL	100	512×512	$*$, BLOCK
ADI2D	100	512×512	BLOCK, $*$
BLTS	22	$[5\times]5 \times 32 \times 24 \times 24$	[$*$,]$*$, BLOCK, $*$, $*$

hand based on information provided by the compiler. The selected program fragments, in increasing order of the amount of computation performed within the pipelined loop, include:

- Successive Over-Relaxation Iterative Method (SOR) [95]
- Implicit Hydrodynamics, Livermore kernel 23 (IMPL) [106]
- 2-D Alternating Direction Implicit (ADI2D[3]) iterative method [95]
- Block Lower Triangular Solver (BLTS, from APPLU in the NAS benchmarks) [110]

SOR and IMPL fit the two-level model quite well while ADI2D and BLTS contain different properties. For ADI2D, even though there is an outer loop, there is actual no chaining between successive pipelines since they are separated by parallel sections containing bidirectional shift communication operations (which tend to serve as primitive self-synchronizing barriers). Therefore, L is set to 1 for the two-level ADI2D estimate. For BLTS, the computation does not contain any forward references so there is not any synchronization between successive pipelines. For this reason, the non-synchronizing form of the two-level estimate (Eq. (3.4)) will be used for BLTS.

The sizes of the major arrays and the chosen distributions are shown in Table 6.4 for each of the programs.[4] Each of the programs was executed with varying strip sizes to compare the

[3]Even though their names are similar, it is important to note that the computation performed in ADI2D is very different from the computation performed in the ADI kernel previously used in Section 6.1. One main difference is that several of the accesses within ADI2D exhibit cross-iteration dependencies while ADI is fully parallel.

[4]In order to prevent poor serial performance from cache-line aliasing due to the power of two problem size, the arrays were also padded with an extra element at the end of each column. This optimization, although here performed by hand, is automated by aggressive serial optimizing compilers such as the KAP preprocessor from KAI.

Table 6.5: Comparison of the granularity estimates to the optimal granularity (Reporting both the strip size and time in seconds)

		Procs.	Fine	Optimal		$\frac{\text{Fine}}{\text{Opt.}}$	One-Level		$\%\Delta_{\text{Opt}}$	Two-Level		$\%\Delta_{\text{Opt}}$
Paragon	SOR	4	9.00	32	4.28	2.10	10	4.49	4.9%	20	4.31	0.7%
		8	7.44	32	2.58	2.88	10	2.84	10.1%	27	2.60	0.8%
		16	6.69	44	1.71	3.91	10	2.02	18.1%	37	1.73	1.2%
	IMPL	4	56.86	40	17.67	3.22	8	21.71	22.9%	16	18.44	4.4%
		8	30.13	34	11.18	2.69	8	13.06	16.8%	22	11.41	2.1%
		16	16.85	34	6.23	2.70	8	7.26	16.5%	30	6.24	0.2%
	ADI	4	181.44	6	175.04	1.03	9	175.87	0.5%	10	176.20	0.7%
		8	96.47	4	90.78	1.06	9	92.04	1.4%	10	92.44	1.8%
		16	54.49	4	48.89	1.11	9	50.58	3.5%	9	50.58	3.5%
	BLTS	4	0.50	10	0.42	1.19	1	0.50	20.2%	6	0.42	0.5%
		8	0.27	6	0.23	1.17	1	0.26	16.5%	6	0.23	0 %
		16	0.16	6	0.13	1.25	1	0.16	20.7%	7	0.15	8.9%
CM-5	SOR	4	22.05	32	10.02	2.20	10	10.61	5.9%	20	10.12	1.0%
		8	17.99	34	5.82	3.09	10	6.53	12.2%	28	5.88	1.0%
		16	15.83	64	3.48	4.55	10	4.28	23.0%	37	3.50	0.6%
	IMPL	4	114.58	64	20.98	5.46	8	30.23	44.1%	17	23.97	14.3%
		8	57.58	64	10.90	5.28	8	15.65	43.6%	23	11.71	7.4%
		16	36.64	58	5.66	6.47	8	8.80	55.5%	31	5.97	5.5%
	ADI	4	133.91	10	113.13	1.18	9	113.33	0.2%	11	113.21	0.1%
		8	81.44	10	61.78	1.32	9	61.84	0.1%	10	61.78	0 %
		16	54.26	10	32.86	1.65	9	33.29	1.3%	10	32.86	0 %
	BLTS	4	0.84	11	0.72	1.17	1	0.84	16.9%	6	0.72	0.4%
		8	0.59	8	0.43	1.38	1	0.59	37.8%	7	0.44	1.4%
		16	0.49	6	0.27	1.80	1	0.49	77.7%	8	0.28	0.7%

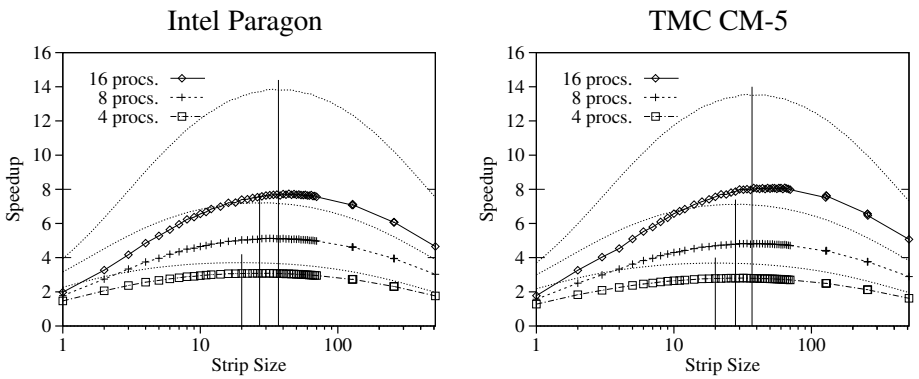
actual performance of the pipelined programs to the estimates. Both the two-level strip size estimate (Eq. (3.3)) as well as the simpler one-level strip size estimate (Eq. (3.5)), developed in the Fortran D project, will be compared to the experimental results.

The optimal granularity is compared with the two forms of strip size estimates in Table 6.5. Both the selected strip size and resulting execution time are presented for each. The speedup achieved using the optimal granularity as compared to fine-grained execution (strip size of one) is also reported as well as the change in performance for each of the estimates as compared to the optimal.

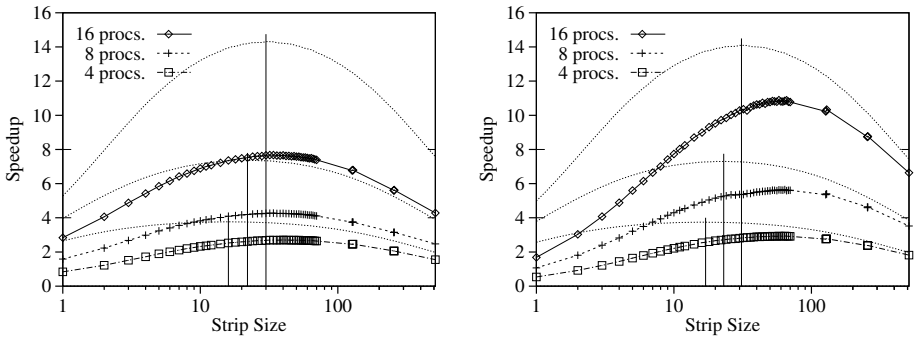
For both the Paragon and the CM-5, the execution time using the two-level estimate is usually within a few percent of the optimal (except for IMPL on the CM-5 for which the estimate of the computational cost was not very accurate). Note that even though strip sizes predicted by the two-level estimate are not exactly the same as for the optimal, the execution times tend to be very similar because there is a range of strip sizes for which the performance curves are fairly flat. For ADI2D, both models did quite well because the pipelines could not be chained for this program. Overall, however, the one-level estimate was, at times, more than 20% (and in some case over 70%) worse than the optimal, predicting strip sizes of roughly half to a third of the size of the two-level estimate. In fact, a further approximation is made in the Fortran D project, resulting in an estimate of $\sqrt{\frac{ovhd}{c}}$ which proves to be even farther from the minimum.

