RICE UNIVERSITY

Advanced Data-Parallel Compilation

by

Daniel Gerardo Chavarría-Miranda

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

John Mellor-Crummey
Senior Faculty Fellow of Computer Science and Electrical & Computer Engineering

Kenneth W. Kennedy
Ann and John Doerr University Professor of Computational Engineering

George J. Hirasaki
A. J. Hartsook Professor of Chemical Engineering

Vijay S. Pai
Assistant Professor of Electrical & Computer Engineering

Houston, Texas
December, 2003
Advanced Data-Parallel Compilation

Daniel Gerardo Chavarría-Miranda

Abstract

Over the past few decades, scientific research has grown to rely increasingly on simulation and other computational techniques. This strategy has been named in silico research. Computation is increasingly important for testing theories and obtaining results in fields where experimentation is not currently possible (e.g., astrophysics, cosmology, climate modeling) or the detail and resolution of the results cannot be provided by traditional experimental methodologies (e.g., computational biology, materials science). The quantities of data manipulated and produced by such scientific programs are often too large to be processed on a uniprocessor computer. Parallel computers have been used to obtain such results.

Creating software for parallel machines has been a very difficult task. Most parallel applications have been written using low-level communication libraries based on message passing, which require application programmers to deal with all aspects of programming the machine. Compilers for data-parallel languages have been proposed as an alternative. High-Performance Fortran (HPF) was designed to simplify the construction of data-parallel programs operating on dense distributed arrays. HPF compilers have not implemented the necessary transformations and mapping strategies to translate complex high-level data-parallel codes into low-level high-performance applications.

This thesis demonstrates that it is possible to generate scalable high-performance code for a range of complex, regular applications written in high-level data-parallel languages, through the design and implementation of several analysis, compilation
and runtime techniques in the dHPF compiler. The major contributions of this thesis are:

- Analysis, code generation and data distribution strategies (multipartitioning) for tightly-coupled codes.
- Compiler and runtime support based on generalized multipartitioning.
- Communication scheduling to hide latency, through the overlap of embarrassingly parallel loops.
- Advanced static analysis for communication coalescing and simplification.
- Support for efficient single-image executables running on a parameterized number of processors.
- Strategies for generation of large-scale scalable executables up to hundreds of processors.

Experiments with the NAS SP, BT and LU application benchmarks show that these techniques enable the dHPF compiler to generate code that scales efficiently up to hundreds of processors with only a few percent overhead with respect to high-quality hand-coded implementations. The techniques are necessary but not sufficient to produce efficient code for the NAS MG multigrid benchmark, which still exhibits large overhead compared to its hand-coded counterpart.
Acknowledgments

I wish to convey my gratitude and admiration towards my advisor John Mellor-Crummey. His invaluable support and professional guidance made this research possible. I also wish to thank the other members of my thesis committee: Dr. Ken Kennedy, Dr. Vijay Pai and Dr. George Hirasaki, for their interest and help.

A mi familia por su cariño y por enseñarme el valor de la responsabilidad y el trabajo.

To my friends for their support and friendship when it was most needed.
# Contents

Abstract ........................................ ii  
Acknowledgments ................................ iv 
List of Illustrations ................................ x  
List of Tables ................................... xiii

1 Introduction ............................................ 1  
1.1 Data Parallel Languages ................................. 2  
1.1.1 High Performance Fortran .............................. 4  
1.2 Thesis ............................................. 6  
1.3 Research Contributions ................................ 6

2 Related Work ........................................... 8  
2.1 Data Parallel Compilation ............................... 8  
2.1.1 Fortran D ........................................ 8  
2.1.2 PCRC .......................................... 9  
2.1.3 SUIF ......................................... 10  
2.1.4 Polaris ........................................ 12  
2.1.5 Other Projects ................................ 14  
2.2 HPF Compilation ...................................... 15  
2.2.1 VFC - Vienna Fortran Compiler .................... 18  
2.2.2 Previous work on dHPF .............................. 19  
2.3 Shortcomings of Existing HPF Compilers ............. 20  
2.4 Data Partitioning ................................... 21  
2.5 Communication Analysis and Generation ............. 22
2.6 Memory Hierarchy Optimizations for Parallel Programs ....... 23

3 The dHPF Compiler 
  3.1 dHPF Compiler Phases ........................................ 26
     3.1.1 Traditional Analyses .................................... 26
     3.1.2 Data-Parallel Analyses ................................ 27
     3.1.3 SPMD Code Generation .................................. 31
     3.1.4 Communication Code Generation ......................... 33
     3.1.5 Symbolic Number of Processors .......................... 33

4 Data-Parallel Optimizations for High Performance .......... 36
  4.1 Simplifying Formulations for Compact Sets .................. 36
  4.2 Coalescing ...................................................... 37
     4.2.1 Normalization ............................................. 38
     4.2.2 Coalescing Normalized Communication Sets .............. 40
  4.3 Processor Set Constraints ..................................... 43
     4.3.1 Necessity of Constraints .................................. 43
  4.4 HPF/IA-inspired Extensions .................................... 45
  4.5 Partially Replicated Computation ............................ 47
     4.5.1 Interprocedural Communication Elimination ............. 48
  4.6 General Communication Aggregation ........................... 49
  4.7 Load-balanced BLOCK-style partitionings ..................... 49
     4.7.1 Implementation in dHPF ................................. 51

5 Line Sweeps and Multipartitioning ............................... 56
  5.1 Line sweeps ..................................................... 56
  5.2 Data Partitionings for Line Sweep Computations ............. 57
     5.2.1 Multipartitioning ........................................ 60
     5.2.2 Generalized Multipartitioning ........................... 64
5.3 Implementation of Multipartitioning in dHPF .................. 67
  5.3.1 HPF Directives & Multipartitioning ....................... 67
  5.3.2 Syntactic Extensions ..................................... 67
  5.3.3 Virtual Processors ........................................ 70
  5.3.4 Code Generation .......................................... 70
  5.3.5 Runtime Support ......................................... 72
5.4 Tile Scheduling & Tile Loop Placement ......................... 72
  5.4.1 Tile Scheduling for Generalized Multipartitioning ........ 73
5.5 Unidirectional Communication & Tile Enumeration ............. 76
5.6 Multipartitioning Optimizations in dHPF ....................... 79
  5.6.1 Aggressive Communication Placement ..................... 80
  5.6.2 Formal Properties of Multipartitioned Communication ..... 82
  5.6.3 Aggregating Communication Across Tiles .................. 86
  5.6.4 General Communication Scheduling ....................... 89
  5.6.5 Carried Communication Scheduling ....................... 90
  5.6.6 Deadlock and Communication Scheduling .................. 93
  5.6.7 Tile-based Layout for Multipartitioned Arrays ........... 94

6 Optimizing Scalar Code of SPMD Programs ..................... 97
  6.1 Integer Set-based Code Generation ......................... 97
  6.2 Generating Optimizable Code ................................ 99
  6.3 Overlap Regions vs. Direct-Access Buffers .................. 100
    6.3.1 Aggregation and Direct Buffer Access .................. 104
  6.4 Padding for Dynamically-allocated Arrays .................. 104
  6.5 Memory Allocation for Communication Buffers ............... 105

7 Experimental Validation ..................................... 109
  7.1 NAS SP and BT Application Benchmarks ...................... 109
  7.2 NAS LU Application Benchmark ............................. 110
7.3 NAS MG Kernel Benchmark ........................................... 111
7.4 SPEC95 tomcatv benchmark ............................................. 116
7.5 SGI Experimental Framework .......................................... 116
  7.5.1 Performance Comparison ........................................... 117
  7.5.2 Data Partitioning Support ......................................... 122
  7.5.3 Partial Computation Replication .................................. 123
  7.5.4 Interprocedural Communication Elimination ....................... 124
  7.5.5 Normalized Coalescing ............................................. 125
  7.5.6 Multi-variable Aggregation ......................................... 126
  7.5.7 Processor Set Constraints ......................................... 127
  7.5.8 Direct Access Buffers .............................................. 128
  7.5.9 Tile-based Layout for Multipartitioned Arrays .................... 129
  7.5.10 Padding for Dynamically Allocated Arrays ....................... 130
  7.5.11 Generalized Multipartitioning .................................. 131
7.6 HP Experimental Framework ............................................ 134
  7.6.1 NAS SP .............................................................. 134
  7.6.2 NAS BT .............................................................. 138
  7.6.3 NAS LU .............................................................. 141
  7.6.4 NAS MG .............................................................. 145
  7.6.5 SPEC95 tomcatv ...................................................... 150

8 Conclusions .............................................................. 153
  8.1 Future Work ............................................................. 155
    8.1.1 Short & Medium Term Issues ..................................... 155
    8.1.2 Longer Term Issues .............................................. 157
  8.2 Final Remarks .......................................................... 157

A Definitions of Omega set operations .................................. 158
B Dependence Definitions

Bibliography
Illustrations

3.1 Processor Grid .................................................. 35

4.1 Simple overlapping non-local data references ...................... 38
4.2 Complex overlapping non-local data references .................... 38
4.3 Coalescing non-local data across loops .......................... 39
4.4 Unconstrained Right Partners .................................. 43
4.5 Constrained Right Partners .................................... 44
4.6 Extended ON HOME vs. EXT_HOME ............................. 46
4.7 Extended REFLECT directive .................................. 47
4.8 Partially Replicated Computation along a processor boundary ...... 48
4.9 2D unbalanced BLOCK partitioning .............................. 50
4.10 Fixed-size communication partners ................................ 54
4.11 Different-sized communication partners ......................... 55

5.1 Line sweeps with transpose .................................... 58
5.2 Serialization induced by line sweep ................................ 59
5.3 Pipelined computation of a line sweep ............................ 60
5.4 2D Multipartitioning on 5 processors ............................ 61
5.5 3D Multipartitioning on 16 processors ............................ 63
5.6 Generalized multipartitioning for 6 processors .................... 67
5.7 Using dHPF's layout directive extensions for multipartitionings ...... 68
5.8 Tile Enumeration Order for a 2D multipartitioning ................ 71
5.9 Algorithm for selecting data dimension and direction for tile enumeration. ........................................... 74
5.10 Tile scheduling for generalized multipartitioning. .................. 75
5.11 General loop nest for a one-dimensional sweep .................... 76
5.12 Tile coordinates in the global domain. ................................. 77
5.13 Communication Aggregation for multipartitioned tiles ............ 87
5.14 Communication event components and their interleaving to aggregate communication across tiles. ....................... 89
5.15 General structure of a tiling loop ..................................... 91
5.16 Multipartitioned Arrays ............................................... 94
5.17 Contiguous tiles for arrays ............................................ 95
5.18 Contiguous corresponding tiles ........................................ 96

6.1 An overlap region on an array in HPF. ................................. 100
6.2 SPMD code using an overlap region. ................................... 101
6.3 Overlap Regions ....................................................... 101
6.4 SPMD code using a direct-access buffer. .............................. 102
6.5 Optimized use of a direct-access buffer. .............................. 103
6.6 Array Padding ....................................................... 105
6.7 Communication Buffer Reuse ......................................... 108

7.1 Data partitioning for MG .............................................. 114
7.2 Single template alignment for multiple grid levels ................... 115
7.3 Multiple template alignment for MG grid levels ....................... 115
7.4 Parallel efficiency for NAS SP (class 'B'). .......................... 118
7.5 Parallel efficiency for NAS SP (class 'A') ............................ 119
7.6 Parallel efficiency for NAS BT (class 'B') ............................ 119
7.7 Parallel efficiency for NAS BT (class 'A') ............................ 120
7.8  Parallel efficiency for NAS SP (HP, class 'A').  
7.9  Parallel efficiency for NAS SP (HP, class 'B').  
7.10 Parallel efficiency for NAS SP (HP, class 'C').  
7.11 Parallel efficiency for NAS BT (HP, class 'A').  
7.12 Parallel efficiency for NAS BT (HP, class 'B').  
7.13 Parallel efficiency for NAS BT (HP, class 'C').  
7.14 Parallel efficiency for NAS LU (HP, class 'A').  
7.15 Parallel efficiency for NAS LU (HP, class 'B').  
7.16 Parallel efficiency for NAS LU (HP, class 'C').  
7.17 Parallel efficiency for NAS LU (small scale SGI, class 'A').  
7.18 Parallel efficiency for NAS MG (HP, class 'A').  
7.19 Parallel efficiency for NAS MG (HP, class 'B').  
7.20 Parallel efficiency for NAS MG (small scale SGI, class 'A').  
7.21 Parallel efficiency for the SPEC95 tomcatv benchmark (HP, $514^2$  
(small) size).  
7.22 Parallel efficiency for the SPEC95 tomcatv benchmark (HP, $1028^2$  
(large) size).
### Tables

7.1 2D-block CGP vs. multipartitioning. ........................................ 122
7.2 1D-block transpose (PGI) vs. multipartitioning. .......................... 123
7.3 Impact of partial computation replication ................................. 124
7.4 Impact of using the HPF/JA directives to eliminate communication
    interprocedurally. ....................................................... 125
7.5 Impact of normalized coalescing. ......................................... 125
7.6 Impact of aggregation. .................................................... 127
7.7 Impact of processor set constraints. ...................................... 128
7.8 Impact of direct-access buffers on SP and BT. ............................ 129
7.9 Impact of tile-based for layout for SP and BT. ............................ 129
7.10 Impact of array padding on SP and BT. ................................. 130
7.11 Comparison of hand-coded and dHPF speedups for NAS SP (class B). 132
7.12 dHPF optimizations applied to NAS SP ................................. 136
7.13 dHPF optimizations applied to NAS LU ................................. 143
7.14 dHPF optimizations applied to NAS MG ................................. 145
7.15 dHPF optimizations applied to SPEC95 tomcatv ....................... 152
Chapter 1

Introduction

Over the past few decades, scientific research has grown to rely increasingly on simulation and other computational techniques. This strategy has been named *in silico* research. Computation has become an extremely important research tool for testing theories and obtaining results in fields where experimentation is not currently possible (e.g. astrophysics, cosmology, climate modeling) or the detail and resolution of the results cannot be provided by traditional experimental methodologies (e.g. computational biology, materials science).

The quantities of data manipulated and produced by such scientific programs are often too large to be processed on a uniprocessor computer. In addition, the growing appetite of scientists for better and more detailed results has, in many cases, outstripped the improvements in single processor computer performance, which track the 60% annual increase predicted by “Moore’s law”.

Parallel computers have been used to provide the necessary power to obtain such results. Currently, the most common parallel architecture is a cluster of shared-memory multiprocessors (SMPs) connected by a high-bandwidth, low-latency interconnect. Large supercomputers now include thousands of processors and can deliver raw computational power measured in teraflops ($10^{12}$ floating-point operations per second).

Harnessing the power of machines of such scale to deliver scientific results is the task of parallel software. Creating software for large-scale parallel machines has been a very difficult task. Most applications developed for them have been written using low-level message-passing libraries, such as MPI [SOHL95], which require applica-
tion programmers to deal with *all* aspects of programming such a machine: designing the parallel algorithms, partitioning the computation, distributing the data, coordinating process synchronization, providing access to remote data, managing computation/communication overlap, implementing data allocation and storage policies, etc. All of these details have to be managed when writing a high-performance scientific application to take advantage of the power of parallel computers.

This low-level message-passing approach has the advantage that an experienced application programmer can fully control how the application executes in parallel. This low-level approach also leverages the support of sequential language compilers, since most message passing libraries have interfaces for C and Fortran. Compilers for these languages are usually the most mature on a particular platform and produce high-quality machine code for numerical applications. The combination of good-quality sequential machine code and a good implementation of a message passing interface, together with a large effort by an experienced application programmer can produce good parallel applications, at the cost of many programmer-hours of debugging and performance testing and tuning.

A goal of research on parallelizing compilers and parallel languages has been to reduce the amount of effort needed to write parallel applications. Many of these languages and compilers have been either ineffective or have required that the application programmer switch to a new programming paradigm, with different programming language semantics and compiler support that might not be as mature as that provided by traditional sequential languages.

1.1 Data Parallel Languages

An alternative approach is that of data parallel languages. Experimental application studies have determined that many numerical algorithms exhibit a large degree of parallelism, repeating the same operations over a large collection of data elements. This observation has led to the design of several data parallel languages. In this
paradigm, applications can be designed as a collection of processes which operate on different portions of the data domain and communicate data among them as necessary. Each process can be viewed as a traditional sequential program with the addition of communication and synchronization primitives, leveraging the strengths of sequential language compilers and low-level communication libraries. It is also possible for a parallelizing compiler to accomplish the task of partitioning the data for such a program.

High-level data-parallel languages offer an attractive model for writing parallel applications. They enable the application programmers to concentrate on solving the problem at hand and insulate programmers from low-level parallelization details, such as synchronization and data movement between processes. A significant obstacle to the acceptance of data-parallel languages has been that compilers for such languages do not routinely generate code that delivers performance comparable to that of carefully-tuned, hand-coded parallel implementations. This has been especially true for tightly-coupled applications, which require frequent communication and synchronization in computational loops. In addition, the lack of several important optimizations in compilers for data-parallel languages have made it impossible for programmers using them to approach the performance of hand-coded parallel implementations with any amount of source-level rewriting. Since high performance is the principal motivation for developers of parallel implementations, scientists have preferred to hand-craft parallel applications using lower level programming models, such as MPI. Applications using these low-level models can be tuned to deliver the necessary level of performance, albeit with a larger programming effort.

An obstacle to attracting application programmers to data parallel languages has been the difficulty of effectively tuning the performance of applications by transforming the source code, due to the semantic gap between the high-level language model and the actual code that a data-parallel compiler must generate. They must rely on a compiler to perform the necessary transformations and adequately map the high-
level semantics of the source code into low-level code for a particular architecture. This low-level code must efficiently use both the processors and the communication interconnect to achieve high performance on modern parallel computer architectures.

1.1.1 High Performance Fortran

High Performance Fortran (HPF) [Hig93] was designed to simplify the construction of data-parallel programs operating on dense distributed arrays. The main idea behind the language is that by adding a few data distribution directives to a sequential Fortran program written in an implicit data-parallel form, the compiler would then construct an explicitly parallel SPMD (Single Program Multiple Data) program by reducing loop and array bounds and synthesizing the necessary communication and synchronization between processors.

To synthesize efficient communication, an HPF compiler must be able to statically analyze the source program. Regular programs are defined to be those that are statically analyzable by a compiler. Their main characteristic is that subscripts used in array accesses are affine functions of loop induction variables, loop invariant variables or constants. A program that has other types of array accesses is said to be irregular. The first version of the design of the HPF language implicitly targeted regular codes.

There have been several proposals on extensions to HPF and other data-parallel languages to support irregular applications [Han94, BMRZ98, UZCZ97, MR95]. Several authors have described analysis and optimization techniques for compiling irregular data-parallel applications [AS95, DHSS01]. The combination of the language extensions and the proposed compilation techniques have provided the necessary compiler support for irregular applications in HPF-like languages. Their results show that the performance of the compiler-generated irregular codes can closely approach that of hand-coded irregular applications. On the other hand, HPF compilers have not been able to deliver high-performance code for complex regular applica-
tions [CDS00, AJMCY98, FHJ+98]. The resulting codes are usually an order of magnitude slower than hand-coded applications.*

Regular applications fall into two classes:

1. Loosely synchronous applications are those in which all loops are fully parallel and communication and synchronization are only needed between loop nests;

2. And tightly-coupled applications require communication and synchronization inside loop nests.

Loosely synchronous applications, for which the compiler only requires local knowledge to generate code and communication statements, are relatively simpler to compile efficiently. HPF compilers have been able to routinely generate sufficiently good quality code for them.

HPF compilers have not implemented the necessary transformations and mapping strategies to translate complex (especially tightly-coupled) high-level data-parallel codes into low-level high-performance applications. Generated code for complex HPF programs performs poorly due to several factors:

- Poor local program performance on each processor node.
- Generation of inefficient communication (poor schedules, unneeded data, redundant data transfers).
- Dependence on runtime resolution for communication and data partitioning, even for statically analyzable code.

Tightly-coupled applications require more sophisticated techniques to compile them into high-performance code. One important focus of this thesis is investigating techniques to generate scalable high-performance code for this class of applications.

*Compilers based on the HPF/JA extensions to the HPF language [SIOT98, ISK00] have been an exception: the HPF/JA-based NEC HPF compiler generated code for a plasma simulation that achieved 45% of peak performance on 512 nodes of the Earth Simulator [SMYS02]
1.2 Thesis

My thesis is that it is possible to generate scalable high-performance code for a range of complex, regular applications written in high-level data-parallel languages. To support my thesis, I have designed and implemented a set of compiler techniques in the Rice dHPF compilation system. My work has focused on designing, implementing and evaluating optimizations to remove the performance gap between low-level hand-coded parallel applications and those produced by a high-level data-parallel compiler.

1.3 Research Contributions

The main contribution of this thesis is a set of techniques to enable the generation of high-performance scalable parallel code from a high level data-parallel language. This thesis describes techniques to increase the performance of compiler generated code for regular HPF applications. These techniques can be classified into three different groups: optimizations to reduce communication volume; optimizations to improve communication schedules; and techniques to increase local node performance, primarily by increasing cache efficiency. Some of the techniques improve both communication and memory hierarchy efficiency. These schemes have been implemented in the Rice dHPF compiler in order to evaluate their effectiveness in the compilation of data-parallel codes.

In particular, these techniques include:

- Analysis and code generation strategies for tightly-coupled codes
- Support for advanced data distribution schemes (multipartitioning) for tightly-coupled codes
- Compiler and run-time support based on generalized multipartitioning
- Effective communication vectorization for tightly-coupled codes
• Communication scheduling to hide latency, through the use of non-blocking primitives

• Communication and computation overlap by fusing loops containing embarrassingly parallel computation with those containing loop-carried communication

• Advanced static analysis for communication coalescing and simplification

• Efficient communication buffer management for multi-procedure SPMD programs

• Support for efficient single-image executables running on a parameterized number of processors (for both BLOCK and multipartitionings)

• Strategies for generation of large-scale scalable executables up to hundreds of processors
Chapter 2

Related Work

This chapter summarizes related work on data parallel and HPF compilation, data and computation partitioning, communication analysis and generation, and memory hierarchy optimizations for parallel programs. It also provides comparisons with our work on dHPF (where appropriate).

2.1 Data Parallel Compilation

2.1.1 Fortran D

Tseng [Tse93] in his doctoral dissertation presents a compiler for the Fortran D language. Fortran D is an extension to both Fortran 77 and Fortran 90, which adds data decomposition specifications in two levels: array alignment and data distribution. Fortran D is an immediate predecessor to the HPF language and the Fortran D compiler is itself and immediate predecessor to our dHPF compiler. The Fortran D compiler uses mathematical functions to represent the set of data elements owned by a particular processor. These functions are used to determine computation partitions, through the “owner computes” rule, and to detect non-local array accesses statically. Tseng clearly identifies the need for dependence analysis and dependence-based transformations to produce good quality parallel code. He introduces and describes several communication optimizations which target the need to reduce communication frequency and volume on distributed memory parallel machines. His work also includes interprocedural analysis to compute reaching decompositions for subroutine parameters, as well as to detect the needed extents for overlap regions.

His identification of the functions that map array elements to processors and
iterations to processors (according to the “owner computes” rule) is a very important precursor to the formalizations of these functions as set mappings in the context of the Omega library in dHPF [AMC98].

Tseng assumes that the data distribution options available in Fortran D are enough to support all kinds of data parallel applications, even for tightly-coupled codes. For this kind of codes, he proposes using a BLOCK partitioning together with a form of coarse-grain pipelining. Further studies have shown that these distributions are not sufficient, specially for tightly-coupled codes [AJMCY98].

2.1.2 PCRC

Bozkus [Boz95], in his doctoral dissertation, presents and discusses the implementation of a Fortran 90D compiler. He extends the Fortran 90 language with the data distribution directives from Rice’s Fortran D project. His analysis and compilation techniques support optimization and code generation for distributed memory machines. His work is related to that of Tseng, in the sense that both of them use a version of Fortran as a base language and extend it with data distribution and alignment directives. Tseng uses more powerful dependence-based and loop-based analysis and transformation scheme, while Bozkus uses a statement-by-statement translation approach with little provision for inter-statement optimization. He also relies on parallel language constructs (such as Fortran 90 array statements and the forall statement) provided by the user to detect parallelism. Since his work targets distributed memory compilation, the Fortran 90D compiler must generate communication statements to support non-local accesses to distributed arrays. The compiler relies on pattern matching to recognize special cases of communication (shifts, transfers of specific values, broadcasts) or else defaults to an element-by-element runtime scheme. Bozkus’s work proposes several optimizations to eliminate communication overhead by reducing communication frequency and volume, however it is not clear how these optimizations are effective in the context of a statement-by-statement anal-
ysis, instead of a loop-nest level or global analysis. His work does support the de-
tection and generation of special communication patterns for irregular array accesses
and some optimizations on them (schedule reuse).

2.1.3 SUIF

Anderson and Lam [AL93b] present algorithms for decomposing the computation and
the data of a program on a scalable parallel machine. They identify this decomposition
as one of the most important issues in compiling for parallel architectures. This work
was done in the context of the SUIF compiler project to which they added support
for the automatic parallelization of Fortran 77 programs.

The algorithms they present are intended to automatically derive a good data and
computation decomposition that maximizes parallelism and locality and minimizes
communication between processors. Their analyses and transformations are based on
generating affine decompositions for both computation and data *. Their algorithm
finds a partition of the arrays and iterations, an orientation of the array elements
and iterations, and a displacement for them. Affine decompositions are found that
minimize communication and maximize parallelism for multiple loop nests. They also
support dynamically changing and pipelined decompositions.

Their approach is more flexible and simpler for the programmer than the ap-
proach used in directive-based data parallel languages, but the compiler analysis is
significantly more complex. It requires sophisticated symbolic manipulation. Their
algorithms are also limited in selecting computation decompositions for loops, be-
cause they can only select a single decomposition for a loop nest. Their approach will
compensate by selecting complex data decompositions with changes in orientation
and displacement.

*Regular applications are affine if the loop bounds and array accesses are affine functions of the
loop indices and symbolic constants.
Amarasinghe and Lam [AL93a] introduce several algorithms to solve code generation and optimization issues associated with distributed memory parallel machines. Their work is also implemented in the SUIF compiler. They can generate a SPMD node program from a sequential Fortran 77 program with affine loop bounds and array accesses.

Instead of dependence analysis they use a data-flow technique based on Last Write Trees. They claim that they have a precision advantage over traditional data dependence analysis, but it is not clear why would dependence analysis be less precise on affine, regular codes.

They have extended their affine analysis and code generation framework to handle distributed memory compilation. Data and computation are decomposed across processors and communication is generated using affine sets. They discuss several code generation issues, including how to enumerate all the integer elements in a constrained affine set, which is the basic component for generating communication code. They use similar affine set enumeration techniques for the partitioned computation portion of the code.

Their compiler is also able to apply several optimizations to reduce the costs of the communication code. They eliminate redundancy in communication sets and aggregate messages that have the same source and destination. In their terminology, aggregation includes message vectorization. They also support data replication to reduce communication.

Code generated for a distributed memory machine with a complicated data distribution can be very costly in terms of evaluating the expressions necessary to determine ownership of array elements. Their approach will suffer from this problem if they need to use sophisticated data partitioning because their computation partitioning choices are limited to a single decomposition per loop nest. They also depend on very precise data decomposition analysis for obtaining efficient code.

Lim [Lim01, LL98] extends the previous work done on affine transforms for paral-
elization to enhance data locality. She also proposes generalizations of the previous algorithms to apply them to imperfect loop nests and more general program structures.

2.1.4 Polaris

Blume et al. [BEF+95] present a description of their Polaris automatic parallelizing compiler. At the time (1995), it was oriented towards shared memory compilation only, but it was intended to perform fully automatic parallelization from sequential Fortran 77 code. They discuss some software engineering and implementation issues as well as the important transformations and optimizations in their compiler. Polaris supports automatic inlining of procedures, reduction recognition, auxiliary induction variable substitution, they also provide a new symbolic dependence test based on loop ranges. The compiler also performs automatic scalar and array privatization. Their work includes a runtime dependence testing technique which allows for speculation on loops that are not known to be statically parallel.

Their work differs from the dHPF project in important ways. The most important one being our focus on distributed memory compilation and locality transformations, which involve a whole new set of analysis and code generation techniques.

Pack et al. [PNZ+02] present their work on a compiler for non-cache coherent (NCC) distributed shared memory machines with a non-uniform memory access (NUMA) architecture. They claim that programming such machines is as difficult as programming fully distributed machines with a message passing library. For this reason, they propose using a parallelizing compiler based on the Polaris infrastructure developed at the University of Illinois at Urbana-Champaign (UIUC).

Polaris is a fully automatic parallelizing compiler, that is, it supports the parallelization of programs written in Fortran 77 without any extra support from the programmer. Their extensions for compiling for NCC machines are divided into three major compilation steps: detecting parallelism, distributing iterations and data, and
generating NCC code. This approach is similar to the one taken by SUIF, although Polaris’s analysis is less powerful.

Their parallelism detection stage includes interprocedural analysis for determining dependence and array regions accessed by each loop. Iteration and data distribution are somewhat limited because they only exploit unidimensional loop parallelism. Locality for the arrays is analyzed for each phase of the program and between phases, to determine an optimal iteration and data distribution scheme. Finding the optimal scheme, involves formulating an integer programming problem, which minimizes communication and load imbalance, while maximizing locality and parallelism. The last phase involves NCC code generation.

A particular feature of NCC machines is that shared data is not cached, which means that shared objects need to be copied to private, cacheable memory for computational loops to run efficiently. This involves a particular case of communication which they name *copy-in/copy-out*. This is pure overhead which doesn’t occur on other architectures (fully distributed or cache-coherent NUMAs).

Their approach is powerful enough to handle automatic parallelization of Fortran 77 for NCCs, but it is limited by the fact that their compiler can only exploit unidimensional loop parallelism. On the other hand, having a shared memory substrate available, enables them to employ sophisticated data distributions that might compensate their limited selection of computation distributions. Their experimental results show the effectiveness of their techniques and also the effectiveness of the NCC architecture at executing codes with enhanced locality.

Navarro, Zapata and Padua [NZP03] extended these techniques to handle cache-coherent NUMA machines, as well. They improved their techniques to achieve better speedups and efficiencies on the Cray T3E and the Origin 2000.
2.1.5 Other Projects

Lee and Kedem [LK02] present several techniques to distribute data and computation on distributed memory machines. Their techniques are similar to the approach used in the SUIF project: both of them use automatic parallelism detection and selection of appropriate data and computation decompositions. They provide algorithms to compute the decompositions automatically using heuristics. Their approach includes selecting a “principal” array to be decomposed, other arrays will be aligned to this one. Limited support for computation partitionings other than the owner-computes rule, is provided in the form of the owner-stores rule. They present an algorithm for tiling data and computation spaces, but it is limited to two-dimensional problems. Their experimental validation is somewhat limited because it was conducted on antique hardware platforms, that do not have the same performance tradeoffs as more modern machines.

Wu et al. [WDS+95] discuss the design of the ARF compiler for sparse problems on distributed memory machines. Their techniques are based on a compiler for a simplified version of the Fortran D language, which is translated into a parallel Fortran program with calls to the PARTI runtime library for irregular problems. They identify, for irregular applications, many of the same issues that other researchers have identified for regular applications. Namely, data and computation partitioning, element aggregation, redundancy elimination for communication, etc. Due to the nature of the applications they target, their techniques are mostly runtime based and limited in the classes of code their approach can handle. For example, they cannot handle loop carried dependencies and indirect accesses on distributed arrays. An important contribution of this work is identifying the cost involved in executing inspectors for irregular communication.

Creusillet and Irigoin [CI95] support very precise interprocedural array region data flow analysis in the PIPS compiler. Their main focus is to perform interprocedural array privatization. Array reshaping across procedures is supported. Their work is
complementary to the work on the dHPF compiler: dHPF currently does not perform interprocedural array section analysis. dHPF does support programmer-controlled array privatization.

Govett et al. [GHH+03] introduce a new directive-based tool for parallelizing codes based on regular, dense grids. They support code generation for both shared and distributed memory computers, using different communication libraries. Their approach is very similar to that of the HPF/JA extensions to the HPF base language. The Scalable Modeling System (SMS) directives that they propose have a more “executable”, lower-level flavor than the traditional high-level HPF directives. This approach has been shown to produce good performance by enabling the programmer to tune performance through the careful application of these low-level directives.

2.2 HPF Compilation

Zhang et al. [ZCF+97] discuss a strategy for implementing a HPF compiler using partial static code generation and relying on some runtime transformations. They are also able to generate code that performs better than that of commercial HPF compilers, for small benchmarks. Their approach is an extension of the original work on Fortran 90D, done by Bozakus at Syracuse.

Coelho, Germain and Pazar [CGP96] provide an overview of techniques proposed and used to compile HPF. This work illustrates very clearly the difficulties found in compiling diverse HPF applications efficiently, for a broad class of parallel machines. They also identify clearly the tradeoff between usability and efficiency found in current (1996) HPF compilers. They state that the application programmer must understand the workings of the compiler to produce efficient applications, in contrast to the language designers’ goals.

They identify the difficulties in analyzing and implementing all the possible variations of data distributions and alignments efficiently. Some of the combinations might involve non-static knowledge that the compiler cannot determine (symbolic
number of processors: \texttt{numberOfProcessors()}, and others involve inherent ambiguity (\texttt{inherit} directive). They also identify the difficulties of implementing Fortran 77's legacy features and HPF's distributed memory model in a simple way.

Their work also discusses the necessary support to compile cyclic distributions and the extensions for irregular applications which introduce yet another set of difficulties.

The significance of this paper is that it identifies several difficulties in compiling HPF efficiently, specially supporting the more esoteric alignment and distribution options. This is something that was not identified in the original design of the HPF language.

Ancourt, Coelho, Irigoin and Keryell [ACIK93] describe a linear algebra framework for compiling HPF. They define formal linear expressions based on transformation matrices and vectors for HPF objects: arrays, templates, alignments, distributions, processors, etc. More importantly, they also express the relationships between these objects in algebraic terms. They define the sets of iterations that have to be computed by each local processor, using the owner computes rule; they also define the set of array elements accessed by the local iteration set and thus determine the necessary non-local data elements that form the send and receive communication sets.

They also extend their framework to handle symbolic linear algebra expressions that appear when some of the runtime parameters, such as the number of processors, are not known. Their work includes a specification on how local partitioned arrays should be allocated, this is very important in cases where arrays have complex alignment and distribution relationships, because their local expression, if naively allocated, may have "holes".

This work is very related to the SUIF affine partitioning model, although here the authors start with a data decomposition explicitly given by the programmer. They identify more clearly some of the problems that have to be solved to handle general, complex decompositions.

Gupta et al. [GMS+95] discuss the design and implementation of the pHF com-
piler done at IBM Research. They implement several optimizations available in the dHPF compiler, but in a more limited form. pHPF can handle symbolic processor numbers and by extension, symbolic block sizes. Their compiler also does message vectorization and message coalescing but in a more restricted form than what we have implemented in dHPF. They also only support the owner-computes rule for partitioning computation.

Gupta, Kaushik, Huang and Sadayappan [GKHS96] use a virtual processor approach in combination with closed forms for communication sets, to support efficient communication for block-cyclic distributions in data-parallel languages. This approach is similar to how we handle communication sets and computation partitionings for multipartitioned data distributions (see section 5.3.3). They only test their strategy in the context of a loop nest with a single assignment using the owner-computes rule. It is not clear if their approach can handle a whole application with diverse data and computation partitionings.