As can be expected, the major gain seen with the two-level estimate comes from modeling the chaining of consecutive pipelines. In the one-level estimate, the pipeline startup costs have a more significant contribution and tend to reduce the strip size in order to minimize the total execution time. When chaining is taken into account in the two-level estimate, the contribution of the startup phase is much less in comparison to the overall execution time. For both machines examined, however, modeling the communication rate did not have a significant effect on the two-level estimate since only small messages were communicated (from Table 3.1, the overhead of communication was two to three orders of magnitude greater than the transmission rate). For machines in which the ratio of the communication overhead to the transmission rate is decreased, the effect of the communication rate on the model will become more significant.

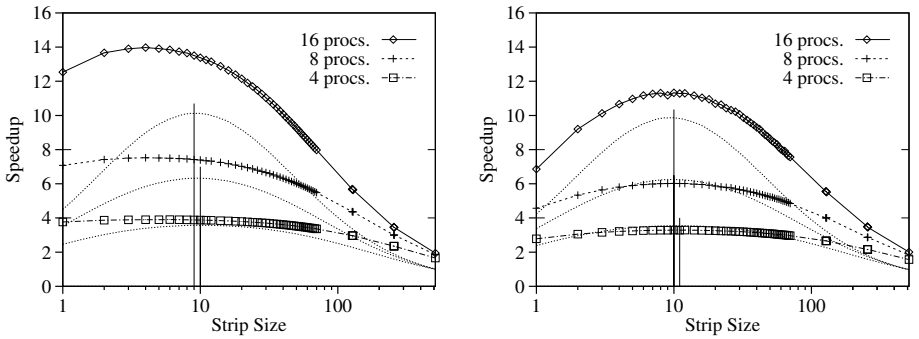
In Figure 6.7, speedup curves are shown for the measured data as well as that predicted by the two-level estimate (along with the predicted optimal granularity). The predicted speedup curves in Figure 6.7 were computed by dividing the pipeline estimate by the estimated serial time. Since there tends to be a range of strip sizes which all provide similar levels of performance, predicting the trend is more important than exactly predicting the optimal point. With the correct trend, it is possible to automatically select a granularity that approaches the minimum execution time. The computation estimates for the inner loop instructions usually differed from the experimentally measured value by a constant factor, but even though they were



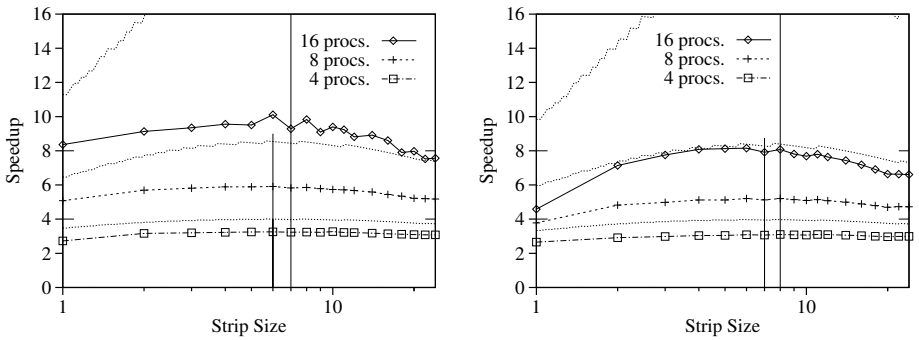
(a) Successive Over-Relaxation



(b) Implicit Hydrodynamics (Livermore kernel 23)

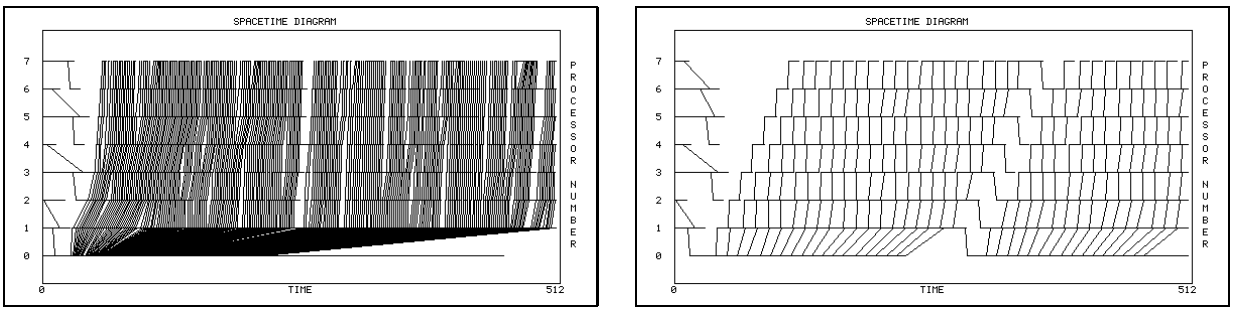


(c) 2-D Alternating Direction Implicit Iterative Method



(d) Sparse Block Lower Triangular Solver

Figure 6.7: Performance comparison of pipelining vs. granularity
(shown with performance estimates and the predicted optimal granularities)



(a) Fine-grain pipelining (b) Coarse-grain pipelining

Figure 6.8: Traces from SOR with pipelining on an Intel Paragon
(each time unit represents 100 μs)

not extremely accurate it can be seen from the graphs that the pipeline model still followed the performance trend quite well.

Comparing the performance of the different applications, it is possible to see that as the amount of computation performed within the pipelined loop is increased, the optimal granularity decreases, as would be expected. By further examining the estimates, it is also interesting to note that the granularity required to achieve the minimum execution time will slightly increase with increasing machine sizes when the data set size remains constant. Based on this observation, the optimal granularity should also increase for decreasing data set sizes when the machine size remains constant. Both will effectively reduce the amount of data owned by each processor and, therefore, also reduce the amount of computation that each processor must perform while the amount of communication remains constant.[5] This serves to confirm how, intuitively, granularity must be increased to compensate for an increasing communication to computation ratio.

Finally, traces from the SOR kernel run on the Paragon (again using PICL [15] and ParaGraph [109]) are shown in Figure 6.8. By examining these traces, the improvement in performance can also be seen as the coarse grain execution (using the granularity selected by the two-level estimate) has completed the first pipeline and is working on the second while the fine grain

[5]It was not always possible to observe this trend in the measured data most likely due to improvements in cache utilization as the size of the working set is reduced.

execution is still working on the first. These figures also show the correlation of the execution profile to the framework presented in Section 3.4.

6.2 Optimizing Data Distribution and Redistribution

To further evaluate the quality of the data distributions selected using the techniques presented in Section 4.3 as implemented in the PARADIGM compiler, we analyze two programs which exhibit different access patterns over the course of their execution. These programs are individual Fortran 77 subroutines which range in size from roughly 60 to 150 lines of code:

- 2-D Alternating Direction Implicit iterative method (ADI2D) [95]
- Shallow water weather prediction benchmark [111]

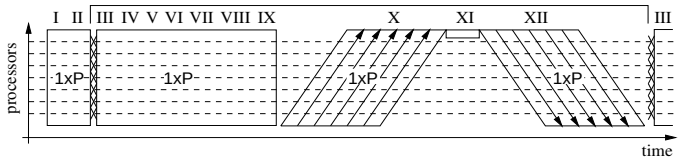
6.2.1 2-D Alternating Direction Implicit (ADI2D) iterative method

In order to evaluate the effectiveness of dynamic distributions, the ADI2D program, with a problem size of 512×512,[6] is compiled with a fully static distribution (one iteration shown in Figure 6.9(a)) as well as with the selected dynamic distribution[7] (one iteration shown in Figure 6.9(b)). These two parallel versions of the code were run on an Intel Paragon and a Thinking Machines CM-5 to examine the performance of each on the different architectures.

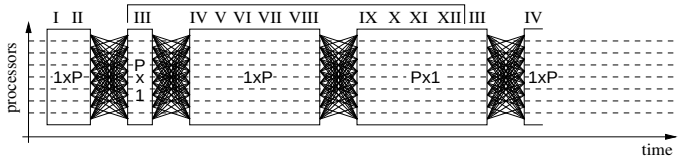
The static scheme illustrated in Figure 6.9(a) performs a shift operation to initially obtain some required data and then satisfies two recurrences in the program using software pipelining [10, 13]. Since values are being propagated through the array during the pipelined computation, processors must wait for results to be computed before continuing with their own part of the computation. The amount of computation performed before communicating to the next

[6] To prevent poor serial performance from cache-line aliasing due to the power of two problem size, the arrays were also padded with an extra element at the end of each column. This optimization, although here performed by hand, is automated by aggressive serial optimizing compilers such as the KAP preprocessor from KAI.

[7] In the current implementation, loop peeling is not performed on the actual code. As previously mentioned in Section 4.3.4, the single additional startup redistribution due to not peeling will not be significant in comparison to the execution of the loop (containing a dynamic count of 600 redistributions)



(a) Static (pipelined)



(b) Dynamic (redistribution)

Figure 6.9: Modes of parallel execution for ADI2D

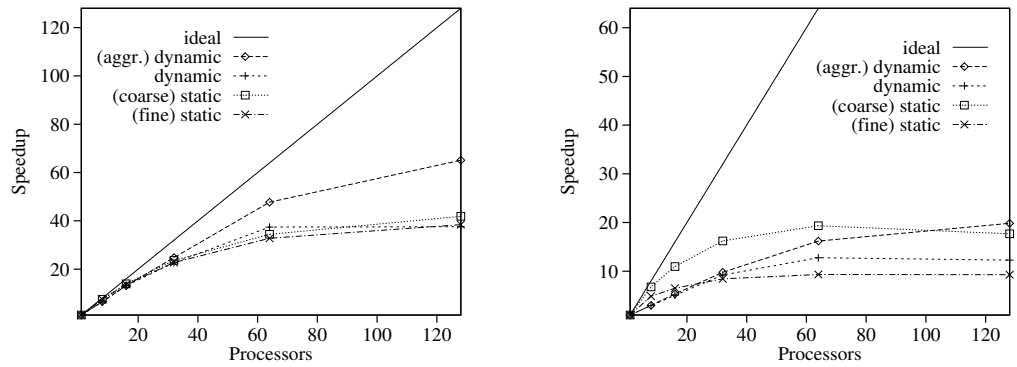
processor in the pipeline will have a direct effect[8] on the overall performance of a pipelined computation. Since we had previously shown, in Figure 6.7(c), that the performance of a static partitioning for ADI can be improved for both the CM-5 and Paragon, both a fine-grain and the optimal coarse-grain static partitioning will be compared with the dynamic partitioning.

The redistribution present in the dynamic scheme appears as three transposes[9] performed at two points within an outer loop (the exact points in the program can be seen in Figure 4.8). Since the sets of transposes occur at the same point in the program, the data to be communicated for each transpose can be aggregated into a single message during the actual transpose. It has been previously shown that aggregating communication improves performance by reducing the overhead of communication [13], so we will also examine aggregating the individual transpose operations here.

In Figure 6.10, the performance of both dynamic and static partitionings for ADI2D is shown for an Intel Paragon and a Thinking Machines CM-5. For the dynamic partitioning, both aggregated and non-aggregated transpose operations were compared. For both machines, it is apparent that aggregating the transpose communication is very effective, especially as the program is

[8] According to the ratio of communication vs. computation performance for a given machine.

[9] This could have been reduced to two transposes at each point if we allowed the cuts to reorder statements and perform loop distribution on the innermost loops (between statements 17, 18 and 41, 42), but these optimizations are not examined here.



(a) Intel Paragon (b) TMC CM-5

Figure 6.10: Performance of ADI2D

executed on larger numbers of processors. This can be attributed to the fact that the start-up cost of communication (which can be several orders of magnitude greater than the per byte transmission cost) is being amortized over multiple messages with the same source and destination.

For the static partitioning, the performance of the fine-grain pipeline was compared to a coarse-grain pipeline using the optimal granularity. The coarse-grain optimization yielded the greatest benefit on the CM-5 while still improving the performance (to a lesser degree) on the Paragon. For the Paragon, the dynamic partitioning with aggregation clearly improved performance (by over 70% compared to the fine-grain and 60% compared to the coarse-grain static distribution). On the CM-5, the dynamic partitioning with aggregation showed performance gains of over a factor of two compared to the fine-grain static partitioning but only outperformed the coarse-grain version for extremely large numbers of processors. For this reason, it would appear that the limiting factor on the CM-5 is the performance of the communication.

As previously mentioned in Section 4.3.4, the static partitioner currently makes a very conservative estimate for the execution cost of pipelined loops [8]. For this reason a dynamic partitioning was selected for both the Paragon as well as the CM-5. If a more accurate pipelined cost model [13] were used, a static partitioning would have been selected instead for the CM-5. For the Paragon, the cost of redistribution is still low enough that a dynamic partitioning would still be selected for machine configurations with a large number of processors.

To complete our analysis of ADI2D, it is also interesting to estimate the cost of performing a single transpose in either direction (P$\times$1 $\leftrightarrow$ 1$\times$P) from the communication overhead present in the dynamic runs. Ignoring any performance gains from cache effects, the communication overhead can be computed by subtracting the ideal run time (serial time divided by the selected number of processors) from the measured run time. Given that three arrays are transposed 200 times, the resulting overhead divided by 600 yields a rough estimate of how much time is required to redistribute a single array as shown in Table 6.6.

From Table 6.6 it can be seen that as more processors are involved in the operation, the time taken to perform one transpose levels off until a certain number of processors is reached. After this point, the amount of data being handled by each individual processor is small enough that the startup overhead of the communication has become the controlling factor. Aggregating the redistribution operations minimizes this effect thereby achieving higher levels of performance than would be possible otherwise.