Hwang, Lee and Ju [HLJ01] present a set of techniques designed for scalarizing Fortran 90/HPF non-local array expressions for distributed memory machines. Their work is orthogonal and complementary to our work on dHPF.

van Reeuwijk et al. [vDSP96] discuss the design and implementation of a framework to support HPF distributed arrays on distributed memory machines. The focus of their work is the support of general block-cyclic distributed arrays. They present methods for efficient local index enumeration, local storage allocation and communication set enumeration. They emphasize supporting arbitrarily aligned arrays with general block-cyclic distributions, which potentially lead to allocated data with “holes” in it, and they show schemes to avoid such holes and to enhance locality by reducing storage requirements for local portions of distributed arrays. These techniques are applicable to dHPF, but we have not focused on supporting general block-cyclic distributions yet, only dense block-like ones.

Iwashita, Sueyasu and Kamiya [ISK00] and Seo et. al. [SIOT98] discuss the exten-
sions proposed in Japan to the HPF language (HPF/JA). These extensions focus on improving the quality of the generated code through the use of lower level directives, than those in the core HPF language. These directives specify explicit communication for distributed arrays, as well as locality properties that the compiler can use to improve the generated code.

Compilers based on the HPF/JA directives have been able to generate code that performs at 45% of peak performance on 4096 processors of the Earth Simulator [SMSY02]. They have achieved this very high performance through the careful use of the HPF/JA directives to optimize shift communications for scalability. The plasma simulation they analyze is loosely synchronous, which requires less sophisticated data-parallel techniques to achieve high performance. They achieve a significant fraction of peak performance through the effective use of each node's vector unit.

2.2.1 VFC - Vienna Fortran Compiler

Benkner [Ben99] describes the work done at the University of Vienna on their Vienna Fortran Compiler (VFC). Their project focuses on compiling a superset of HPF, with extensions targeted towards supporting irregular applications. Their compilation model for irregular applications is based on the inspector-executor model. They support multi-level inspectors and executors and have language support and optimizations to reuse schedules computed in past inspections. The results that they report show that their extensions and compilation techniques are quite effective in achieving good performance for irregular applications. The achieved performance is competitive with that of hand-coded irregular applications using inspectors and executors.

Fahringer and Mehofer [FM99] discuss a series of techniques for optimizing communication in data-parallel HPF programs, in the context of the Vienna Fortran Compiler. They consider different criteria for placing, aggregating and coalescing communication: latency hiding, reducing the number and volume of messages, as
well as the size of the buffers required to hold the communicated values. They also use a sophisticated performance estimator to guide the compile-time decisions in their tool. They provide experimental evidence of the importance of their approach by evaluating different communication placements for programs with varying data sizes and varying numbers of processors involved.

Fahringer and Scholz [FS00] describe their work on incorporating highly sophisticated symbolic analysis in the Vienna Fortran Compiler. They model every major Fortran program construct to obtain precise static information on the values of variables, symbolic expressions constraints between variables and conditional flow expressions. Their representations uses a construct called a program context which uses first-order logic formulas to represent control and data flow across a program. This sophisticated information is then used to analyze and transform programs for parallelization under VFC. In dHPF, the compiler uses symbolic analysis for different phases of the compiler: communication set analysis, construction and representation; computation partitioning and range propagation for SPMD code. dHPF’s symbolic analyses are not as cohesive as this approach in VFC, but they provide considerable flexibility and power for the respective transformations [AMC98, AJMCY98, MCA97].

2.2.2 Previous work on dHPF

Adve and Mellor-Crummey [AMC00, AMC98] describe a general integer set framework and techniques to implement code generation for HPF. This was done as part of the dHPF compiler project at Rice. They describe several analysis and code generation algorithms implemented as part of this set-based framework, and provide qualitative evidence of the generality of the approach compared to case and pattern-based approaches.
2.3 Shortcomings of Existing HPF Compilers

Frumkin et al. [FHJ+98] studied the performance of code generated by several parallelizing compilers and tools as well as their ease of use. They compared the parallelizations produced by the interactive CAPTools toolkit, from the University of Greenwich; the Portland Group’s pgHPF compiler [BMN+97]; the MIPSPro Fortran compiler from SGI, with directives for shared memory parallelization; and the same MIPSPro compiler with a hand-coded parallelization using the MPI library. The evaluation included parallelizing a subset of the NAS benchmarks (LU, SP, BT, FT, MG, and CG) and NASA’s ARCh3 application. The comparison was performed on a SGI Origin 2000 for all versions.

Their results show that the performance of the HPF versions of the benchmarks is among the worst, and that parallelization of the codes involved not only adding HPF directives but also substantial code restructuring to match the compiler’s paradigm.

Chamberlain, Deitz and Snyder [CDS00] compared the performance of the NAS MG benchmark implemented using several parallel languages: Co-Array Fortran (CAF), HPF, Single Assignment C (SAC), and ZPL (from the University of Washington). They measured the resulting performance on multiple parallel platforms: Cray T3E, IBM SP, SGI Origin, Sun Enterprise 5500 and a Linux cluster.

The resulting performance of the HPF versions is the worst of all, even though the HPF source code requires fewer lines than other versions.

Adve, Jin, Mellor-Crummey and Yi [AJM CY98] show that the performance of vendor-tuned HPF implementations of the NAS parallel benchmarks BT and SP, is relatively poor when compared to the NASA hand-coded implementations. The performance of a HPF version, based on the sequential code, compiled with Rice’s dHPF compiler is superior to that of the vendor-tuned version.

Hu et al. [HJY+00, JH99] describe their HPF Bench benchmark package carefully designed for HPF programs. Their results show that commercial compilers do not generate code with good performance even on HPF-specific code. Their analysis
points to poor local node performance.

Yau, Fox and Hawick [YFH97] also defined a set of benchmarks for HPF compilers. They tested their programs on more commercial platforms and compilers than the previous authors, and they found many of the same problems: poor scalability due to very high local node overheads and lack of flexibility in diagnosing the problems and solving them with source code changes.

Nucciaroni, Ozyoruk and Long [NOL97] ported their aeroacoustic application from CM Fortran [TMC91] to High-Performance Fortran. They compiled their application port using PGI’s pgfhpf compiler and IBM’s xlfhpf compiler, on an SGI Origin 2000 and an IBM SP2 respectively. The performance of the pgfhpf compiled code on the SGI Origin 2000 was acceptable and even showed some speedup, but on the SP2 the code exhibited slowdowns as more processors were added.

2.4 Data Partitioning

Shindo et al. [SIK+94] implemented a data partitioning strategy equivalent to two-dimensional multipartitioning in a HPF compiler for the Fujitsu AP1000 machine. Their approach allows the use of multipartitioning with either BLOCK or CYCLIC data partitionings in HPF. They achieve this by separating the skewing effect of assigning one tile per hyperplane from the data partitioning itself. It is not clear whether their method is applicable to higher dimensional arrays. Their results indicate that this approach is more scalable than regular 1D or 2D distributions.

Pande and Bali [PB99] propose a technique for partitioning iterations and data in the context of DOALL loops, that is fully parallel loops with no carried dependences. Their approach will perform data and iteration partitioning on a DOALL loop nest by analyzing directions (in data and iteration space) in which locality can be increased. After this phase, the granularity of the partitions must be adjusted to guarantee load balance. Then the partitions must be mapped onto the available physical processors. The data space is also partitioned using the “larger partition owns” rule which they
introduce.

Their approach works fairly effectively in the context of single loop nests, which is what they target, but adapting it to a multi-loop context or an interprocedural context, poses multiple complex problems, specially with respect to data restructuring. This prevents this kind of methods from being effective for larger, more complex applications.

Ramanujan and Sadayappan [RS91] present an approach which partitions computation and data for distributed memory machines. Their approach is based on creating deadlock-free iteration partitionings (tiles) for perfect multidimensional loop nests. They use an approach based on finding extreme vectors that generate the dependence vectors of the analyzed loop nest, these extreme vectors, in turn, induce a partitioning of the iteration space. Choosing the extreme vectors carefully can minimize the communication requirements for the loop nest. Their approach which creates atomic tiles that can be executed to completion without synchronizing is similar to the tiles in a multipartitioned distribution, whose computation can be completed without synchronizing. Their approach is limited to single, perfectly nested loops.

2.5 Communication Analysis and Generation

Chakrabarti, Gupta and Choi [CGC96] present a global algorithm for communication analysis and placement in the context of IBM’s pHPF research compiler. They focus on procedure-level communication placement and also on minimizing the volume of transferred data and its redundancy. Their placement techniques are based on the procedure’s control flow graph.

Kandemir et al. [KBC+99] present the techniques implemented inside their HPF compiler to globally optimize communication. They use a data-flow approach combined with linear algebra techniques to accurately reduce communication volume and frequency across whole HPF programs. Their framework supports message vectorization, message coalescing and redundancy elimination across multiple loop nests.
This approach is somewhat limited because they require compile-time knowledge of the exact number processors under which the program will be executed.

In comparison to dHPF's techniques, their approach has the advantage of global communication scheduling and movement, while dHPF is loop nest-based. dHPF supports symbolic communication set analysis and code generation for statically unknown processor counts. More importantly, their framework does not support non-owner-computes computation partitionings, which previous work on dHPF [AJMCY98] has shown to be of critical importance to reduce communication requirements. Also, the benchmarks selected for their experimental analysis are somewhat small and do not necessarily benefit from a sophisticated approach.

Stichnoth, O'Hallaron and Gross [SOG94] present techniques to support communication for array statements on a distributed memory parallel machine. They describe a method to compute the communication sets associated with an array statement and provide support for all standard HPF distributions: block, cyclic and block-cyclic. To handle such a variety of distributions, they represent communication sets using unions of array sections. The innovative aspect of their investigation is the idea of shipping the target array addresses from the sender to the receiver to avoid having the overhead of computing them in both places.

dHPF's approach for generating communication sets is more general than that of unions of array sections. The tradeoffs involved in computing the receiver's target addresses have changed since the time the author's published their work. For example, computation is much cheaper compared to communication than in 1994. Still, we have not experimented with cyclic-based distributions which require more costly local address generation.

2.6 Memory Hierarchy Optimizations for Parallel Programs

Kandemir, Ramamjum and Choudhary [KRC00] propose a series of techniques used to optimize the locality, parallelism and communication requirements of parallel pro-
grams, in an unified framework. The main difference of their work with respect to similar projects is that they perform global data space transformations on arrays, to rearrange their memory layouts. In this way, temporal and spatial locality can be achieved in cases where it is impossible to achieve them with iteration space transformations only. They also support traditional iteration space transformations and communication optimizations such as message vectorization. The main limitations on their approach is that they can only handle arrays distributed in one dimension, and that the data layout transformations can only handle reordering the dimensions of an array, not arbitrary affine data space layouts (which might be needed in some cases). This is a very different approach from the dHPF project, in which we have focused on code generation and optimizations to transform global HPF source code into very efficient distributed SPMD code.

Ehold et. al. [EGKU02] propose using the HPF EXTRINSIC mechanism to incorporate high-performance single-processor kernels into HPF parallel programs. The mechanism they use allows for the integration of sequential routines into a parallel program, in such a way that the sequential routines can be called to compute on locally allocated data on each processor. The authors demonstrate how this can lead to higher performance and better efficiency than compiling the same routines through the standard HPF compilation process because there is no need to deal with global parallel program issues within these kernels.
Chapter 3

The dHPF Compiler

The dHPF compiler has been in development at Rice University for the past 8 years. It builds upon compiler research conducted at Rice over the last 25 years. The dHPF project has focused on design, implementation and evaluation of analysis and code generation techniques for High-Performance Fortran.

dHPF supports whole-program compilation of HPF programs written in a Fortran 77 style. It also includes support for key Fortran 90 constructs such as array sections, interface blocks and dynamically allocated arrays. It supports the HPF data layout directives, along with extensions inspired by HPF/JA, the Japanese HPF project.

As output of its compilation process, dHPF produces an SPMD program expressed as Fortran source code. The generated program contains calls to a runtime library that supports interprocess coordination and communication. Currently, the only supported runtime coordination and communication library is MPI (Message Passing Interface), although an earlier version of the compiler supported shared-memory style coordination as well. Support for shared memory coordination was dropped after a major restructuring of the communication code generation subsystem because of the effort it would require to update it.

This chapter provides an overview of the dHPF compiler and its main characteristics: dHPF serves as the infrastructure upon which the techniques described in this thesis were built.
3.1 dHPF Compiler Phases

Each phase in the dHPF compiler operates on one procedure at a time. Procedures are compiled in a bottom-up order to support interprocedural propagation of requirements for processor involvement from callee to caller.

3.1.1 Traditional Analyses

The dHPF compiler performs a number of traditional analysis phases in preparation for data-parallel analyses and code generation. Most of these phases are adaptations of their classical formulations to the abstract syntax tree level at which dHPF operates; this contrasts with traditional compilers that generate machine code, which typically operate on a lower-level intermediate representation, such as three-address code.

The principal traditional dHPF compiler phases include:

- Parsing and construction of source-level Abstract Syntax Tree (AST) [ASU86]
- Construction of a source-level Control Flow Graph (CFG) [ASU86]
- Computation of an auxiliary source-level Static Single Assignment (SSA) form [CFR89] of the procedure, taking into account subscripted array expressions
- Computation of source-level value numbers [RWZ88] for all expressions, including subscripted array operations and array indices
- Construction of a full data and control dependence graph [Ban88]

All of these phases operate on the original HPF source program. Ignoring, for now, any implications of HPF data distribution and alignment directives that are present.
3.1.2 Data-Parallel Analyses

After the initial analyses on the global form of the source program, dHPF proceeds to use the computed information as the basis for transforming the code into a parallel form. The HPF data layout directives describe how to partition the program data. The main objective of this phase is to build the necessary information to determine where to place communication and synchronization events in the resulting SPMD program, to provide access to off-processor data. This phase also determines the effects of data partitioning on the computational structure of loops and other Fortran constructs (e.g., reducing loop bounds operating on local portions of the data).

First of all, dHPF has to determine which data distribution and alignment directives reach each statement operating on arrays. This step helps the compiler determine whether a particular statement will need communication. The following data-parallel analysis step is to determine where communication should be placed in the resulting SPMD program. This is performed by using SSA and dependence information computed on the global program.

Initial Communication Placement

Logical communication reads should be placed before SSA uses of distributed array expressions and logical communication writes should be placed after SSA defs of distributed array elements. The data dependences incident on a reference determine the loop level at which its communication should be placed, communication for a reference is not needed at any loop level deeper than the innermost loop carrying a true or output data dependence incident on the reference. This is a well-known transformation known as communication vectorization [GD95]. Of course, communication is only needed if the particular dimension of the array has been partitioned. Scalar variables may cause communication, but they are assumed to be replicated across processors and are frequently privatizable. More details can be found in [AMCO00].
Computation Partitioning

The initial dependence-based communication placement performed by dHPF considers all possible communication that might be needed without considering the data partitioning. This placement is later refined by considering data partitionings and by selecting computation partitionings (CPs) for each statement in the procedure.

Data partitionings for arrays in HPF are explicitly written by the programmer using HPF directives such as \texttt{DISTRIBUTE}, \texttt{ALIGN} and \texttt{TEMPLATE}. The principal HPF directives used to specify data distributions are the following:

- \texttt{PROCESSORS}: Specifies an $n$-dimensional processor array onto which data will be distributed. It is a logical grouping of processors that does not necessarily match the physical topology of the parallel computer.

- \texttt{TEMPLATE}: Specifies an abstract space of indexed positions. In essence, it is a "virtual array" to which other real data arrays can be aligned.

- \texttt{DISTRIBUTE}: Specifies a distribution of the elements of a \texttt{TEMPLATE} onto a processor array. Each dimension can be distributed independently using a \texttt{BLOCK} distribution, in which contiguous elements of that dimension are assigned to each processor, or using a \texttt{CYCLIC}(k), in which contiguous groups of $k$ elements are assigned to each processor in a round-robin fashion.

- \texttt{ALIGN}: Specifies that a data element of an array mapped to an element of a template should be colocated on the same processors as elements of other arrays mapped to the same template element. The relation between elements is specified using an index expression on a per-dimension basis.

Given such data layout directives the HPF compiler computes a data distribution and determines which processors should execute each instance of a computational statement of the program.
To determine computation partitionings, it is enough to consider affine assignment statements, because computation partitionings for other types of statements are principally derived from the partitioning of the assignment statements. An affine assignment in Fortran can be defined in general terms as follows:

\[ A_f(f(i_1, \ldots, i_n)) = g(A_{r1}(f_{r1}(i_1, \ldots, i_n)), \ldots, A_{rn}(f_{rn}(i_1, \ldots, i_n))) \] (3.1)

Each array \( A_f \) may be distributed (or not) and has a unique non-negative number of dimensions \((\geq 0)\). The \( i_k \) are induction variables defined by a loop nest enclosing the assignment or loop invariant variables or constants. The functions \( f_\gamma \) are affine vector-valued functions of the \( i_k \) variables and \( g \) is a function of the array expressions \( A_{f}(f_{i_1}, f_{i_2}, \ldots, f_{i_n}) \), where \( n \) is the number of enclosing loops for the statement.

Distributing the computation of instances of such assignments can be done by analyzing which processors own the array elements involved in the expressions on the left hand side and the right hand side of the assignment. The ideal situation would be to execute the assignment on the processor that owns most of the array elements involved in it. This would minimize the communication needed. Selecting this optimal partitioning might be costly [CGST93] and requires detailed information about the relative array sizes of \( A_f \) and the \( A_{r\gamma} \)'s, as well as knowledge about the computation defined by \( g \). In fact, this might not be statically determinable if \( g \) is an opaque function call.