Table 6.6: Empirically estimated time (ms) to transpose a 1-D partitioned matrix (512×512 elements; double precision)

	Intel Paragon		TMC CM-5	
processors	individual	aggregated	individual	aggregated
8	36.7	32.0	138.9	134.7
16	15.7	15.6	86.8	80.5
32	14.8	10.5	49.6	45.8
64	12.7	6.2	40.4	29.7
128	21.6	8.7	47.5	27.4

6.2.2 Shallow water weather prediction benchmark

Since not all programs will necessarily need dynamic distributions, we also examine another program which exhibits several different phases of computation. The Shallow water benchmark is a weather prediction program using finite difference models of the shallow water equations [111] written by Paul Swarztrauber from the National Center for Atmospheric Research.

As the program consists of a number of different functions, the program is first inlined since data partitioner is not yet fully interprocedural. Also, a loop which is implicitly formed by a `GOTO` statement is replaced with an explicit loop since the current performance estimation framework does not handle unstructured code. The final input program, ignoring comments and declarations, resulted in 143 lines of executable code.

In Figure 6.11, the phase transition graph is shown with the selected path using costs based on a 32 processor Intel Paragon with the original problem size of 257×257 limited to 100 iterations. The decomposition resulting in this graph was purposely bounded only by the productivity of the cut, and not by a minimum cost of redistribution in order to expose all potentially beneficial phases. This graph shows that by using the decomposition technique presented in Figure 4.9, Shallow contains six phases (the length of the path from the start to stop node) with a maximum of four (sometimes redundant) candidates for any given phase.

As there were no alignment conflicts in the program, and only `BLOCK` distributions were necessary to maintain a balanced load, the distribution of the 14 arrays in the program can be inferred from the selected configuration of the processor mesh. By tracing the path back from the stop node to the start, the figure shows that the dynamic partitioner selected a two-dimensional (8×4) static data distribution. Since there is no redistribution performed along this path, the loop peeling process previously described in Section 4.3.4 is not performed on this graph to improve its clarity.[10]

As the communication and computation estimates are best-case approximations (they don't take into account communication buffering operations or effects of the memory hierarchy), it is safe to say that for the Paragon, a dynamic data distribution does not exist which can out-perform the selected static distribution. Theoretically, if the communication cost for a machine were insignificant in comparison to the performance of computation, redistributing data between the other phases revealed during the decomposition of Shallow would be beneficial. The dynamic partitioner performed more work to come to the same conclusion that a single application of the static partitioner would have found, but it is interesting to note that even though it was con-

[10] As peeling is only used to model the possibility of implicit redistribution due to the back-edge of a loop, it is only necessary when there is actually redistribution present within a loop.

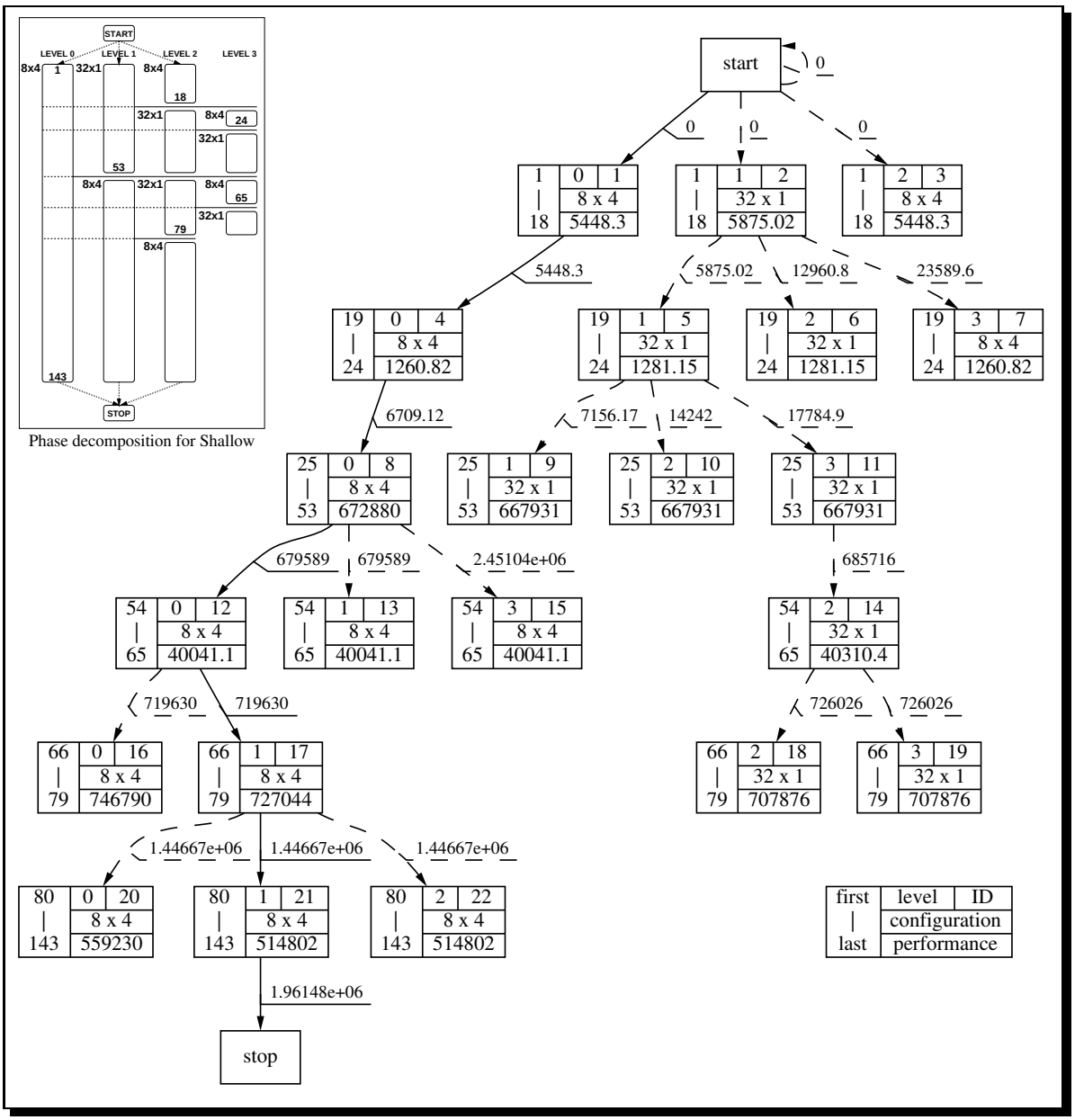
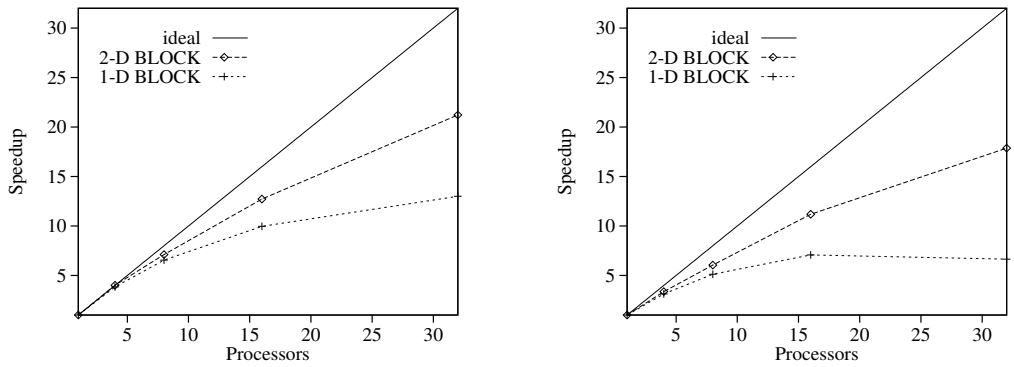


Figure 6.11: Phase transition graph and solution for Shallow
(Displayed with the selected phase transition path and cummulative cost.)