A simple heuristic to partition the computation for an affine assignment statement is to assume that the computation will be performed by the processor that owns the left-hand side (or target) of the assignment \( (A_f) \). This partitioning strategy is known as the “owner-computes rule” [RP94]. After a partitioning is selected, the HPF compiler can determine if array elements indexed by affine expressions on the right hand side of the assignment are located on the same processor as the target of the assignment, as long as the array in the assignment target is also indexed by affine expressions.
Using the owner-computes partitioning strategy, communication must be synthesized by the compiler for array elements which do not reside on the same processor as the left hand side target. This “owner-computes” strategy is clearly not optimal: consider the statement $A(i, j) = B_1(i + 1, j) + B_2(i + 1, j) + \ldots + B_{24}(i + 1, j)$. If the arrays are distributed and aligned equivalently, and if elements $i$ and $i + 1$ do not reside on the same processor, applying the “owner-computes” strategy to this assignment would require sections of 24 different arrays to be communicated. Obviously, in this case, it would be better to perform the computation of the expression on the processors that own the $i + 1$ array elements and then communicate the result to the processor owning array element $i$. The array reference chosen for computation partitioning, is called the HOME reference (from the HPF directive ON HOME).

The dHPF compiler uses a more sophisticated approach than the “owner-computes” strategy [AMC98]. It will consider computing the affine assignment on processors that own any subset of the array expressions involved in the assignment. For the example as described, dHPF would perform the computation on the processor that owns the array elements $i + 1$ (ON HOME $B_1(i + 1, j)$). The flexibility of dHPF’s partitioning strategy enables the compiler to partially or totally replicate computation by computing the target of an assignment for partitioned array elements on multiple processors. Computation replication can reduce communication by making the target available on multiple processors, as long as it does not require communicating more right-hand side elements.

**CP Propagation & Final Communication Placement**

Once computation partitionings have been chosen for elemental statements, they are propagated to other Fortran statements, such as control flow statements (conditionals and loops) as well as to procedure call sites.

Since the initial communication placement is performed without taking into consideration the computation partitionings, it must be refined to include their effects.
This involves choosing CPs for the communication statements themselves: processors that need to send data should execute the send portions of the communication event, similarly for processors that need to receive data. Communication statements are placed in the different scopes of each procedure according to dependence information: they can be at the procedure scope or inside loops or other compound statements.

Once CPs have been assigned to all statements (computation & communication), the compiler uses a heuristic to determine an approximate communication cost for these choices. The compiler then proceeds to test different CP choices for elemental statements and repeats the propagation and CP synthesis process for communication until the communication cost cannot be minimized further (CP choices for elemental statements depend on the distributed array references they contain). For more details see [AJMCY98].

This makes CP propagation and communication an iterative, fixed-point process which stops when the set of computation partitionings for all statements in the procedure does not change.

### 3.1.3 SPMD Code Generation

After the information collected by the previous analysis phases is available, the dHPF compiler generates an SPMD program based on it. SPMD code generation is based on an equational framework using parameterized sets of integer tuples and mappings.

This symbolic framework constructs the following sets and mappings:

- \( \text{data}_k \): The index set of an array of rank \( k, k \geq 0 \)
- \( \text{loop}_k \): The iteration space of a loop nest of depth \( k, k \geq 0 \)
- \( \text{proc}_k \): The processor index space in a processor array of rank \( k, k \geq 0 \)
- \( \text{Layout}: \text{proc}_n \rightarrow \text{data}_k \): \{ \([p] \rightarrow [a]: \text{array element } a \in \text{data}_k \text{ is allocated to processor } p \in \text{proc}_n \} \)
• RefMap: $\text{loop}_k \rightarrow \text{data}_n$: \{ $[i] \rightarrow [a]$: array element $a \in \text{data}_n$ is referenced in iteration $i \in \text{loop}_k$ \}

• CPMa$p: \text{proc}_n \rightarrow \text{loop}_k$: \{ $[p] \rightarrow [i]$: statement instance $i \in \text{loop}_k$ is assigned to processor $p \in \text{proc}_n$ \}

dHPF uses the Omega library [KMP+96] to represent these sets and mappings at compile time. Omega supports the representation and manipulation of symbolic sets of integer tuples and mappings described by Presburger formulae. dHPF relates the abstract integer spaces described by these sets and mappings to the HPF source code using value numbers computed from the source code. For example, the abstract variables representing the range of a loop iteration space are mapped to the value number of the induction variables in the source code. More details on the construction and manipulation of the dHPF symbolic integer sets can be found in [AMC98].

The computation of the mapping CPMa$p$ involves the computation partitionings that were determined in the previous phases of the compiler (e.g. propagated from a called procedure), as well as taking into account the data partitioning specified by the user. Applying the Omega library code generation algorithm to these sets produces code templates that enumerate all points in the symbolic integer iteration spaces defined by CPMa$p$. The statements inside the original global loops can then be placed inside these templates to execute them in the reduced iteration space of the SPMD node program.

One critical aspect in generating efficient code for an SPMD program from these sets is representing the final sets that are going to be used for template generation in a compact and simple form. The Omega library by itself cannot optimize the representation of the sets before code generation due to the lack of information about the context in which the generated code will appear. For example, it is possible to factor out common satisfiability conditions for the computation partitionings for a set of statements and check them only once before the group of statements, instead of verifying them for each statement individually.
In general, it is our experience that Omega can reason more rapidly about simple, convex integer sets and mappings than non-convex representations. For this reason, several aspects of code generation in dHPF that could naturally use non-convex representations (e.g., code generation for imperfect loop nests, communication set generation for multiple arrays and multiple regions of the same distributed array) were instead designed to split sets into simpler, convex components before code generation and generate code for each component separately.

### 3.1.4 Communication Code Generation

Using the basic sets and mappings defined in section 3.1.3, the dHPF compiler can derive sets that identify data elements referenced by statements which do not reside on the “local” processor (according to the computation partitioning), or data produced by the “local” processor which needs to be sent back to an owning processor (again depending on the computation partitioning).

Each communication event placed in the previous analysis phases, operates on the set of data elements that the local processor expects to receive from some other remote processor, or the set of data elements that the local processor must send to other remote processors. dHPF uses the Omega code generator to produce templates that enumerate each point in the receive and send sets for the local processor. Replacing the statement placeholders in the code templates produced by Omega by the appropriate operation, makes it possible for the compiler to generate code that counts the number of elements in a set (for buffer allocation purposes), packs the elements into a buffer to be sent, and then unpacks elements from the receive buffer into the appropriate storage for the remote data.

### 3.1.5 Symbolic Number of Processors

The dHPF compiler can generate code for a symbolic number of processors. To do so, it must analyze and generate communication for data distributions partitioned into
blocks whose sizes are symbolic [AMC98]. There are significant challenges that must be met in order to generate fast and efficient code for this case.

As described in [AMC98], the principal challenge for generating efficient code comes from the presence of a symbolic block size and its implications inside the integer-set based compiler core. A precise mapping of symbolically-sized blocks to processors is not possible in the Presburger arithmetic formalism underlying the Omega library; such a mapping can only be described with products of two symbolic variables: the processor id and the symbolic block size.

For example, distributing a 1D array of extent $N$ over $P$ processors would yield a block size of $B = \lceil N/P \rceil$. Figure 3.1 illustrates a processor grid with these characteristics. The mapping that naturally represents the set of template elements owned by processor $p$ can be expressed as:

$$\{[t] \rightarrow [p] : Bp + 1 \leq t \leq Bp + B \wedge 1 \leq t \leq N \wedge 1 \leq p \leq P\} \quad (3.2)$$

where $t$ represents a template element, $B$ is the block size and $p$ is a particular processor. The $Bp$ product terms are not legal in Presburger arithmetic and thus this formula cannot be represented directly using Omega’s integer tuples.

For this reason, dHPF represents such data distributions less precisely using a virtual processor formulation that avoids symbolic multiplication. The set of elements owned by a virtual processor $v$ can be expressed as:

$$\{[t] \rightarrow [v] : v \leq t \leq v + B - 1 \wedge 1 \leq t \leq N \wedge 1 \leq v \leq N\} \quad (3.3)$$

where $t$ represents a template element, $B$ is the block size and $v$ represents a virtual processor. This formula represents the mapping of data to a grid of virtual processors, with a virtual processor aligned on every template element. However, this formulation fails to precisely represent the relationship between the data tiles owned by real processors. To generate correct code for enumerating processor partners with this representation, dHPF first uses the Omega library to generate loops that enumerate
every possible virtual partner, then the compiler transforms the loops to enumerate only real processor partners. For complex communication sets, this two-step process leads to complex code.
Chapter 4

Data-Parallel Optimizations for High Performance

In the quest to tune dHPF’s generated code to achieve high performance, several new types of optimizations were devised that improve the overall quality of the generated code.

4.1 Simplifying Formulations for Compact Sets

Many of these optimizations are refinements of the analysis and code generation strategies that construct and manipulate sets of integer tuples using the Omega Library [KMP+96]. Omega represents sets of integer tuples with Presburger Formulas [Coo72] as constraints.

When using the Omega library for analyzing sets representing data locations, iterations spaces and processors and generating code templates to enumerate these sets, we found that compilation time and generated-code complexity are directly related to the complexity of the set representation. Omega contains a sophisticated procedure for generating code templates that enumerate the elements of a set [KPR95]. However, if the sets are complicated, non-convex disjunctions then these code templates will be complex and inefficient. The dHPF compiler tries to ensure that sets are represented in their simplest possible form, namely, as convex sets or as unions of simple, convex sets. Appendix A defines the Omega operations dHPF uses to manipulate integer sets.

dHPF’s flexible computation partitioning strategy can give rise to disjunctive sets. For one of the benchmarks studied, communication analysis failed as it tried to manipulate an overly complex set representation. The problem was caused by
selecting partially-replicated non-owner-computes partitionings for loops with unrolled code. Computation partitioning analysis resulted in a set representation that was a disjunction of 25 conjuncts. Inspection of the set showed that its form was
\[ \bigcup_{1 \leq i, j, k \leq 5} \{ l_1, l_2, i, j, k \} : i \leq i_u \land j < j_u \land k < k_u \}, \]
where \( i, j, k, j_u, k_u \) are symbolic constants for the upper and lower bounds of \( i, j, \) and \( k, \) respectively. Intuitively, this can be represented compactly as a single conjunct with the constants in the first two positions of the tuples replaced by the ranges \( 1 \leq l_1, l_2 \leq 5. \) Omega’s set simplification machinery failed to exploit this opportunity for simplification, we discovered that such Omega sets with compact ranges of constant terms could be collapsed by recomputing a set \( S \) as \( \text{Hull}(S) - (\text{Hull}(S) - S) \).
This process forces the set to be recomputed as a difference from a single conjunct, which has the effect of reducing the set to its most compact form.

4.2 Coalescing

For best performance, an HPF compiler should minimize the number and volume of messages. Analyzing and generating messages for each non-local reference to a distributed array in isolation produces too many messages and the same values might be communicated multiple times.

In dHPF, a communication set is initially computed and placed separately for each individual non-local reference. A communication set is represented in terms of an \texttt{ON HOME} reference (corresponding to the computation partitioning where the data is required) and a non-local reference. Both references are represented in terms of value numbers that appear in their subscripts (if any). Whenever possible, dHPF vectorizes communication and hoists it out of loops. When multiple communication events are scheduled at the same location in the code, dHPF tries to coalesce them to avoid communicating the same data multiple times.

Consider the code in Figure 4.1, an HPF compiler should generate a single message to communicate a single copy of the off-processor data required by both references to
CHPF$ distribute a(*, block), b(*, block) onto P
do j = 2, n
  do i = 1, n
    a(i, j) = b(i, j - 1) + c ! ON_HOME a(i, j)
    a(i, j) = a(i, j) + d + b(i, j - 1) ! ON_HOME a(i, j)
  end do
end do

Figure 4.1: Simple overlapping non-local data references.

CHPF$ distribute a(*, block), b(*, block) onto P
do j = 2, n
  do i = 1, n - 1, 2
    a(i, j) = b(i, j - 1) + c ! ON_HOME a(i, j)
    a(i + 1, j) = d + b(i, j) ! ON_HOME a(i + 1, j)
  end do
end do

Figure 4.2: Complex overlapping non-local data references

b(i, j - 1). However, detecting when sets of non-local data for multiple references overlap is not always so simple. References that are *not* syntactically equivalent may require identical or overlapping non-local data. In Figure 4.2, references b(i, j - 1) and b(i, j) require identical non-local values and can be satisfied by a single message. To avoid communicating duplicate values, overlap between sets of non-local data required by different loop nests should be considered as well, as shown in Figure 4.3.

### 4.2.1 Normalization

To avoid communicating duplicate values, we developed a normalization scheme as a basis for determining when communication sets for different references overlap. To
do timestep = 1, T
  Coalesce data exchange at this point
  do j = 1, n
    do i = 3, n
      a(i, j) = a(i + 1, j) + b(i - 1, j) ! ON_HOME a(i, j)
    enddo
  enddo
  do j = 1, n
    do i = 1, n - 2
      a(i + 2, j) = a(i + 3, j) + b(i + 1, j) ! ON_HOME a(i + 2, j)
    enddo
  enddo
  do j = 1, n
    do i = 1, n - 1
      a(i + 1, j) = a(i + 2, j) + b(i + 1, j) ! ON_HOME a(i + 1, j)
    enddo
  enddo
enddo

Figure 4.3: Coalescing non-local data across loops.

compensate for differences in computation partitionings selected for different statements, normalization rewrites the value numbers representing a communication set into a form relative to its ON_HOME reference expressed in a canonical form. In our discussion of normalization, we refer to the ON_HOME reference for a communication set as the computation partitioning (CP) reference and the non-local reference as the data reference.

dHPF’s value-number based representation for communication sets has the disadvantage that non-local references that arise in different loops are incomparable because their subscripts have different value number representations. To enable us to detect when such references may require overlapping sets of non-local data, we convert all data and CP references to use a canonical set of value numbers for the loop induction variables involved.

We apply our normalization algorithm to communication sets represented in terms of value numbers based on affine subscript expressions of the form ai + b, where i is
an induction variable, $a$ is a known integer constant and $b$ is a (possibly symbolic) constant. This restriction comes from the need to compute a symbolic inverse function for such expressions*. If this restriction is not met, the communication set is left in its original form.

A communication set is in normal form if:

- The CP reference is of the form $A(i_1, i_2, i_3, ..., i_n)$
- The data reference is of the form $A(a_1i'_1 + b_1, a_2i'_2 + b_2, a_3i'_3 + b_3, ..., a_ni'_n + b_n)$

where $A$ is an $n$-dimensional array, each $i_k$ is an induction variable or a constant, and each $i'_j$ corresponds to a unique $i_k$. We say that a particular subscript expression in the CP reference is normalized if it is of the form $i_j$, where $i_j$ is an induction variable or a constant.

If a communication set is not in normal form but meets our restriction of affine subscript expressions, we normalize it by computing symbolic inverse functions for each non-normalized CP subscript position. We then apply this function to each subscript position in both the CP and data references. After this step, only the data reference has subscripts of the form $a_ji_j + b_j$. Each subscript position gets a canonical value number represented by an artificial induction variable that has the range and stride of the original $a_ji_j + b_j$ subscript expression. We apply this normalization step to all communication events at a particular location in the code before attempting to coalesce them.

### 4.2.2 Coalescing Normalized Communication Sets

Following normalization, we coalesce communication events to eliminate redundant data transfers. There are two types of coalescing operations that can be applied to a pair of communication events:

---

*Normalization could be extended to support more complex affine expressions without much difficulty.
• Subsumption: Identify and eliminate a communication event (nearly) completely covered by another.

• Union: Identify and fuse partially overlapping communication events, which do not cover one another.

Both operations eliminate communication redundancy, however these two cases are handled separately at compile time.

For both cases, the coalescing algorithm first tests to see if communication events are compatible. To be compatible, both communication events must correspond to the same distributed array, be reached by the same HPF alignment and distribution directives, have the same communication type (read or write) and be placed at the same location in the intermediate code.

**Subsumption** The subsumption of one communication event by another requires that the non-local data of the subsuming event be a superset of that of the subsumed one. According to the model for normalized data and CP references, this implies that both messages represent data shifts of constant width in one or more dimensions. For example, a single-dimensional shift of a submatrix owned by a processor would correspond to sending a few rows or columns to a neighboring processor. A pair of shifts must be in the same direction and along the same dimensions for communication events to be compatible. For instance, trying to coalesce a shift that sends two rows and a shift that sends two columns of a submatrix is infeasible.

For two compatible shift communications, where the *shift width* of one is larger than that of the other, the volume of data transferred can be reduced by completely eliminating the smaller shift since its data values will be provided by the larger one. For dimensions not involved in a shift, the ranges of the data reference subscript value numbers in the subsuming communication event must be supersets of the corresponding ranges in the subsumed event. For data dimensions involved in a shift,
subsumption does not require that their range be a strict superset of the corresponding ranges in the subsumed event. If the subsuming communication has a wider shift width, but doesn’t have a large enough range for its loop induction variable, the coalescer synthesizes a new induction variable with extended range to cover the induction variable in the subsumed event as well. This range-extension technique is very effective for avoiding partially-redundant messages and generating simple communication code.

**Union** Coalescing partially overlapping communication events requires less strict conditions than subsumption. Given normalized data and CP references, The coalescer will only try to union communication events that have common shift dimensions and directions.

The dHPF compiler uses the Omega library [KMP+96] to implement operations on integer tuples for its communication analysis and code generation [AMC98]. We apply integer set-based analysis to determine the profitability of unioning two communication events. We construct sets that represent the global, non-partitioned data accessed by the non-local references of each communication event. If these sets intersect, this implies that the communication sets for some pair of processors may intersect. In this case, the algorithm will coalesce the two communication events by unioning them. If the sets do not intersect, then the communication sets for any pair of processors also do not intersect, which implies there is no redundancy to eliminate, so coalescing is not performed.