(a) Intel Paragon (b) TMC CM-5

Figure 6.12: Performance of Shallow

sidering any possible redistribution, it will still select a static data distribution if that is what is predicted to have the best performance.

To evaluate the performance of the selected distribution, a parallel MPI program is generated using the prototype PARADIGM compiler [11, 89]. Since the PARADIGM compiler does not fully support statements with differing left-hand side index expressions for partitioned array dimensions, loop distribution was first performed by hand on several loops to remove the need for any masks around such statements. Shallow also contains a number of loops that simply copy data from one region of the array to another. When such a loop is parallelized, vectorized communication operations are generated by PARADIGM, which perform the data movement directly between the local address spaces of the communicating processors. However, PARADIGM does not currently detect when such loops are made unnecessary by the data movement of the communication so such loops were also removed by hand after the parallel program was generated.

In Figure 6.12, the performance of the selected static 2-D distribution (`BLOCK`, `BLOCK`) is compared to a static 1-D row-wise distribution (`BLOCK`, $*$) which appeared in some of the subphases. The 2-D distribution matches the performance of the 1-D distribution for small numbers of processors while outperforming the 1-D distribution for both the Paragon and the CM-5 (by up to a factor of 1.6 or 2.7, respectively) for larger numbers of processors.

Kennedy and Kremer have also examined the Shallow benchmark, but predicted that a one-dimensional (column-wise) block distribution was the best distribution [112] for up to 32 processors of an Intel iPSC/860 hypercube (also showing that the performance of a 1-D column-wise distribution was almost identical a 1-D row-wise distribution). They chose this distribution because they only exhaustively enumerated one-dimensional candidate layouts for the operational phases.[11] As their operational definition already results in 28 phases (over four times as many in comparison to our approach), the complexity of their resulting 0-1 integer programming formulation will also only increase further by considering multidimensional layouts.

6.3 Summary

In this chapter, we have presented an experimental evaluation of the optimization techniques presented in this thesis. All of these techniques have been implemented in the PARADIGM compiler framework except for the coarse-grain pipelining transformation and optimal granularity selection technique which were performed by hand simulating the actions that would have been taken by the compiler. In the final chapter, Chapter 7, we present our conclusions on this work, and discuss directions for future work in this area.

[11] They justify this decision [67] since the Fortran D prototype compiler can not compile multidimensional distributions [80].

CHAPTER 7

CONCLUSIONS

In this thesis, we have presented our contributions which were focused in three main areas of performance optimization compilation techniques for distributed-memory multicomputers, each of which was shown to build upon the previous:

(1) Optimizing communication,

(2) Optimizing data distribution using the available communication optimizations, and

(3) Optimizing redistribution for dynamic data distributions.

7.1 Summary of Contributions

One of the most complex tasks facing a user when writing parallel programs for distributed-memory machines (using a message-passing programming model) is efficiently managing interprocessor communication. In this thesis, it has been shown that this task can be performed by the compiler at a higher level through the use of static analysis and good estimates of the costs of communication and computation. It has also been shown that the application of the presented optimization techniques yields high performance on several different distributed-memory multicomputers. By applying the presented optimizations, it was possible to amortize communication overhead obtaining large gains in performance. As observed, the performance of the optimizations is directly influenced by the relative costs of communication and computation on a given machine. The estimates presented in this thesis account for variation in both computa-

tional power as well as the communication latency and bandwidth of different machines. By comparing the estimated optimal granularity for message pipelining to that for the measured data, it is also apparent that automatic selection of a granularity that gives near-optimal performance is possible.

Currently, the PARADIGM compiler can generate programs based upon the owner-computes rule using either run-time resolution or loop bounds reduction through the use of static program analysis. The message coalescing, message vectorization, and message aggregation optimizations have all been implemented as described in this thesis. For this research, the coarse grain pipeline transformation and optimal granularity selection technique were performed by hand simulating the actions that would have been taken by the compiler. In future work, these techniques will be integrated into the compiler.

We have also demonstrated how dynamic data partitionings can provide higher performance than purely static distributions for programs containing competing data access patterns. The distribution selection technique presented in this thesis provides a means of automatically determining high quality data distributions (dynamic as well as static) in an efficient manner taking into account both the structure of the input program as well as the architectural parameters of the target machine. Heuristics, based on the observation that high communication costs are a result of not being able to statically align every reference in complex programs simultaneously, are used to form the communication graph. By removing constraints between competing sections of the program, better distributions can potentially be obtained for the individual sections and if the gains in performance are high enough in comparison to the cost of redistribution, dynamic distributions are formed. Communication still occurs, but data movement is now a dynamic reorganization of ownership as opposed to simply obtaining any required remote data based on a (compromise) static assignment of ownership.

A key requirement in automating this selection process is to be able to obtain estimates of communication and computation costs which accurately model the behavior of the program under a given distribution. Furthermore, by building upon existing static partitioning techniques, the number of phases examined as well as the amount of redistribution considered are kept to a minimum. Other than the additional analysis required for handling cut windows during the

phase decomposition process, and some additional support required for automatically performing loop peeling in the phase transition graph, these techniques have been fully implemented in the PARADIGM compiler.

In this thesis we have also presented an interprocedural data-flow technique that can be used to transform redistribution to distributions or distributions to redistribution as well as optimize redistribution while always maintaining the semantics of the original program. Aliasing information is also used to ensure that the resulting distributions specified by the HPF program yield identical distributions for all aliased arrays. Other than for the support to physically perform redistribution in the middle of a statement (required for a return from a function call with different distributions) and support for array reshaping (which can occur through function parameters), these techniques have been fully implemented in the PARADIGM compiler.

HPF compilers that are currently available as commercial products [113–116] or those that have been developed as research prototypes [3, 80, 117, 118] do not yet support transcriptive argument passing or the `REDISTRIBUTE` and `REALIGN` directives as there is still much work required to provide efficient support for the HPF subset (which does not include these features). Since the techniques presented in this thesis can convert all of these features into HPF constructs which are in the HPF subset (through the use of SDA), this framework can be used to provide these features as a front end to an existing commercial or research compiler.

7.2 Future Work

In this research, we have covered a wide spectrum of interrelated optimization problems from communication optimization to distribution selection and redundant redistribution elimination. The techniques we have developed and implemented as part of the PARADIGM compiler framework form a solid basis from which a number of different research directions could be pursued in any of the areas of program optimization addressed in this thesis.

7.2.1 Optimizing communication

For this research, the coarse-grain pipeline transformation and granularity selection technique were performed by hand since the prototype PARADIGM compiler does not currently detect opportunities for pipelining. Future research will examine different approaches for determining when the pipeline transformation is valid and automatically applying the granularity optimization technique as presented in this thesis.