As described in the previous section, generation of a coalesced communication set can require that new induction variables be synthesized with extended range. When unioning communication sets, this transformation is applied to an induction variable appearing in any dimension as necessary.
4.3 Processor Set Constraints

As described in chapter 3, a symbolic number of processors can lead to complex communication sets and thus complex generated code to enumerate these sets. To reduce the complexity of the generated code, we augmented dHPF to introduce constraints that sharpen information about relationships between processors, outside of the virtual processor mapping formulation.

4.3.1 Necessity of Constraints

The virtual processor data mappings described previously, are used to derive the partner processors with which a particular processor $p$ has to exchange data. These sets are overly general because they represent possible communication with any partner on the virtual processor grid, not with the real partners that exist at runtime.

![Diagram showing some possible right partners](image)

Figure 4.4: Unconstrained Right Partners

Figure 4.4 shows a two-dimensional BLOCK style data distribution and shows possible communication of elements along the right boundary with a neighbor of
processor $p$. If the compiler can determine that the communication event is a single-dimensional shift, then there is no need to contemplate possible virtual processors that have a coordinate that differs on the other dimension. The compiler can assert that the coordinate in the other dimension on the partner processor is the same as that of $p$.

More precisely, if processor $p$ has coordinates $(p_0, p_1, ..., p_{d-1})$ in a $d$-dimensional processor grid and the communication event has been determined to occur only along dimension $i$, then if the partner processor is $w$ with coordinates $(w_0, w_1, ..., w_{d-1})$, we can add the following constraints to the integer set that represents the potential virtual partner processors:

$$\bigwedge_{j \neq i} p_j = w_j$$  \hspace{1cm} (4.1)

These constraints do not require the introduction of symbolic products, and thus
can be used in conjunction with dHPF’s virtual processor representation to simplify the analysis of and code generation for processor sets. Figure 4.5 represents a constrained virtual processor set for a right shift communication. The set still contains imprecision in the horizontal dimension, but we have eliminated the imprecision in the vertical dimension.

Adding processor set constraints is also necessary to make integer-set based analysis and code generation techniques efficient at compile time.

4.4 HPF/JA-inspired Extensions

The Japanese Association for High Performance Fortran proposed several new HPF directives to enable users to fine-tune performance by precisely controlling communication placement and providing a limited means for partially replicating computation. These directives are known as the HPF/JA extensions [SIOT98]. dHPF implements variants of the HPF/JA extensions for several purposes, as we describe below.

Eliminating Communication. To enable an HPF programmer to avoid unnecessary communication that can’t be eliminated automatically by an HPF compiler without sophisticated analysis, dHPF supports the HPF/JA LOCAL directive. The LOCAL directive asserts to the compiler that communication for a set of distributed arrays (specified as parameters to LOCAL) is not needed in a particular scope.

Partially Replicating Computation. Enabling an HPF programmer to partially replicate computation to reduce communication can be important for high performance. For this reason, we provide extended support in dHPF for the ON HOME directive to enable users to partially replicate computation in shadow regions. This support was inspired by the HPF/JA ON EXT_HOME directive, but enables more precise replication control. The HPF/JA ON EXT_HOME directive causes computation to be partially-replicated for all elements in an array’s shadow regions. A drawback of the
ON EXT_HOME is that it can cause computation for elements in an array’s shadow region that are not needed. To avoid this undesirable effect, we extended HPF’s ON HOME directive to allow multiple ON HOME references to be specified; this provides more precise control over partial replication than ON EXT_HOME and enables the manual specification of partially replicated computation partitionings similar to ones generated semi-automatically by the dHPF compiler for localization [AJMCY98]. Figure 4.6 shows how using our extended ON HOME directive enables an application developer to selectively partially replicate a computation to fill a portion of the shadow region (not all of the shadow region may be needed in a particular step of the computation). In the figure, each light gray square represents an owned section for a processor; the enclosing squares represent the processor’s shadow region. dHPF’s extended ON HOME directive enables us to include or exclude the “corners” that would be filled by the HPF/JA EXT_HOME directive. Dark gray sections represent areas where computation was partially replicated using the directives shown in the figure.

Enabling Explicit Communication. The HPF/JA REFLECT directive was designed to support explicit data replication into shadow regions of neighbors. REFLECT was designed for use in conjunction with the LOCAL directive to avoid redundant
communication of values. At the point a \texttt{REFLECT} directive occurs in the code, communication will be scheduled to fill each processor’s shadow regions for a distributed array with the latest values from neighboring processors. In dHPF, we implemented support for an extension of the \texttt{REFLECT} directive that enables more precise control over the filling of the shadow regions. Our extension to \texttt{REFLECT} enables the programmer to specify the dimensions and the depth of an array’s shadow regions to fill. By specifying a single dimension and width, we can \textit{selectively} fill all or part of the shadow region along that dimension. If corner elements are needed, multiple dimensions can be specified at once in a single entry, which will include them. Figure 4.7 shows 2D array $a$’s left and top shadow regions selectively filled in from the values owned by its left and top neighbors.

### 4.5 Partially Replicated Computation

The dHPF compiler’s ability to compute statements on multiple home locations enables it to partially replicate computation. This approach, used judiciously, can significantly reduce communication costs and even, in some cases, eliminate com-
communication altogether for certain arrays.

With dHPF, computation can be partially replicated either automatically by the compiler [AJMCY98], or manually using the extended `ON HOME` directive. Figure 4.8 illustrates the technique for a simple case: computation for a panel of columns has been replicated on neighboring processors (the owning processor and the non-owning one) by using an extended computation partitioning as described in section 4.4. The figure shows the locally-allocated sections of distributed array \( \textbf{a} \) for two different processors. The diagonally-striped regions represent the each processor’s “owned” section; the squares enclosing each owned section represent surrounding shadow regions. The shaded portion of processor \( p+1 \)’s shadow region represents the replicated computation of one column owned by processor \( p \), as shown by the shaded portion of processor \( p \)’s owned region.

### 4.5.1 Interprocedural Communication Elimination

dHPF’s communication analysis and generation is procedure-based; dHPF does not currently support interprocedural communication analysis and placement. However, using our HPF/JA-inspired directive extensions enables us to eliminate communication of values that we know are available as a side effect of partially-replicated
computation or communication elsewhere in the program.

The combination of the HPF/JA LOCAL, extended ON HOME and REFLECT directives yields all of the benefits of interprocedural communication analysis, without its complexity.

4.6 General Communication Aggregation

Often, applications access multiple arrays in a similar fashion. When off-processor data is needed, this can lead a pair of processors to communicate multiple variables at the same point in the program. Rather than sending each variable in a separate message, we extended dHPF to ship multiple arrays in a single message to reduce communication overhead.

The dHPF compiler is able to combine multiple messages addressed to the same recipient from a particular sender into a single message when certain criteria are satisfied. This optimization enabled dHPF to reduce communication frequency even more, at the minor cost of sending bulkier messages.

Aggregation Criteria

Two single logical communication events for different distributed arrays may be safely aggregated if the communication characteristics are conformant. To be conformant (1) both arrays should be mapped to the same HPF template and aligned in the same manner and (2) the communication should flow in the same direction along the same dimensions of the arrays. This guarantees that the partner processor sets are conformant too.

4.7 Load-balanced BLOCK-style partitionings

Initially, the number of elements per partition in block-style partitionings in dHPF was determined by dividing the number of elements in the array dimension by the number of processor partitions in that dimension. If the division was not exact (i.e.
the result is not an integer), then the ceiling of the result was taken:

\[
b_d = \left\lceil \frac{s_d}{p_d} \right\rceil \tag{4.2}\]

In this formula, \(b_d\) is the block size for dimension \(d\), \(s_d\) is the total number of elements of the partitioned array in dimension \(d\), and \(p_d\) is the number of processor partitions along dimension \(d\).

![Diagram of unbalanced partitioning]

Figure 4.9: 2D unbalanced BLOCK partitioning.

This type of partitioning results in imperfect load balance for processors owning data tiles at the end of a dimension, if \(\frac{s_d}{p_d}\) is not integral. More formally, if the number of elements assigned to the first \(p_d - 1\) tiles is \(b_d = \left\lceil \frac{s_d}{p_d} \right\rceil\), then the number of elements \(l_d\) assigned to the last tile is \(s_d - (p_d - 1)b_d\). If \(l_d < b_d - 1\) then load imbalance will result. Figure 4.9 illustrates the problem: processors 0 through 2 own full sized blocks of a one-dimensional partitioning of a two-dimensional array, processor 3 owns only a half-sized block leading to an unbalanced computation partitioning. In this example, \(s_d = 25\), \(p_d = 4\) and \(b_d = 7\). The first three processors will get blocks of size 7 and the last processor will get a block of size \(l_d = 4\), which is only slightly larger than \(\frac{7}{2}\).

The solution to this load imbalance problem is to have some processors own \(\left\lfloor \frac{s_d}{p_d} \right\rfloor\) elements and others own \(\left\lceil \frac{s_d}{p_d} \right\rceil\) elements. With this distribution, the maximum difference in the number of elements owned by two tiles is only one, which leads to the
most even balance of the partitioning among processors. In fact, \( s_d \mod p_d \) processors get blocks of size \( \left\lfloor \frac{a_d}{p_d} \right\rfloor \) and the remaining processors get blocks of size \( \left\lceil \frac{a_d}{p_d} \right\rceil \).

4.7.1 Implementation in dHPF

The most important implementation insight for implementing block-style partitionings where the extent along a partitioned dimension can differ among blocks is that different block sizes for processor data tiles need to be handled \textit{only} in portions of the SPMD program that deal with communication sets. Computation-only sections of the program can be handled by assuming that they work using the \textit{current tile’s} block size.

Integer-set formulations using symbolic block sizes, as described in sections 3.1.5 and 4.3, as well as in [AMC98], are critical to support the \textit{multiple} symbolic block sizes needed for load-balanced partitions.

We enhanced several aspects of dHPF to support balanced tile sizes:

- the integer-set formulation for communication sets,
- code generation to deal with different block sizes for communication partners, and
- virtual processor enumeration

We describe the changes for each of these aspects of the compiler in turn.

\textbf{Integer-set Formulation}

As mentioned earlier in this section, different block sizes should only affect communication sets due to the fact that computation partitionings should only deal with data tiles owned by the local processor; this is true even when using non-owner-computes computation partitionings.

The main change in the integer-set formulation for communication sets is an adjustment to the Layout map (see section 3.1.3), which describes which processors own
which portions of the global array. For block-style partitionings, this map previously
used a compiler-generated variable that held the value $\left\lceil \frac{s}{p} \right\rceil$, namely the ceiling of the
division between the global dimension size, $s$, and the number of processor partitions,
$p$, on that dimension. In the new formulation, the block size of the processor owned
tile is an unknown but fixed variable that can have two possible values: either $\left\lfloor \frac{s}{p} \right\rfloor$
or $\left\lceil \frac{s}{p} \right\rceil$.

Due to the limitations on the expression of symbolic products (and also sym-
monic divisions) in Presburger arithmetic (see section 3.1.5), these integer expressions
cannot be represented directly inside the integer-set core of dHPF. They can be re-
presented by describing the relation between the unknown but fixed local block size
and the partner’s block size. The following relation holds for the unknown local block
size ($\beta$) and the unknown partner’s block size ($\pi$):

$$\left\lfloor \frac{s}{p} \right\rfloor \leq \beta \leq \left\lceil \frac{s}{p} \right\rceil$$
$$\left\lfloor \frac{s}{p} \right\rfloor \leq \pi \leq \left\lceil \frac{s}{p} \right\rceil$$
$$\pi - 1 \leq \beta \leq \pi + 1$$

That is, both $\beta$ and $\pi$ can take either the floor value or the ceiling value of the
division, but for a particular block it is not known (within the framework) whether
its value is the floor or the ceiling. For this reason, its partner’s block size may either
be one less, one more, or equal to the local block size.

This is an imprecise formulation, but can be harnessed effectively to produce
simple communication set formulations, while supporting this extended block size
generality.

Generating Omega code templates for sets with this imprecision can produce many
conditionals that verify the relations between block sizes. For this reason, it is im-
portant to use this imprecise formulation only when necessary (when dealing with
communication). Sets that deal only with computation partitionings should not have
this overhead, except when communication is pinned inside a loop nest.

The decoupling of differing block sizes for computation partitioning sets is effective only when communication can be fully vectorized or hoisted out of enough loop levels to make evaluation of these differing block sizes less significant when compared to the computation costs. If communication cannot be vectorized and is pinned inside heavy computational loops then the cost of supporting differing block sizes can be elevated.

**Code Generation**

Code generation for these generalized sets proceeds as it did for the fixed block size sets, mainly because the previous formulation already supported a single unknown block size. The extension to two unknown block sizes (local and partner) with some constraints in the relation between them does not change the code generation for the integer-tuple enumeration loops.

The changes come in the code that needs to be generated around the loops to retrieve the local block size as well as the partner’s block size for the current local tile and its communication partner. To simplify the generated code and to avoid recomputation, these operations have been implemented as runtime lookups in pre-computed tables managed by the dHPF runtime library. This choice also gives the advantage of altering the mapping of tile sizes in any way necessary to achieve better load balance, i.e. assign all $s \mod p$ blocks of size $\left\lfloor \frac{A}{p} \right\rfloor$ to the first processors in the dimension or assign them in a staggered fashion.

**Virtual Processor Enumeration**

As described in section 3.1.5, dHPF generates additional code to manage the imprecise processor enumeration templates produced by the Omega library. This additional code handles the enumeration of communication partners for particular data tiles owned by the local processor. Previously, this enumeration could be done in terms of
the global array coordinates owned by the particular partner processor. These global array coordinates could be used to identify the start of each data tile owned by a processor.

Generating the precise partner enumeration loop was a question of determining what the starting and ending global coordinates were and mapping them onto the respective tile starting points. The enumeration loop could then stride by the fixed block size to get to the next tile’s starting point. Figure 4.10 represents this fixed block size enumeration. Each partner tile in the figure has the same size in both dimensions, as the tile owned by processor $p$.

![Enumerated Communication Partners](image)

Figure 4.10: Fixed-size communication partners.

With different block sizes for data tiles, the previous strategy for generating the precise partner enumeration loop is not feasible. Specifically, it is not possible to stride by a constant (runtime) block size between tile starting points. For this reason, dHPF enumerates communication partners by their coordinates in the processor array space. This processor array space corresponds to that declared by the HPF PROCESSORS directive. Without losing generality, processors can be identified by a $n$-tuple with
each element in the tuple varying between 0 and \( k_d - 1 \), where \( k_d \) is the number of elements of the \texttt{PROCESSORS} array in dimension \( d \), and \( n \) is its rank. In this space, the stride between enumerated partners is always one.

![Enumerated Communication Partners](image)

**Figure 4.11**: Different-sized communication partners.

Using a precomputed runtime mapping it is possible to translate between a tile’s coordinates in the processor array space and its \textit{starting coordinates} in the global array space needed for the correct execution of each communication phase. Figure 4.11 illustrates the case where partners differ in the sizes of one or more of their dimensions. In particular, the tiles have the same width, but different heights for this two-dimensional example.

In multiple dimensions, tile sizes are constrained so that all adjacent tiles share the same extent on each dimension of an edge they have in common. This property could be exploited to simplify the integer-set formulation for the multiple block size support, but the dHPF compiler has handled the current experiments without problems.
Chapter 5

Line Sweeps and Multipartitioning

5.1 Line sweeps

The implementation of numerical methods on parallel computers requires careful algorithm design to minimize communication volume and synchronization frequency to maximize the parallel efficiency.

There are many numerical methods on multidimensional domains that are fully parallel. In such cases, the computations can be done independently for every point on the domain and communication can be executed in an efficient loosely-synchronous manner.

Other numerical methods have more restricted forms of parallelism. One important class of numerical methods is parallel by line, with the computation of a point on the domain depending upon other points along the line, but independent of other points on all other lines in the domain [NNN93]. Alternating Direction Implicit (ADI)/Approximate Factorization (AF) methods are in this category, as well as fractional step methods, red/black line Gauss-Seidel, and Jacobi by line [NNN93, Na92a]. These methods are characterized by constant-sized stencil computations on a 1D sub-domain (line). When implemented in Fortran or other imperative languages, these stencils appear as single-dimensional data dependences * in the line sweeps.

There are other numerical methods with even more restricted parallelism, e.g., methods that are only parallel by hyperplane. For such methods, the computation of a point on the domain depends on other points on a particular plane. Implicit LU

*Formal definitions of data dependence appear in Appendix B
and Successive Over Relaxation (SSOR) are examples of this category [NNN93].

5.2 Data Partitionings for Line Sweep Computations

The strategies used to partition an application’s data and computation for parallel execution play a fundamental role in determining both the range of the application’s suitable parallelizations and their potential efficiency. High Performance Fortran (HPF) and OpenMP, both standard high-level models for parallel programming, provide a narrow set of choices for data partitioning and computation partitioning, respectively. While these partitionings can lead to good performance for loosely synchronous computations, they are problematic for more tightly-coupled codes.

Line-sweep computations, used to solve one-dimensional recurrences along each dimension of a multi-dimensional discretized domain, are an important class of tightly-coupled computations that are not supported well by the HPF and OpenMP partitioning choices.

This style of computation is the basis for Alternating Direction Implicit (ADI) integration, a widely-used numerical technique for solving partial differential equations such as the Navier-Stokes equation. Recurrences along each dimension of the data domain make this class of computations difficult to parallelize effectively.

Line-sweep computations on a multidimensional domain can be implemented using block-style data distributions. Near-neighbor communication on the partition boundaries will be required because of the 1D stencil nature of the sweep. A cyclic distribution is not profitable for line-sweep computations because it would require communication of data values on every block boundary.

Considering HPF block-style partitionings for a multidimensional array on which line sweeps will be performed, there are only two options: to partition the swept * dimension or not to partition it. The partitioning of the swept dimension will require

\[\text{We call the dimension along which the line sweep will occur the swept dimension}\]
near-neighbor communication on the processor boundary, thus implying that the number of partitions should be small. On the other hand, having few partitions restricts the number of processors that can be active, so efficient parallelism requires a number of partitions at least equal to the number of processors. If the dimension is not partitioned, the line sweep computation can be performed efficiently as a 1D local stencil.