The communication optimizations presented in this thesis currently optimize communication within a loop nest. By extending these techniques using existing data-flow analysis techniques, [33, 35] it would be possible to further optimize communication between separate loop nests by removing communication that can be determined to be redundant when taking into account the program's control flow

Since communication is also currently generated as send/receive pairs that are placed at the same point in a program, this has the potential for causing more synchronization than is necessary. By using data-flow analysis techniques, it should also be possible to move send operations to the earliest point in the program at which the data is read forming a window over which the communication operation can be performed. The earlier communication is started in this window, the higher the message buffer requirements become when taking into account overlapping windows from other communication operations. This trade-off between latency hiding and the memory requirements for the operating system to buffer incoming messages must be taken into account when applying such an optimization.

7.2.2 Optimizing data distribution

The data partitioning algorithms can also be further extended by defining a common interface (similar in spirit to the `libsum` pass in Parafrase-2 [119]) for defining input and output distributions and performance estimation profiles for library functions. Further investigation into the application of statement reordering and loop distribution during the selection of the cut in the phase decomposition step could also be performed to reduce the amount of required redistribution. To handle more general control flow during the phase selection process, the applicability

of existing static and profile-driven trace selection techniques should also be investigated. To be able to perform whole program partitioning analysis, future work will also address interprocedural partitioning analysis using representations such as the Hierarchical Component Affinity Graph.

As we have concentrated on applications with regular access patterns that are amenable to static analysis, future work will also address data partitioning for programs that also contain irregular references (i.e., subscripted subscripts in which the access patterns are not determined until run time) [120]. Although much of the static analysis will no longer apply, the data partitioner may still be able to recommend alignments between irregularly accessed arrays which are indexed by the same run-time dependent pattern or to optimize the distribution for regular accesses while minimizing the worst-case effect for irregular references.

In this thesis we have only considered optimizing data distributions using only the data parallelism present in an application. Other recent work in the PARADIGM project that utilizes both task and data parallelism simultaneously [121] currently assumes that the task graphs [122] and data distributions for the individual tasks are selected separately. Using the techniques presented in this thesis, the data distributions for the individual tasks could be selected in isolation, but would ignore the intertask communication. Ideally, the data distributions for the individual tasks should be selected while taking into account any distribution preferences due to the task dependencies.

The current static cost estimation framework requires all program variables to be known constants at compile time [8]. As this is not always possible for most real programs which can have variable sized data sets that may be constant but are not determined until run time, it would be desirable to be able to manipulate compile-time unknowns symbolically within the cost expressions. Since most applications that use cost estimates only require comparisons to be made between two different costs, symbolic comparison could be performed in situations where numerical costs are not available. Another option would be to integrate the performance estimation framework with feedback from a parallel performance profiling system, similar to the joint work currently being performed between the Fortran D and Pablo projects [123]. By using a hybrid approach of static estimation and profiling techniques, it would also be possible to take

into account the effects due to the memory hierarchy to further improve the accuracy of the estimates.

7.2.3 Optimizing data redistribution

Further work will focus on the application of code transformations (such as loop peeling), which will help reduce the number of reaching distributions to uses of variables with multiple active distributions. When it is not possible to completely eliminate such regions, code replication approaches, similar to the selective function cloning techniques used in this thesis, will also be examined to allow an HPF compiler to efficiently compile such code regions. This type of situation is not currently supported in PARADIGM due to the difficulty in handling such situations.

Currently, redistribution communication is synthesized in a demand-driven fashion (just in time, or lazy, redistribution). Although not examined in this thesis, it would also be possible to take a lazy redistribution solution and determine the earliest possible time that the redistribution could be performed forming a window over which redistribution could occur. It would be advantageous to position multiple redistribution operations in overlapping windows to the same point in the program in order to aggregate the communication thereby reducing the amount of communication overhead. It would also be possible to consider split phase redistribution operations in order to hide communication latency by sending all data in an eager fashion and receiving it in a lazy fashion. The trade-offs of such an approach are similar to those found when hiding latency by separating communication send and receive pairs.

7.2.4 Distributed-shared memory architectures

Finally, as all of the research in this thesis has focused on purely distributed-memory architectures, future research could also address the application of these optimizations in the scope of distributed-shared memory architectures. In these architectures, memory is still physically distributed among the processors, but additional hardware in the memory system is used to provide a global name space (thereby providing a simpler mechanism to access remote memory

in addition to also being able to support message-passing programming models). As this type of architecture will still have differing costs between local and remote memory accesses, it is still beneficial to optimize interprocessor communication (now in the form of data prefetching and forwarding, as previously discussed in Section 2.1) as well as optimize the distribution and redistribution of program data. These techniques now reduce the amount of remote memory references generated by underlying hardware mechanisms as opposed to individual messages.

REFERENCES

[1] Office of Science and Technology Policy, *Grand Challenges 1993: High Performance Computing and Communications*. Washington, D.C.: National Science Foundation, 1992.

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

[3] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM compiler for distributed-memory multicomputers," *IEEE Computer*, vol. 28, no. 10, pp. 37–47, Oct. 1995.

[4] Message-Passing Interface Forum, "MPI: A message-passing interface standard," University of Tennessee, Knoxville, TN, Tech. Rep. CS-94-230, 1994.

[5] High Performance Fortran Forum, "High Performance Fortran language specification, version 1.1," Center for Research on Parallel Computation, Rice University, Houston, TX, Tech. Rep., Nov. 1994.

[6] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors," in *Proceedings of the 18th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1989, pp. II:39–48.

[7] E. W. Hodges IV, "High Performance Fortran support for the PARADIGM compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, Oct. 1995, CRHC-95-23/UILU-ENG-95-2237.

[8] M. Gupta, "Automatic data partitioning on distributed memory multicomputers," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Sept. 1992, CRHC-92-19/UILU-ENG-92-2237.

[9] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," *The Journal of Supercomputing*, vol. 2, no. 2, pp. 151–169, Oct. 1988.

[10] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD distributed memory machines," *Communications of the ACM*, vol. 35, no. 8, pp. 66–80, Aug. 1992.

[11] E. Su, D. J. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A data structure for compiling for distributed-memory multicomputers," in *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, Montréal, Canada, Aug. 1994, pp. 123–132.

[12] 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 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995, pp. 424–433.

[13] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, "Compiler optimizations for distributed memory multicomputers used in the PARADIGM compiler," in *Proceedings of the 23rd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994, pp. II:1–10.

[14] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM 3.0 User's Guide and Reference Manual*, Oak Ridge National Laboratory, Oak Ridge, TN, Feb. 1993.

[15] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "PICL: A portable instrumented communication library, C reference manual," Oak Ridge National Laboratory, Oak Ridge, TN, Tech. Rep. ORNL/TM-11130, July 1990.

[16] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *Journal of Parallel and Distributed Computing*, vol. 13, no. 2, pp. 213–221, Oct. 1991.

[17] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, "Automatic program transformatoins for virtual memory computers," in *Proceedings of the 1979 National Computer Conference*, June 1979, pp. 969–974.

[18] W. Jalby and U. Meier, "Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system," in *Proceedings of the 15th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1986, pp. 429–432.

[19] K. Gallivan, W. Jalby, and D. Gannon, "On the problem of optimizing data transfers for complex memory systems," in *Proceedings of the Second ACM International Conference on Supercomputing*, Saint Malo, France, 1988, pp. 238–253.

[20] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, Apr. 1991, pp. 63–74.

[21] M. J. Wolfe, "More iteration space tiling," in *Proceedings of Supercomputing '89*, Reno, NV, Nov. 1989.