The complexity of deciding how to partition an array for multidimensional line-sweep computations, comes from the fact that, in general, line sweeps will be needed along all of the dimensions of the domain. If only one dimension is partitioned, then $d - 1$ one-dimensional local sweeps can be performed without any communication between processors. To perform the remaining sweep, one option is to transpose the partitioned dimension with any other dimension in the multidimensional array, and then compute the sweep locally. In an iterative computation, the transposed dimensions must be transposed back for the next iteration. Figure 5.1 illustrates the transpose scheme for performing multidimensional line sweeps.

![Diagram showing sweeps along x and z, transpose, and sweep along y](image)

Figure 5.1 : Line sweeps with transpose.

A second possibility is to compute along the partitioned dimension, which requires communication at the processor boundaries. This involves a degree of serialization
because only the processor owning the first block along the partitioned dimension can start computing while the other processors must wait. There is the possibility of obtaining partial parallelism by pipelining the computation. In a pipelined computation, some boundary elements needed by the next processor along the dimension are sent soon after they are computed, instead of waiting for the whole boundary to be completed. Figure 5.2 shows the fully serialized case, in which one column of data is transferred from processor P0 to processor P1 as a single message, causing processor P1 to wait for P0 to finish all of its computation and transfer the data. Figure 5.3 shows the partially parallel pipelined case, in which the same column of data needs to be transferred, but the column is divided into blocks and as P0 completes the computation of each block, it immediately sends it to P1 which can then start computing rows that depend on the elements in the block it received. Selecting a good granularity for the pipeline is necessary to balance parallelism against communication overhead. If the granularity is too fine, then many small messages need to be sent between processors, which can cause a high communication overhead. If the granularity is too coarse, then parallelism will suffer.

![Diagram](image)

Figure 5.2: Serialization induced by line sweep.

Previous work [CFH+92, BCG+95, AJMCY98, CMC02, CMC03] showed that a medium-grained pipelined approach is more efficient than the transpose method.
for moderately large arrays. Our work makes compiler-generated multipartitioning possible, which is superior to both of these schemes.

5.2.1 Multipartitioning

Multipartitioning, as defined by Naik et al. [Naik92b, NNN93], is a data distribution strategy for line-parallel methods, which enables their execution in a fully-parallel, load-balanced fashion. It is a strategy for partitioning multidimensional arrays. Its main property is that for any directional sweep across the array, all processors are active in each step of the computation, there is nearly perfect load-balance, and only coarse-grain communication is needed.

Johnson et al. [JSS87] describe a 2D domain decomposition strategy, now known as a multipartitioning, for parallel implementation of ADI (Alternating Direction Implicit) integration on a multiprocessor ring. They partition both dimensions of a 2D domain to form a $p \times p$ grid of tiles. They use a tile-to-processor mapping $\theta(i,j) \equiv (i - j) \mod p$, $0 \leq i,j < p$, to map from the $[i,j]$ coordinates of each tile to its corresponding processor. Using this mapping for an ADI computation, each processor exchanges data with only its 2 neighbors in a linear ordering of the processors, which maps nicely to a ring.
Bruno and Cappello [BC88] devised a 3D partitioning for parallelizing 3D ADI integration computations on a hypercube architecture. They describe how to map a 3D domain cut into $2^d \times 2^d \times 2^d$ tiles on to $2^{2d}$ processors with a tile-to-processor mapping $\theta(i, j, k)$ based on Gray codes: $\theta$ maps tiles adjacent along the $i$ or $j$ dimension to adjacent processors in the hypercube, whereas tiles adjacent along the $k$ dimension map to processors that are exactly two hops distant. They also show that no hypercube embedding is possible in which adjacent tiles always map to adjacent processors. Bruno and Cappello also noted that multipartitionings could be generalized with processors assigned more than one tile per hyperplane of a multipartitioning.

![Figure 5.4: 2D Multipartitioning on 5 processors.](image)

Descriptions of multipartitioning in the literature correspond to **diagonal multipartitionings** for 2D or 3D problems. Diagonal multipartitionings are a generalization of Johnson et al.'s 2D partitioning strategy that are more broadly applicable than the Gray code based mapping described by Bruno and Cappello.

Diagonal multipartitioning achieves load balance by partitioning data into $p^{d-1}$ tiles, where $p$ is the number of processors and $d$ is the number of partitioned array dimensions. Each processor is assigned $p^{d-1}$ tiles along diagonals through each of
the partitioned dimensions. Figure 5.4 shows a 2D multipartitioning distribution for 5 processors; the number in each tile represents the processor that owns the block. Each processor owns a single tile in each column and row, thus achieving load-balanced execution for any dimensional sweep. The 3D diagonal multipartitionings described by Naik et al. partition the data into $p^3$ tiles, with each processor's tiles arranged along wrapped diagonals through the 3D volume.

A study by van der Wijngaart [Van93] of implementation strategies for hand-coding parallelizations of Alternating Direction Implicit Integration (ADI) found that 3D multipartitioning was superior to both static block partitionings using a wavefront parallelization strategy, as well as a dynamic block partitioning strategy where repartitioning was performed by transposing between phases of the computation. Another study by Naik [Nai92a] examined differences between multipartitioning with static block partitioning strategies with respect to their tolerance for load imbalance. His findings show that three-dimensional partitionings of 3D data are more tolerant of load imbalance than 2D partitionings, and that three-dimensional multipartitionings are the most tolerant of load imbalance as long as the number of processors is not overly large for the data size.

Naik et al. [NNN93] present a detailed analytical model to compare the predicted performance of the partitioning schemes for the Beam-Warming Approximate Factorization scheme implemented in NASA's ARC3D application. Their analytical results show that the best speedup for these types of algorithms, operating on 3D domains, will be under three-dimensional multipartitioning. This analysis also considers other parameters of execution such as communication and buffering overheads, which might be larger for multipartitioning than for unpartitioning. They present implementation results of ARC3D using a 2D unpartitioning scheme and a 2D multipartitioning scheme on the Victor parallel processor (16 processors). Their results indicate that the 2D unpartitioning scheme with a pipelined communication structure, achieves better speedup than the 2D multipartitioning version. The authors also present the
predicted performance for a larger problem size, in which 3D multipartitioning would perform better than the others. These results are very dependent on the particular machine parameters and constraints and should vary significantly on more modern architectures.

Diagonal multipartitionings described in the literature consider only one tile per processor per hyper-rectangular slab along a partitioned dimension. In 2D, these partitionings can be performed on any number of processors, \( p \); however, in 3D they are only useful if \( p \) is a perfect square.

![3D Multipartitioning](image)

Figure 5.5: 3D Multipartitioning on 16 processors.

For an \( n \)-dimensional multipartitioning on \( p \) processors, the expression \( p^{\frac{1}{n}} \) must be an integer. Thus, a 3D multipartitioning requires the number of processors to be a perfect square. However, multipartitioning can be applied to any two dimensions of an \( n \)-dimensional array allowing use of an arbitrary number of processors. Figure 5.5 shows a 3D multipartitioning distribution for 16 processors, each processor owns a tile in each column, row and plane. The number in each tile indicates the processor that owns the block. This 3D diagonal multipartitioning (there are many) is specified
by the tile to processor mapping \( \theta(i, j, k) \equiv ((i - k) \mod \sqrt{p}) \sqrt{p} + ((j - k) \mod \sqrt{p}) \) for a domain of \( \sqrt{p} \times \sqrt{p} \times \sqrt{p} \) tiles where \( 0 \leq i, j, k < \sqrt{p} \), where \( \sqrt{p} = 4 \).

### 5.2.2 Generalized Multipartitioning

As described in the literature (and summarized in section 5.2.1), multipartitionings for three or more dimensions have only been considered for numbers of processors that are perfect powers. In the case of 3D multipartitionings, the number of processors has to be a perfect square. This section describes a generalization of multipartitioning that solves the problem of computing optimal multipartitionings for \( d \)-dimensional data volumes for an arbitrary number of processors.

Diagonal multipartitionings can be applied to partition \( d \)-dimensional data onto an arbitrary number of processors \( p \) by cutting the data into \( p \) slices in each dimension, \( i.e., \) into an array of \( p^d \) tiles. In 2D, this yields an optimal multipartitioning (equivalent to those described by Johnsson et al. [JSS87]). A multipartitioning is optimal for a particular number of processors if no other multipartitioning exists that has lower communication cost according to a cost model that considers both fixed overhead for communicating and overhead proportional to the area of the hyper-surfaces that must be communicated. For \( d > 2 \), diagonal multipartitionings are only optimal and efficient when \( p^{d-1} \) is integral.

How general can multipartitioning mappings be? A necessary condition to support load-balanced line-sweep computation is that in any hyper-rectangular slab defined by adjacent cuts along a partitioned dimension, each processor must have the same number of tiles.

To enable us to formally define a generalized multipartitioning, we first introduce some notation.

- \( p \) denotes the number of processors. We write \( p = \prod_{j=1}^{s} \alpha_j^{r_j} \), to represent the decomposition of \( p \) into prime factors. Each prime factor \( \alpha_j \) of \( p \) occurs with some multiplicity \( r_j \).
- $P$ denotes the set of processors \{1,\ldots,p\}.

- $d$ is the number of dimensions of the array to be partitioned. The array is of size $n_1,\ldots,n_d$. The total number of array elements is $n = \prod_{i=1}^{d} n_i$.

- $\tilde{\gamma}$ denotes an array partitioning. Each $\gamma_i$, $1 \leq i \leq d$, in $\tilde{\gamma}$ represents the number of tiles into which the array is cut along its $i$-th dimension. We can consider a $d$-dimensional array as a $\gamma_1 \times \ldots \times \gamma_d$ array of tiles. Each tile in the array of tiles is identified by its coordinates $\vec{t}$, a $d$-dimensional vector with each element $t_i$, $1 \leq i \leq d$, such that $0 \leq t_i \leq \gamma_i - 1$. A hyper-rectangular slab $H$ in dimension $i$ is defined as a set of all tiles with the same $i$-th coordinate, i.e., there exists $k$, $0 \leq k < \gamma_i$, such that $H = \{\vec{t} \mid t_i = k\}$.

Using this notation, we define a generalized multipartitioning as an ordered pair $(\tilde{\gamma}, \theta)$, where $\tilde{\gamma}$ is an array partitioning (as defined above) and $\theta$ is a tile-to-processor mapping. A tile-to-processor mapping $\theta$ maps a tile $\vec{t}$ to a processor $\theta(\vec{t})$ in $P$. The mapping $\theta$ must satisfy two constraints for $(\tilde{\gamma}, \theta)$ to be a multipartitioning. First, it must have the balance property, i.e., for any hyper-rectangular slab $H$ along any dimension $i$, when $\theta$ is applied to each of the elements in $H$, it must map an equal number of tiles to each processor; namely, $\forall(q,r) \in P \times P, |\theta^{-1}(r) \cap H| = |\theta^{-1}(q) \cap H|$. Second, it must have the neighbor property, i.e., for all tiles $\theta^{-1}(q)$ owned by a single processor $q \in P$, and any particular dimension $i$, $1 \leq i \leq d$, there exists a single processor that owns all tiles “right-adjacent” to any tile in $\theta^{-1}(q)$ along dimension $i$; similarly, there exists a single processor that owns all tiles “left-adjacent” to any tile in $\theta^{-1}(q)$ along dimension $i$. More formally, if $\vec{e}$ is a canonical basis vector, then we must have $\theta(\vec{t}) = \theta(\vec{u}) \Rightarrow \theta(\vec{t} + \vec{e}) = \theta(\vec{u} + \vec{e})$ when the tiles with coordinates $\vec{t}$, $\vec{u}$, $\vec{t} + \vec{e}$, and $\vec{u} + \vec{e}$ exist in the array of tiles.

We call any slab in which each processor has the same number of tiles a balanced slab. This raises the question: is it possible to find a way to partition a $d$-dimensional array into tiles and assign the tiles to processors so that the mapping
possesses the **balance** and **neighbor** properties of a multipartitioning? The answer is yes. In [DCFMC02] and [CDFMC01] we show that such an assignment is possible if and only if the number of tiles in each hyper-rectangular slab along any partitioned dimension is a multiple of \( p \). They include the description of a “regular” solution that enables the method to guarantee that the neighboring tiles along any one coordinate direction of all tiles mapped to a processor all belong to a single processor. This property of multipartitionings is essential for fully-vectorized, directional-shift communication to be efficient.

The strategy for computing generalized multipartitionings has three parts:

- An objective function for computing the cost of a line sweep computation for a given multipartitioning: it determines the number of elements transferred between all processors in all communication phases of a full, multi-dimensional line sweep. The function assumes that line sweeps are going to be computed along all distributed dimensions, and then computes the cost of transferring hyper-surfaces between adjacent tiles.

- A cost-model-driven algorithm for enumerating all valid general multipartitionings which uses the objective function just described to compute the dimensionality and tile size of the best multipartitioning. This exhaustive evaluation is computationally tractable because the number of valid solutions is small [DCFMC02, DMCFC03].

- and an algorithm for computing a mapping of tiles to processors: Once the size and shape of the multipartitioning have been determined, we must compute a valid mapping of tiles to processors. So that the **balance** and **neighbor** properties are preserved.

Figure 5.6 shows a generalized multipartitioning distribution for 6 processors. The number inside each square represents the processor assigned to that tile. Note that
each processor owns several tiles per hyperplane. The details on how to construct
generalized multipartitionings are described in [DMCFC03].

5.3 Implementation of Multipartitioning in dHPF

5.3.1 HPF Directives & Multipartitioning

High Performance Fortran (HPF) [KLS+94] was designed as a data-parallel extension
of Fortran 90 with the addition of data partitioning directives. These directives
are used by an HPF compiler to distribute arrays between processors in a parallel
machine. Chapter 3 contains a description of the PROCESSORS, TEMPLATE, DISTRIBUTED
and ALIGN directives.

The combination of expressions using the DISTRIBUTED and ALIGN directives is
not enough to specify a multipartitioning directly in HPF, because dimensions in a
multipartitioning cannot be distributed or aligned separately: they have to be coupled
to preserve the properties of a multipartitioning.

5.3.2 Syntactic Extensions

To specify that multipartitioning should be used to distribute a multidimensional
array, we extended the dHPF compiler to accept MULTI(d) as a distribution specifier.

Figure 5.7 shows two four-dimensional arrays x and y multipartitioned using our
parameter (n = 64)

double precision x(0:n-1,0:n-1,0:n-1,15)
double precision y(0:n-1,0:n-1,0:n-1,5)

CHPFS PROCESSORS p(NUMBER_OF_PROCESSORS())
CHPFS TEMPLATE t3(0:n-1,0:n-1,0:n-1)
CHPFS TEMPLATE t2(0:n-1,0:n-1,0:n-1)

CHPFS ALIGN x(i,j,k,*) with t3(i,j,k)
CHPFS ALIGN y(i,j,k,*) with t2(i,j,k)

CHPFS DISTRIBUT t3(MULTI(1),MULTI(1),MULTI(1)) onto p
CHPFS DISTRIBUT t2(MULTI(1),*,MULTI(1)) onto p

Figure 5.7: Using dHPF's layout directive extensions for multipartitionings.

extensions to HPF's data layout directives. The x and y arrays are aligned, respectively, to templates t3 and t2, which represent arrays of virtual processors. These templates are then multipartitioned onto a linear array of processors p that contains an element for each available physical processor. The template t3 is distributed onto the first (and only!) dimension of the processor array p using a three-dimensional multipartitioning; t2 is distributed using a two-dimensional multipartitioning. (For conciseness, we have shown the use of two completely different multipartitioning distributions in a single example; it is unlikely that such using more than one multipartitioning distribution in a program would be useful.) The presence of the MULTI keyword in a template dimension in a distribution directive indicates that this dimension is part of a multipartitioning. Following the MULTI keyword in each dimension is a subscript 1, which indicates that the template dimension should be distributed
over the first dimension of the target processor array. Unlike BLOCK or CYCLIC distributions, which map one-to-one onto processor array dimensions, multipartitioned template dimensions map many-to-one onto a processor array dimension. For a multipartitioning to be valid, at least two template dimensions must be mapped to the same processor array dimension with a MULTI directive. The processor array dimension index is used with the MULTI keyword so as not to preclude distributing an array with two or more independent distributions (e.g., using a combination of MULTI and BLOCK, or two independent MULTI distributions) onto a multi-dimensional processor array. In practice, we don't see any need for such complex distributions; in the dHPF compiler, if any subset of a template's dimensions are distributed using a multipartitioning, we require that all non-multipartitioned dimensions be unpartitioned. (For this reason, our prototype implementation of multipartitioning directives in dHPF uses a simpler syntax.)

All array dimensions distributed (through a template) onto the same processor dimension using MULTI are partitioned using a tiling induced by a generalized multipartitioning appropriate for the number of available physical processors. The number of template dimensions that are distributed using a MULTI directive represents an upper bound on the number of dimensions that will actually be partitioned at run time. If \( d \) dimensions of a template are marked using the MULTI keyword, then, depending on the number of processors and the template extent, the run-time system will choose to partition between 2 and \( d \) dimensions to achieve balanced parallelism with minimum communication cost according to our objective function. Thus, at run-time \( x \) may be partitioned in either two or three dimensions. Using the objective function, the run-time library selects how many dimensions of \( x \) to partition, which dimensions to partition, and how many cuts are necessary. For \( y \), the multipartitioning must partition only the first and third dimensions; as before for \( x \), the run-time library determines the number of cuts and the mapping of tiles to processors.
5.3.3 Virtual Processors

The integer set analysis framework used by dHPF [AMC98, MCAB+02] supports BLOCK partitionings of arbitrary size onto a symbolic number of processors. To support multipartitioning, we extended this model to treat each tile in a multipartitioned array as a block in a virtual BLOCK partitioned array. In the case of diagonal multipartitionings, this is an array of $p^{\frac{d}{d-1}}$ virtual processors, with each processor owning $p^{\frac{d-1}{d-1}}$ of these virtual processors.