[22] M. E. Wolf, "Improving locality and parallelism in nested loops," Ph.D. dissertation, Computer Systems Laboratory, Stanford University, Stanford, CA, Aug. 1992, CSL-TR-92-538.

[23] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, Apr. 1991, pp. 40–52.

[24] T. C. Mowry, M. S. 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*, Boston, MA, Oct. 1992, pp. 62–73.

[25] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington D.C., July 1992, pp. 104–113.

[26] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors," in *Proceedings of the 16th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1987, pp. 28–31.

[27] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proceedings of the Fourth ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, May 1990, pp. 354–368.

[28] D. K. Poulsen, "Memory latency reduction via data prefetching and data forwarding in shared memory multiprocessors," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, Sept. 1994, CSRD-1377.

[29] A. Rogers and K. Pingali, "Process decomposition through locality of reference," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 69–80.

[30] M. Gerndt, "Updating distributed variables in local computations," *Concurrency: Practice and Experience*, vol. 2, no. 3, pp. 171–193, Sept. 1990.

[31] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "An interactive environment for data partitioning and distribution," in *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC, Apr. 1990, pp. II:1160–1170.

[32] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines," in *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington D.C., July 1992, pp. 1–14.

[33] C. Gong, R. Gupta, and R. Melhem, "Compilation techniques for optimizing communication on distributed-memory systems," in *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1993, pp. II:39–46.

[34] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993, pp. 126–138.

[35] M. Gupta, E. Schonberg, and H. Srinivasan, "A unified data-flow framework for optimizing communication," in *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithica, NY, 1994, vol. 892 of *Lecture Notes in Computer Science*, pp. 263–282, Springer-Verlag, 1995.

[36] A. Rogers and K. Pingali, "Compiling for distributed memory architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 281–298, Feb. 1994.

[37] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Feb. 1971, UICS 71-424.

[38] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM*, vol. 14, no. 3, pp. 563–590, July 1967.

[39] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, pp. 83–93, Feb. 1974.

[40] R. H. Kuhn, "Optimization and interconnection complexity for: Parallel processors, single-stage networks, and decision trees," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Feb. 1980, UICS 80-1009.

[41] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Oct. 1982, CSRD-329.

[42] D. I. Moldovan, "On the design of algorithms of VLSI systolic arrays," *Proceedings of the IEEE*, vol. 71, no. 1, pp. 113–120, 1983.

[43] D. I. Moldovan, *Parallel Processing: From Applications to Systems*. San Mateo, CA: Morgan Kaufman Publishers, 1993.

[44] U. Banerjee, "Unimodular transformations of double loops," in *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, Aug. 1990, pp. 192–219, The MIT Press, 1991.

[45] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*. Boston, MA: Kluwer Academic Publishers, 1993.

[46] H. T. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[47] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, no. 1, pp. 1–12, 1986.

[48] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *Proceedings of the 15th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1986, pp. 836–814.

[49] D. Padua and M. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986.

[50] P. S. Tseng, "A parallelizing compiler for distributed-memory parallel computers," Ph.D. dissertation, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 1989, CMU-CS-89-148.

[51] P. S. Tseng, "Compiling programs for a linear systolic array," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 311–321.

[52] H. Zima, H. Bast, and M. Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, vol. 6, pp. 1–18, 1988.

[53] M. Mace, *Memory Storage Patterns in Parallel Processing*. Boston, MA: Kluwer Academic Publishers, 1987.

[54] K. Knobe, J. Lukas, and G. Steele, Jr., "Data optimization: Allocation of arrays to reduce communication on SIMD machines," *Journal of Parallel and Distributed Computing*, vol. 8, no. 2, pp. 102–118, Feb. 1990.

[55] J. Ramanujam and P. Sadayappan, "Compile-time techniques for data distribution in distributed memory machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 472–481, Oct. 1991.

[56] S. Wholey, "Automatic data mapping for distributed-memory parallel computers," in *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington D.C., July 1992, pp. 25–34.

[57] M. Gupta and P. Banerjee, "PARADIGM: A compiler for automated data partitioning on multicomputers," in *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[58] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng, "Automatic array alignment in data-parallel programs," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993, pp. 16–28.

[59] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993, pp. 112–125.

[60] D. Bau, I. Koduklula, V. Kotlyar, K. Pingali, and P. Stodghill, "Solving alignment using elementary linear algebra," in *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithica, NY, 1994, vol. 892 of *Lecture Notes in Computer Science*, pp. 46–60, Springer-Verlag, 1995.

[61] D. E. Hudak and S. G. Abraham, "Compiler techniques for data partitioning of sequentially iterated parallel loops," in *Proceedings of the Fourth ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990, pp. 187–200.

[62] D. E. Hudak and S. G. Abraham, *Compiling Parallel Loops for High Performance Computers – Partitioning, Data Assignment and Remapping*. Boston, MA: Kluwer Academic Publishers, 1993.

[63] B. Chapman, T. Fahringer, and H. Zima, "Automatic support for data distribution on distributed memory multiprocessor systems," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993, vol. 768 of *Lecture Notes in Computer Science*, pp. 184–199, Springer-Verlag, 1994.

[64] T. Fahringer, "Automatic performance prediction for parallel programs on massively parallel computers," Ph.D. dissertation, University of Vienna, Austria, Sept. 1993, TR93-3.

[65] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, vol. 1, no. 1, pp. 31–50, Aug. 1992.

[66] R. Bixby, K. Kennedy, and U. Kremer, "Automatic data layout using 0-1 integer programming," in *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, Montréal, Canada, Aug. 1994, pp. 111–122.

[67] U. Kremer, "Automatic data layout for High Performance Fortran," Ph.D. dissertation, Rice University, Houston, TX, Oct. 1995, CRPC-TR95559-S.

[68] J. Garcia, E. Ayguadé, and J. Labarta, "A novel approach towards automatic data distribution," in *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction*, Houston, TX, Apr. 1995.

[69] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle, "Automatic data layout for distributed-memory machines in the d programming environment," in *Automatic Parallelization – New Approaches to Code Generation, Data Distribution, and Performance Prediction*, 1993, pp. 136–152.

[70] T. J. Sheffler, R. Schreiber, J. R. Gilbert, and W. Pugh, "Efficient distribution analysis via graph contraction," in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995, vol. 1033 of *Lecture Notes in Computer Science*, pp. 377–391, Springer-Verlag, 1996.

[71] T. J. Sheffler, J. R. Gilbert, R. Schreiber, and S. Chatterjee, "Aligning parallel arrays to reduce communication," in *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995, pp. 324–331.

[72] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler, "Array distribution in data-parallel programs," in *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithica, NY, 1994, vol. 892 of *Lecture Notes in Computer Science*, pp. 76–91, Springer-Verlag, 1995.

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

[74] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, Palo Alto, CA, June 1986, pp. 162–175.

[75] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe, "Interprocedural analysis for parallelization," in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995, vol. 1033 of *Lecture Notes in Computer Science*, pp. 61–80, Springer-Verlag, 1996.

[76] P. Havlak and K. Kennedy, "Experience with interprocedural analysis of array side effects," in *Proceedings of Supercomputing '90*, New York, NY, Nov. 1990, pp. 952–961.

[77] R. Triolet, F. Irigion, and P. Feautrier, "Direct parallelization of call statements," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, vol. 21, no. 7, pp. 176–185, July 1986.