Implementing this virtual processor model, requires mapping between virtual and physical processors. Each virtual processor is identified by its tile position indices, a $d$-dimensional tuple representing its coordinates in a virtual processor array (in column-major order). These indices are used to index into a virtual to physical processor mapping. When a tile needs data from another tile, the other tile’s coordinates are computed from the data indices, the virtual to physical processor mapping is then used to determine which physical processor owns the required data.

5.3.4 Code Generation

Code generation for multipartitioning is a generalization of code generation for BLOCK partitioning. Within dHPF, the integer set framework is used to generate code for blocks of arbitrary size, at arbitrary positions. The generated code for computations and communications for such a block corresponds to the kernel code for each tile of the multipartitioned distribution.

All communication and computation performed for a tile is defined in terms of the data mapped to that tile. Since more than one virtual processor is assigned to each physical processor, the tile position indices have to be adjusted on each physical processor as it cycles through its tiles. Computation for a loop nest occurs by having each processor perform the computation on each of its owned tiles.

To generate code that handles multiple tiles per processor, a tiling loop that iterates over all the tiles assigned to a physical processor must be wrapped around
the kernel code for a tile. The order in which a physical processor must iterate
over its tiles, for a loop nest, is determined by the loop’s data dependencies (see
Appendix B for definitions of loop-carried and loop-independent dependences). Since
multipartitioning is a multidimensional distribution, the data dimension in which
the outermost dependence is carried is the one that determines the tile iteration
sequence. The order in which each processor iterates over its tiles corresponds to
the loop direction and must satisfy the loop-carried dependences present in the loop
body. (If there are no loop-carried dependences then the iteration sequence follows the
outermost loop index that indexes a multipartitioned dimension within the array.) In
diagonal multipartitioning, as shown in Figure 5.5, the tiles for a processor fall along
a diagonal which spans the d dimensions of the array.

Figure 5.8 shows the tile enumeration order that a processor follows for a horizontal
sweep. The numbers in each tile indicate the order in which they are processed; all
of them belong to a single processor.
5.3.5 Runtime Support

The main runtime components required to support multipartitioned code generated by dHPF are a function to compute virtual-to-physical processor mappings and support for managing multiple dynamic buffers. Each multipartitioned template distribution requires a different virtual-to-physical processor map. These maps are associated with their corresponding template runtime descriptor. These maps are computed once, at the start of program execution, with very little overhead, since their sizes depend on the number of tiles.

5.4 Tile Scheduling & Tile Loop Placement

To execute a loop nest operating on multipartitioned data, a processor performs a portion of the loop’s computation for each tile in a sequence of virtual processor tiles (as shown in figures 5.4 and 5.5 each processor owns several tiles in the domain). Each tile’s computation may require communication. When tiles for a multipartitioned loop nest require communication for processor-crossing true data dependences, parallelism and overall performance is a function of the order in which tile computations are performed on each processor. To avoid unnecessary serialization, the tile enumeration order must be chosen carefully.

This problem of selecting the tile enumeration order is unique to multipartitioning. For block partitionings, tile enumeration order is trivial: each processor has only one block. For cyclic and block-cyclic partitionings, tile enumeration is straightforward—the enumeration is independent for each partitioned data dimension. For multipartitioned data arrays, tile enumeration order must be selected carefully because choosing to iterate through a processor’s tiles in a particular direction along any one multipartitioned data dimension determines the iteration order for the other multipartitioned dimensions as well.

Figure 5.9 outlines the algorithm dHPF uses to select which multipartitioned data dimension will determine the tile enumeration order within a processor. If there is no
communication for processor-crossing true dependences, then any tile enumeration order is equally good. When there is communication for processor-crossing true dependences, then the dimension along which the communication is flowing is selected as the \textit{tile dimension}, and the tile iteration direction is selected to sweep along the tile dimension in the same direction as the communication (if multiple dimensions induce communication, then the outermost dimension is selected as the tile dimension). The flexibility afforded by dHPF’s general computation partitioning model makes it possible to choose computation partitionings so that all processor crossing dependences along only one dimension flow in the same direction. If communication flows in multiple directions in the swept dimension, then deadlock may occur. dHPF avoids this by carefully selecting computation partitionings and carefully scheduling communication operations.

As long as processor-crossing dependences flow along only one dimension of multipartitioned array, any tile enumeration order that does not sweep in the opposite direction of the communication will be correct. However, full parallelism will be realized for such computations only if the tile dimension is the same as the dimension along which communication occurs, and the tile enumeration order is the same as the direction of the communication. If a multipartitioned computation involves communication along more than one dimension, the computation will be partially serialized regardless of tile enumeration order.

5.4.1 Tile Scheduling for Generalized Multipartitioning

In the context of generalized multipartitioning, each processor may own more than one tile per hyperplane, however \textit{all} tiles in a particular hyperplane can be treated collectively for scheduling purposes.

If there are no carried dependences in a particular loop nest that computes over arrays partitioned using generalized multipartitioning, then any tile iteration order suffices. In particular, any iteration order for each hyperplane is sufficient.
ComputeTileEnumerationOrder(LoopNest)

if \( \exists \) processor-crossing dependences in LoopNest

then

tile iteration dimension = data dimension indexed
by subscript with induction variable from
shallowest loop

else

if comm. for processor-crossing dependences flows
in both directions along a single dimension then
error: any static tile schedule will cause deadlock

else

tile iteration dimension = data dimension along
which comm. occurs which is indexed by
induction variable of shallowest loop
tile iteration direction = direction along which
communication flows along tile dimension

Figure 5.9: Algorithm for selecting data dimension and direction for tile enumeration.
If carried dependences exist, then they will affect the ordering of computation among hyperplanes. If carried dependences flow only in one direction, then tiles within each hyperplane can be computed in any order. If carried dependences flow in multiple directions, then the tile enumeration order within a hyperplane becomes critical for correctness.

Figure 5.10: Tile scheduling for generalized multipartitioning.

Figure 5.10 illustrates how tiles are scheduled in a sweep over an array distributed using generalized multipartitioning. In the figure, the planes are being swept from front to back. The front plane should be processed before computing the back plane. Tiles in each plane can be processed in any order: tiles belonging to different processors are processed concurrently by them, while each processor selects an ordering for each set of tiles it owns in a single plane.

For generalized multipartitioning, the tile enumeration loop becomes a hyperplane enumeration loop that traverses hyperplanes in an order consistent with unidimensional dependences. Within each hyperplane, local tiles must be processed in local tile enumeration loops.
5.5 Unidirectional Communication & Tile Enumeration

In a loop nest that expresses one direction of one dimension of a multidimensional line sweep, all loop-carried dependences flow along the direction of the uni-dimensional sweep, due to the characteristics of line-sweep computations.

More formally, a loop nest representing a one-dimensional sweep in one direction (forward or backward) can be represented as shown in figure 5.11.

\[
\begin{align*}
do & \ i_n = l_n, u_n \\
do & \ i_{n-1} = l_{n-1}, u_{n-1} \\
... & \\
do & \ i_k = l_k, u_k \\
... & \\
do & \ i_1 = l_1, u_1 \\
& \ s_1 \ \forall q (\left(\delta_{q;i_k} \neq 0 \land \forall \delta_{q;j\neq i_k} = 0\right) \lor \forall j \delta_{q;j} = 0) \\
& \ s_2 \ \forall q (\left(\delta_{q;i_k} \neq 0 \land \forall \delta_{q;j\neq i_k} = 0\right) \lor \forall j \delta_{q;j} = 0) \\
... & \\
& \ s_m \ \forall q (\left(\delta_{q;i_k} \neq 0 \land \forall \delta_{q;j\neq i_k} = 0\right) \lor \forall j \delta_{q;j} = 0) \\
\end{align*}
\]

Figure 5.11 : General loop nest for a one-dimensional sweep

Figure 5.11 represents the general structure of a loop nest in a line sweep computation. Each statement, \(s_j\), may be a source or sink of one or more data dependences
carried by loop level $i_k$ (the dependence distance vector $\delta_q$ for statement $s_j$, corresponding to the $q$-th dependence incident on $s_j$, has an entry $\neq 0$ for $i_k$ and all other entries equal 0). Multidimensional line sweeps have the property of having separate loop nests for each dimension and for each direction in a sweep (forwards or backwards along a dimension). In each of these loops, carried dependences flow only on the swept dimension and only in the swept direction. Loop-carried dependences might have different dependence distances. There might be loop-independent dependences between statements in the loop body.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
  \hline
  00 & 01 & 02 \\
  \hline
  10 & 11 & 12 \\
  \hline
  20 & 21 & 22 \\
  \hline
\end{tabular}
\caption{Tile coordinates in the global domain.}
\end{table}

Tiles in a multipartitioned $n$-dimensional array can be identified by an $n$-tuple. Each entry in this $n$-tuple represents a tile’s position in a one-dimensional hyperplane. For simplicity, tiles can be enumerated from 0 to $\pi_k - 1$, where $\pi_k$ represents the number of partitions in dimension $k$. Figure 5.12 shows a two-dimensional enumeration of tiles in a distributed array.

**Definition 1 (Communication Direction)** If $\vec{t}_i$ and $\vec{t}_j$ are tuples identifying distinct tiles in an $n$-dimensional domain, communication between these two tiles is classified as forward if $\vec{t}_i$ is the sending tile, $\vec{t}_j$ is the receiving tile and $\vec{t}_i - \vec{t}_j > \vec{0}$, where “$-$” is the component-wise difference of two vectors and “$>$” is the component-wise comparison of two vectors.
Conversely, communication is classified as backward if $i_j$ is the sending tile, $i_i$ is the receiving tile and $i_i - i_j < 0$.

In dHPF’s computation partitioning framework, communication for distributed arrays can be represented with two different array references: the HOME reference (from the computation partitioning) and the data reference. Communication has to be synthesized if the references may reside on different processors at execution time.

For statically analyzable line sweeps (based on affine array references), it is possible to express the HOME (CP) and data references in a canonical form (see section 4.2.1). In this canonical form, the references have the following shape:

- The CP reference is of the form $A(i_1, i_2, \ldots, i_n)$
- The data reference is of the form $A'(a_1i_1' + b_1, a_2i_2' + b_2, \ldots, a_ni_n' + b_n)$

**Definition 2 (Comm. Direction Induced by a Computation Partitioning)**

Given normalized CP and data references: $\overline{i}$ and $\overline{\rho}$, the sign of the symbolic component-wise difference $\overline{\sigma} = \overline{i} - \overline{\rho}$ indicates the direction of the communication induced by the computation partitioning. That is, if $\overline{\sigma} > \overline{0}$ then the communication is forward, and if $\overline{\sigma} < \overline{0}$ then communication is backward.

Let $\Pi = \{(\pi_{ij}, \rho_{ij}) : 1 \leq i \leq m\}$ be the set of computation partitionings and data references for the statements in a multipartitioned line-sweep loop. This set is composed of tuples $(\pi_{ij}, \rho_{ij})$ where $\pi_{ij}$ represents the normalized CP reference and $\rho_{ij}$ represents the normalized data reference for the reference $j$ within statement $i$ in the loop.

Carried data dependences in the line-sweep loop are either all “forward” ($\delta_{ik} > 0$), or all “backward” ($\delta_{ik} < 0$). If all carried dependences are forward, then it is the case that the computation of an element in the array depends on previous elements in the same 1D hyperplane. Now, at tile partition boundaries, array elements depend on elements owned by a previous tile, this implies that the values of those elements
belonging to the previous tile need to be available to the current tile. This means that these values need to be sent from the previous tile to the current tile: i.e. forward communication. A similar argument can be made to establish the relation between backward dependences and backward communication for line-sweeps.

Given these characteristics, it is possible to choose computation partitionings in such a way that communication induced by statements within the loop will flow in only one direction in the swept dimension. The tile enumeration order has already been chosen according to the algorithm in figure 5.9, in such a way that tiles are enumerated in the direction and dimension of the sweep.

**Theorem 1 (Unidirectional CP Selection)** The set of computation partitionings and data references for the statements in the loop, \( \Pi = (\pi_i, \rho_j) : 1 \leq i \leq m \) can be constructed in a way that it induces communication (for true dependences) in the same direction as that of the carried dependences of the loop.

**Proof:** Since all loop-carried dependences have the same direction, then choosing an owner-computes computation partitioning for each statement (communication will only happen for right-hand side expressions) guarantees that carried communication will flow in the same direction as that of the dependences.

Owner-computes computation partitionings induce read communication for non-local right-hand side array elements. Since tiles are being computed in an order that satisfies dependences (due to the tile enumeration algorithm), any necessary right-hand side source data has already been computed and will be communicated before executing the sink statement that uses it.

5.6 Multipartitioning Optimizations in dHPF

This section describes the optimizations performed by the dHPF compiler that specifically target performance issues that arise in generating efficient compiler-parallelized
code for multipartitioned computations. From a processor's perspective, a multipartitioned computation is organized as computation on each of a series of tiles. To avoid unnecessary serialization between physical processors, each processor's tile computations must be scheduled in the proper order. The approach for tile scheduling is described in section 5.4. To achieve good scalability, communication between processors must be as infrequent as possible. Sections 5.6.1 and 5.6.3 describe optimizations to reduce the frequency of communication for compiler-generated multipartitioned computations.

5.6.1 Aggressive Communication Placement

In a loop nest that iterates over an array's data dimensions, processor-crossing true data dependences incident on a particular reference can be preserved by placing communication for that reference at the level determined by the data dependence. Loop-carried dependences can be preserved by placing communication within the carrying loop. Loop-independent dependences can be preserved by placing it at the loop level that is the least common ancestor of the dependence source and sink [Set96].

For a loop nest iterating over a BLOCK partitioned array, communication that occurs only at a tile boundary can be hoisted out of a bounds-reduced loop, as long as communication flows in one direction only (see section 5.4). However, hoisting such communication out of additional enclosing loops for BLOCK partitioned arrays could hurt performance: although it could reduce communication frequency, it would also reduce the potential for wavefront parallelism.

Loop nests iterating over the data dimensions of a multipartitioned array possess the unique property that if there is only communication along a single direction of a single data dimension, it can be hoisted out of all enclosing loops over data dimensions without reducing the available parallelism. In the dHPF compiler, this property is exploited to vectorize communication for processor crossing true dependences (both loop-carried and loop-independent) out of multipartitioned loops. This optimization
enables programmers to nest loops over the dimensions of a multipartitioned array so that they will access memory using a stride-1 access pattern. In contrast, when a BLOCK distribution is used, either the programmer or the compiler must carefully arrange a loop nest with carried dependences to place the dependence-carrying loop at a level that balances communication frequency with wavefront parallelism; this placement may be inconsistent with a stride-1 access pattern and hurt performance by reducing memory hierarchy utilization.

Although multipartitioned distributions enable full parallelism to be achieved for a loop nest that requires unidirectional communication along any one partitioned dimension, if processor-crossing true dependences flow in multiple directions or along multiple data dimensions, then the loop will be partially serialized.\(^2\)

Communication placement in dHPF occurs in two steps. First, the compiler computes an initial placement in which data dependences are used to determine, for each reference, the loop level at which communication might be needed. This placement is safe for any data layout and any possible computation partitioning for the program’s statements. Second, a placement refinement algorithm is applied that uses information about the data layout and computation partitionings selected to attempt to improve the initial placement.

Placement refinement performs a post-order traversal of the loops in a nest. For each loop, if all communication at the current loop level flows only along the tile iteration data dimension and direction (see Section 5.4), and the loop nest has only multipartitioned computation partitionings within, then all communication is hoisted out one level. dHPF only hoists communication along the tile iteration dimension and direction because vectorizing other communication would increase serialization in a loop with communication in multiple dimensions or directions.

\(^2\)In such cases, loop distribution may be able to eliminate this serialization by converting the original loop nest into multiple loop nests with unidirectional, unidimensional communication that can realize full parallelism.
5.6.2 Formal Properties of Multipartitioned Communication

Lemma 1 (Execution of non-dependent tiles) In a loop nest with unidirectional, unidimensional carried dependences, iterating over block-style partitioned arrays, it is always possible to execute the computation for the subset of data tiles that do not depend on values from other data tiles computed by the loop nest.

Proof: Since dependences flow unidirectionally in a single dimension, there is a subset of tiles that own the subset of data elements whose computation does not depend on previously computed values. This subset of tiles corresponds exactly to the “leftmost” hyperplane, for forward carried dependences, or “rightmost” hyperplane, for backward carried dependences, orthogonal to the dimension along which dependences flow.

Furthermore, the computation of all values belonging to these tiles can be completed without depending on data elements that belong to other tiles (they only depend on tile-local values).

Theorem 2 (Correctness of Vectorized Carried Communication) It is always correct to vectorize unidirectional, unidimensional communication out of a loop nest that references block-style partitioned arrays.

Proof: If the communication does not correspond to carried dependences then principle 14.1 in [AK02] (p. 713) applies and communication can be vectorized completely out of any number of loop levels.

If the communication corresponds to carried dependences then, without loss of generality, it can be assumed to be carried by the innermost loop only (communication is unidirectional and unidimensional). Furthermore, without loss of generality it can be assumed that communication is forward (see definition 1).

These carried dependences are satisfied locally on each tile by executing the computation of a tile in an order which satisfies the dependences. The only problem arises
when a processor boundary has to be crossed. Placing the communication inside the
innermost loop is correct and will satisfy the dependence, as long as the source and
the sink of the dependence execute in a correct relative order.

Given these considerations, we prove that the loop nest executes correctly by
induction on the sequence of hyperplanes of tiles along the dimension carrying the
dependences: $(\Delta_0, \Delta_1, \ldots, \Delta_{n-1})$.