[78] L. Choi and P.-C. Yew, "Interprocedural array data-flow analysis for cache coherence," in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995, vol. 1033 of *Lecture Notes in Computer Science*, pp. 81–95, Springer-Verlag, 1996.

[79] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng, "Interprocedural compilation of Fortran D for MIMD distributed-memory machines," in *Proceedings of Supercomputing '92*, Minneapolis, MN, Nov. 1992, pp. 522–534.

[80] C. W. Tseng, "An optimizing Fortran D compiler for MIMD distributed-memory machines," Ph.D. dissertation, Rice University, Houston, TX, Jan. 1993, COMP TR93-199.

[81] F. Coelho and C. Ancourt, "Optimal compilation of HPF remappings (extended abstract)," Centre de Recherche en Informatique, École des mines de Paris, Fontainebleau, France, Tech. Rep. CRI A-277, Nov. 1995.

[82] E. T. Kalns and L. M. Ni, "Processor mapping techniques toward efficient data redistribution," in *Proceedings of the 8th International Parallel Processing Symposium*, Cancun, Mexico, Apr. 1994, pp. 469–476.

[83] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan, "An approach to communication-efficient data redistribution," in *Proceedings of the 8th ACM International Conference on Supercomputing*, Manchester, U.K., July 1994, pp. 364–373.

[84] S. Ramaswamy and P. Banerjee, "Automatic generation of efficient array redistribution routines for distributed memory multicomputers," in *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995, pp. 342–349.

[85] R. Thakur, A. Choudhary, and G. Fox, "Redistribution of arrays in HPF," in *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, pp. 309–318.

[86] A. Wakatani and M. Wolfe, "A new approach to array redistribution: Strip mining redistribution," in *PARLE '94*, July 1994.

[87] A. Wakatani and M. Wolfe, "Optimization of array redistribution for distributed memory multicomputers," *Parallel Computing*, 1995.

[88] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th International Symposium on Computer Architecture*, Queensland, Australia, May 1992, pp. 256–266.

[89] E. Su, D. J. Palermo, and P. Banerjee, "Automating parallelization of regular computations for distributed memory multicomputers in the PARADIGM compiler," in *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1993, pp. II:30–38.

[90] T. Fahringer, R. Blasko, and H. P. Zima, "Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems," in *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington D.C., July 1992, pp. 347–356.

[91] V. Sarkar, "Determining average program execution times and their variance," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 298–312.

[92] J. Li and M. Chen, "Index domain alignment: Minimizing cost of cross-referencing between distributed arrays," in *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, Oct. 1990, pp. 424–433.

[93] M. Gupta and P. Banerjee, "Compile-time estimation of communication costs on multicomputers," in *Proceedings of the Sixth International Parallel Processing Symposium*, Beverly Hills, CA, Mar. 1992, pp. 470–475.

[94] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA: Benjamin/Cummings Publishing, 1995.

[95] G. Golub and J. M. Ortega, *Scientific Computing: An Introduction with Parallel Computing*. San Diego, CA: Academic Press, 1993.

[96] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.

[97] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[98] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 229–248, Jan. 1993.

[99] E. Ayguadé, J. Garcia, M. Girones, M. L. Grande, and J. Labarta, "Data redistribution in an automatic data distribution tool," in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995, vol. 1033 of *Lecture Notes in Computer Science*, pp. 407–421, Springer-Verlag, 1996.

[100] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[101] D. J. Palermo, E. W. Hodges IV, and P. Banerjee, "Dynamic data partitioning for distributed-memory multicomputers," *Journal of Parallel and Distributed Computing*, 1996, to appear in a special issue on *Compilation Techniques for Distributed Memory Systems*.

[102] D. J. Palermo, E. W. Hodges IV, and P. Banerjee, "Interprocedural array redistribution data-flow analysis," in *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug. 1996, to appear.

[103] A. Holub, *Compiler Design in C*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[104] R. von Hanxleden and K. Kennedy, "GIVE-N-TAKE – A balanced code placement framework," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994, pp. 107–120.

[105] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[106] F. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-53745, Dec. 1986.

[107] Perfect Club, "The Perfect Club benchmarks: Effective performance evaluation of supercomputers," *The International Journal of Supercomputing Applications*, vol. 3, no. 3, pp. 5–40, Fall 1989.

[108] D. J. Palermo, E. Su, E. W. Hodges IV, and P. Banerjee, "Compiler support for privatization on distributed-memory machines," in *Proceedings of the 25th International Conference on Parallel Processing*, Bloomingdale, IL, Aug. 1996, to appear.

[109] M. T. Heath and J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, vol. 8, no. 5, pp. 29–39, Sept. 1991.

[110] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS parallel benchmarks," NASA Ames Research Center, Moffett Field, CA, Tech. Rep. RNR-91-002, 1991.

[111] R. Sadourny, "The dynamics of finite-difference models of the shallow-water equations," *Journal of the Atmospheric Sciences*, vol. 32, no. 4, Apr. 1975.

[112] K. Kennedy and U. Kremer, "Automatic data layout for High Performance Fortran," in *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[113] Applied Parallel Research, Inc., Placerville, CA, *XHPF User's Guide, version 2.0*, 1995.

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

[115] D. B. Loveman, "The DEC High Performance Fortran 90 compiler front end," in *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995, pp. 46–53.

[116] The Portland Group, Inc., Wilsonville, OR, *PGHPF User's Guide*, 1995.

[117] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim, "An overview of a compiler for scalable parallel machines," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993, vol. 768 of *Lecture Notes in Computer Science*, pp. 253–272, Springer-Verlag, 1994.

[118] Z. Bozkus, "Compiling Fortran 90D/HPF for distributed memory MIMD computers," Ph.D. dissertation, Syracuse University, Syracuse, NY, June 1995, SCCS-730.

[119] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2 manual," Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Tech. Rep., Aug. 1990.

[120] A. Lain, "Compiler and run-time support for irregular computations," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Oct. 1995, CRHC-92-22/UILU-ENG-95-2236.

[121] S. Ramaswamy, "Simultaneous exploitation of task and data parallelism in regular scientific applications," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, Jan. 1996, CRHC-96-03/UILU-ENG-96-2203.

[122] M. Girkar and C. Polychronopoulos, "Partitioning programs for parallel execution," in *Proceedings of the Second ACM International Conference on Supercomputing*, Saint Malo, France, 1988, pp. 216–229.

[123] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. A. Reed, "An integrated compilation and performance analysis environment for data parallel programs," in *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

VITA

Daniel Joseph Palermo received the B.S. degree in Computer and Electrical Engineering from Purdue University, West Lafayette, Indiana, in May of 1990 and the M.S. degree in Computer Engineering from the University of Southern California, Los Angeles, California, in December of 1991. He will receive the Ph.D. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1996.

During the summer of 1988, Palermo worked as a technical aide in the area of distributed information systems at The Mitre Corporation in Bedford, MA. During the summer of 1989, he joined Texas Instruments in Lewisville, TX, to work on an expert systems project for prebriefing operational flight software, and during the summers of 1990 and 1991 he worked as a software engineer at TI in the area of image processing. From January 1992 to the present, he has worked as a research assistant in the Center for Reliable and High-Performance Computing at the University of Illinois.

After completing his doctoral dissertation, Palermo will join the Hewlett-Packard Convex Technology Center in Dallas, TX. His research interests include high-performance computing systems, parallelizing and optimizing compilers, and parallel languages and application programming.