**Base Step:** Lemma 1 guarantees the existence of a subset of tiles that can always
execute, independent of other tiles; this subset can be referred to as $I$. For these tiles,
the placement of communication does not affect the correctness of their execution.
Also, the *computation* of all data values belonging to these tiles can proceed to com-
pletion, independently of the communication placement. Letting $I$ correspond to the
first hyperplane in the processor partition, $\Delta_0$, proves the base case.

**Inductive Step:** Assume that execution of the loop nest is correct with vectorized
carried communication up to and including hyperplane $\Delta_{k-1}$, we must now prove
that execution of $\Delta_k$ will be correct with vectorized carried communication. It is
important to point out that the tiles in $\Delta_k$ directly depend on the values computed
on the tiles in $\Delta_{k-1}$. The tiles in $\Delta_{k-1}$ contain sources for processor-crossing carried
dependences, whose sinks lie in $\Delta_k$.

For tiles in $\Delta_{k-1}$, placing the communication inside the innermost loop causes the
data element computed at the source of the dependence, to be communicated as soon
as it is computed, thus guaranteeing correctness. If the communication is placed on an
enclosing loop, then several data elements computed at the source of the dependence
are “accumulated” without actually communicating them to the tile owning the sink
of the dependence. Communication will then happen when several source elements
have been computed, and the tile owning the elements dependent on these sources (in
$\Delta_k$) receive them as a group and proceed to correctly compute elements dependent
on them. This is correct because the computation of the sources happened before
the computation of the sinks of the dependences, even though various sources were
computed as a group before their respective sinks. In this manner, the computation of all elements belonging to tiles in $\Delta_k$ can complete.

By induction, the loop nest executes correctly for all hyperplanes.

\[\square\]

Corollary 1 (Relative Ordering of Tiles) \textit{In a loop nest with unidirectional, unidimensional carried dependences, data tiles can be processed in any order, as long as the relative ordering of sources and sinks of the dependences is preserved.}

Lemma 2 (Full Parallelism for Multipartitioned Independent Tiles) \textit{In a non-triangular loop nest with unidirectional, unidimensional carried dependences, iterating over multipartitioned arrays, and computing a significant\(^8\) number of data elements in the data domain, all processors will be active in the computation of $I$ (the set of tiles whose computation does not depend on other tiles).}

\textbf{Proof:} Lemma 1 proves that the computation for the independent tile subset $I$ can always be completed. Since the loop nest iterates over a significant number of data elements, in particular it iterates over many elements owned by $I$.

Multipartitioning guarantees that there is a balanced number of tiles assigned to each processor on every hyperplane (see section 5.2.1), in particular there is a balanced number of tiles per processor in the hyperplane corresponding to $I$, full parallelism is guaranteed because $I$ contains all tiles in the initial hyperplane (even though not all data points belonging to $I$ need to be computed).

\[\square\]

Theorem 3 (Full Parallelism for Vectorized Carried Communication) \textit{In a non-triangular loop nest with unidirectional, unidimensional carried dependences, iterating over multipartitioned arrays, and spanning a significant range of the total data domain; all processors will be active over the computation of each hyperplane of data in the loop nest.}

\[^8\]Must include data elements belonging to all processors, but not necessarily every single point in the array.
**Proof:** We prove that the loop nest can be executed in parallel by induction on the sequence of hyperplanes of tiles \((\Delta_0, \Delta_1, \ldots, \Delta_{n-1})\), that have to be traversed to execute the loop nest.

**Base Step:** Since the loop nest spans a significant range of the data domain, it includes elements on every processor. By lemma 2, \( I \) contains tiles belonging to every processor: full parallelism is guaranteed for its computation and it does not depend on any communication for its execution.

Thus, by letting \( \Delta_0 = I \), the base step is proved.

**Inductive Step:** Assume that when executing the set of tiles in \( \Delta_{k-1} \) all processors are active and communication was fully vectorized. The tiles in \( \Delta_k \) depend directly on the tiles in \( \Delta_{k-1} \). We have to prove that when executing the tiles in \( \Delta_k \) all processors will also be active.

The **neighbor** property establishes a one-to-one mapping between processors owning adjacent tiles, thus for every tile in \( \Delta_{k-1} \) there is a corresponding adjacent tile in \( \Delta_k \). Theorem 2 guarantees that it is correct to fully vectorize communication corresponding to unidimensional, unidirectional carried dependences up to the processor boundaries: the partition between hyperplane \( \Delta_{k-1} \) and \( \Delta_k \).

Now, since there is a balanced number of tiles per processor in \( \Delta_{k-1} \) and due to the one-to-one mapping between tiles in \( \Delta_{k-1} \) and \( \Delta_k \), there has to be a balanced number of tiles per processor in \( \Delta_k \) and since execution has completed (in parallel with vectorized carried communication) for all tiles up to and including the ones in \( \Delta_{k-1} \) (inductive hypothesis) then the tiles in \( \Delta_k \) can execute after vectorized communication occurs across the boundary, with all processors active.

By induction, in a loop nest with unidimensional, unidirectional carried dependences iterating over multipartitioned arrays, all processors will be active over the execution of the entire loop nest.

**Theorem 4 (Reduced Parallelism for Multidimensional Dependences)** A loop nest with multidimensional carried dependences (on distributed dimensions), iterat-
ing over multipartitioned arrays cannot be executed in a fully parallel manner while preserving dependences.

**Proof:** Suppose the loop nest is iterating over a $d$-dimensional domain, without loss of generality we can assume that there are *forward* carried dependences on multipartitioned dimensions $i$ and $j$. Assume that it is possible to execute such a loop in parallel while preserving the carried dependences.

Since the tile enumeration order can correspond to only one dimension, assume that tiles are going to be enumerated in *forward* $i$ order. This tile enumeration order induces a partition of the domain into hyperplanes across the $i$ dimension. Assume there are $m$ partitions, $\eta_0, \eta_1, \ldots, \eta_{m-1}$, across this dimension.

Hyperplane $\eta_0$ has $m$ partitions $\mu_0, \mu_1, \ldots, \mu_{m-1}$ across dimension $j$. To preserve the dependences carried on dimension $j$, the tiles in partition $\mu_0$ have to be executed before the tiles in partition $\mu_1$. In general, tiles in partition $\mu_{k-1}$ have to be executed before tiles in partition $\mu_k$.

Now to guarantee full parallelism, *all* tiles in hyperplane $\eta_0$ have to be executed in parallel, which violates the ordering constraint imposed by the dependence carried on the $j$ dimension (all $\mu_k$ partitions will be executed simultaneously).

This contradicts the assumption that a loop nest with multidimensional carried dependences, iterating over a multipartitioned domain, can be executed in parallel while preserving dependences.

\[\square\]

### 5.6.3 Aggregating Communication Across Tiles

A key property of multipartitioning distributions is that a single physical processor owns all of the tiles that are neighbors of a particular processor’s tiles along any given direction. For example, consider the multipartitioning shown in Figure 5.5. The right neighboring tile (if any) of each of processor 1’s tiles along the $y$ dimension belongs to processor 5. Thus, if a processor’s tiles need to shift data to their right neighbor
along a particular dimension, the processor needs to send values to only one other processor.

![Diagram](image)

**Figure 5.13 : Communication Aggregation for multipartitioned tiles**

Figure 5.13 shows vectorized communication between two processors for a multipartitioned array: processor 0 is sending data from its tiles that have a right neighbor (globally), and processor 1 is receiving that data for its tiles that have a left neighbor. Data sent from processor 0’s tiles is packed into a single message and also received as a single message by processor 1 which then unpacks it as necessary.

This property must be exploited for multipartitioned computations to exhibit scalable performance. If not, a separate message would be sent per tile for a message that has been vectorized out of all loops over multipartitioned data with the result that a $d$-dimensional diagonal multipartitioning across $p$ processors would require $O(p^{d+1})$ messages to perform one vectorized communication. Since the number of messages a processor must send using this strategy would grow in proportion to $p$, scalability would be poor.

To avoid this scaling problem, when the dHPF compiler generates code for each fully vectorized communication event, it packs data from all tiles on the owning processor and sends a single message to the corresponding recipient. This optimization
is a major factor in reducing communication frequency, although it doesn’t reduce its
volume.

To support message aggregation across multiple tiles, the dHPF compiler splits
communication tasks on both sending and receiving sides into a number of different
components. Figure 5.14 shows the sequence of component operations executed by
a sender and a receiver for a communication event. It also shows which components
need to be executed for all the tiles a processor owns and which ones are executed as
a single operation.

**Sender’s tasks:**

*Count:* So that it can allocate a message buffer of the appropriate size, the sender
determines the number of elements that must be sent for each tile, and totals the
sizes across all of its tiles. Typically, dHPF generates a closed formula for the send
data per tile from the communication set.

*Allocate:* Allocate a buffer to hold the outgoing message.

*Pack:* Pack the data to be sent for each tile into the message buffer.

*Initiate:* Send the message buffer to the partner processor.

*Wait:* Wait for the message transmission to complete.

**Receiver’s tasks:**

*Count:* The receiver counts the number of elements to be received per tile and totals
the sizes across all of its tiles.

*Allocate:* Allocate a buffer to hold the incoming message.

*Initiate:* Post a non-blocking receive operation.

*Wait:* Wait for message to arrive.

*Unpack:* Empty the communication buffer by scattering the received data into storage
for non-local data (typically, shadow regions) for each tile.
For performance, dHPF uses asynchronous sends and receives. This requires that both the sender and receiver wait for the message transmission to complete separately from initiating the communication. The sender waits for transmission completion as its last operation. The receiver must wait for transmission completion before unpacking its message buffer.

5.6.4 General Communication Scheduling

Low-overhead, high-efficiency communication requires avoiding serialization. Without careful scheduling of communication operations, the impact of serialization is significant. A principal cause of serialization can be the interaction between communication scheduling within each SPMD process and the implementation of non-blocking message-passing communication primitives in the runtime library. According to the MPI standard, the completion of a send operation may be delayed, until a matching receive is posted. Experience using MPI implementations by both IBM and SGI is
that if a non-blocking \texttt{mpi\_irecv} has not been posted before its matching \texttt{mpi\_isend} is initiated, a substantial delay may occur.

To address this issue, dHPF generates code that initiates asynchronous MPI receive operations as soon as possible. In fact, dHPF posts asynchronous receives for loop-carried communication before the dependence-carrying computational loop begins. Posting these receives early helps avoid unnecessary delays when the sender initiates communication. Separating communication into components facilitated reorganizing communication for this purpose.

5.6.5 Carried Communication Scheduling

Section 5.6.1 describes how communication can be vectorized outside of dependence-carrying loops for multipartitioned arrays without reducing the available parallelism. The communication events (see figure 5.14) that support this carried communication will be placed inside the tiling loop together with the computational loop nests. The objective of this special handling is to increase the opportunities for overlapping communication with independent computations and to reduce the overall need for buffer space.

Several of the communication tasks are \textit{independent} of the carried computation they support: i.e., they can be executed in any order with respect to the computational loops, as long as the relative order between the tasks is respected. The \textit{count} and \textit{allocate} operations for both senders and receivers are independent and can be executed in any order with respect to the computational loops that depend on them.

A very important observation is that the \textit{initiate} operation on the receiver is also independent of the computational loops and can be executed \textit{before} them.

Tiling loops can be placed around one or more \textit{compatible}\footnote{Multipartitioned loops are \textit{compatible} if their carried dependences flow in the same direction and dimension or if they do not have any carried dependences} imperfectly nested loop nests. The current implementation in dHPF is restricted to placing tiling loops
around the outermost loop of an imperfect loop nest. The outer loop will potentially contain several loop nests that will execute in the context of the same tiling loop.

\[
\text{do tiles = 1, ntiles < Beginning of tiling loop >}
\text{  < Count events for receivers >}
\text{  < Allocate events for receivers >}
\text{  < Initiate communication for receivers (non-blocking) >}
\text{  do k}
\text{    < Computational loop that does not depend on carried communication >}
\text{    ...
\text{  end do}
\text{  < Other independent computational loops >}
\text{  < Wait for data (receivers) >}
\text{  < Wait for completion of send from previous tile (senders) >}
\text{  < Unpack data if necessary (receivers) >}
\text{  do k}
\text{    < Computational loop dependent on carried communication >}
\text{    ...
\text{  end do}
\text{  < Other dependent loops >}
\text{  < Count events for senders >}
\text{  < Allocate events for senders >}
\text{  < Initiate communication for senders (non-blocking) >}
\text{  do k}
\text{    < Computational loop that does no depend on carried communication >}
\text{    ...
\text{  end do}
\text{  < Other independent computational loops >}
\text{end do < End of tiling loop >}
\]

Figure 5.15: General structure of a tiling loop

Figure 5.15 shows how communication events are placed inside a tiling loop. The initial events for receivers are placed at the beginning of the tiling loop, followed by the issuing of non-blocking receive operations. After these operations, computational loops that do not depend on the expected data can be placed, if there are any. The following operations wait for the reception of the expected data and unpack it from
communication buffers if necessary. At this point, it is possible to conditionally check for the completion of the send operation from the previous tile, if the tiling loop is processing the first tile (in tile enumeration order) then this is a null operation, if this is not the case, then waiting for the completion of the send frees up the send buffer for reuse in the current tile. These operations are followed by computational loops that depend on the received data and that compute new data needed by succeeding tiles. The necessary operations to send this computed data follow the loops that compute it. Other independent loops may be placed after the send operations.

This special schedule for carried communication in tiling loops is intended to increase the possibility of overlapping computation and communication, through the initiation of asynchronous operations and the scheduling of independent computations before the completion of the communication. Also, by scheduling the send and receive operations in such a way, it is possible to reuse a single receive and a single send buffer across multiple tiles in a multipartitioned computation.

Carried Communication Scheduling for Generalized Multipartitioning

For generalized multipartitioning, efficient carried communication requires aggregating the communication for multiple tiles in a single hyperplane. This can be done by taking advantage of the neighbor property guaranteed by generalized multipartitioning: for a particular processor, all adjacent tiles in a particular direction are owned by the same processor. This means that all required data for adjacent tiles can be packed into a single message and delivered to a single processor, using mechanisms analogous to those described in section 5.6.3.

Figure 5.10 illustrates how tile scheduling occurs in generalized multipartitioning, communication should flow along the sweep direction from tiles in the front plane to tiles in the back plane. For example, the front tiles owned by processor 0 (shaded in gray) need to send carried data to tiles on the back plane owned by processor 5 (gray hashed pattern). All data from processor’s 0 front tiles is packed into a single
message, and received as such by processor 5.

5.6.6 Deadlock and Communication Scheduling

As mentioned in section 5.4, there is the possibility of producing deadlock when executing multipartitioned carried communication. This can happen because the sending and receiving sets of tiles in a multipartitioned loop belong to the same processors.

The code generated by dHPF avoids deadlock by selecting computation partitionings that produce only unidirectional carried communication for multipartitioned loop nests, as discussed in section 5.5, and by carefully selecting the static ordering of communication operations, as presented in section 5.6.5.

Figure 5.15 shows the general structure of a tiling loop for multipartitioning. The ordering and types (asynchronous) of the communication operations determine whether it will execute correctly or deadlock. The critical observation is that using asynchronous sends to communicate the recently computed data to the following tiles allows the processor to start executing its next tile (by issuing asynchronous receives) while delaying the completion of the sends until after the corresponding receives for them have executed. This guarantees that the sends will not block (on testing for completion) because the corresponding receives will have already been executed. Asynchronous receives are not required for deadlock-free execution, they are used for overlapping non-dependent computation with the communication operations.

Since carried communication only flows in one direction, then this static scheduling of operations is enough to prevent deadlock. Other possibility to avoid deadlock would be to use multiple send buffers (one per hyperplane) and to check for send completion after the tiling loop has been executed for all tiles. The carefully constructed static schedule dHPF uses does not require more than one send and receive buffer per processor for all hyperplanes, thus possibly enhancing cache utilization.
5.6.7  Tile-based Layout for Multipartitioned Arrays

Storing data for the local portion of any non-BLOCK partitioned array involves choosing a storage layout for multiple data tiles of the same array on each processor. This is the case for cyclically distributed arrays in which every $k$-th element (or every $k$-th group of $n$ elements) is owned by the processor.

The same issue arises to a lesser degree with multipartitioned arrays: each processor owns a set of block-like tiles. Each individual tile can be laid out using standard Fortran column-major ordering, but tiles can be ordered in different ways. Figure 5.16 shows the set of data tiles owned by a particular processor for two different 2D multipartitioned arrays.

The obvious way to lay out multipartitioned data tiles in memory is to assign
contiguous space for all of the tiles for each array. Figure 5.17 shows the tiles for the same two arrays laid out in memory: all tiles for each array are contiguous.

![Diagram of contiguous tiles for arrays](image)

Figure 5.17: Contiguous tiles for arrays

The strategy of laying out tiles in the contiguous-by-array fashion has the potential for cache conflicts between simultaneously used tiles of different arrays. In line sweep computations, tiles of different arrays are used in the same order, i.e. the computation uses the first tile of both arrays, then the second tile of both arrays, etc. If the local portion of an array (sum of the sizes of all tiles) is larger than the size of the cache then the possibility of conflicts between tiles of different arrays that are used together, increases.

dHPF’s solution to this problem is to lay out in memory, the tiles that are used together. That is, lay out contiguously the first tile of each array, then the second tile of each array, etc. Figure 5.18 shows tiles for two different arrays, arranged in memory so that tiles from different arrays, which are commonly used together, are contiguous. This decreases the likelihood of cache conflicts between tiles belonging to different arrays, as well as increases spatial reuse. The same strategy can be applied for programs with more than two arrays.
Arrays A & B: proc. 0

Figure 5.18: Contiguous corresponding tiles
Chapter 6

Optimizing Scalar Code of SPMD Programs

To achieve high performance with code generated by a data-parallel compiler, one must pay attention to all aspects of the generated parallel program. Not only must interprocess communication and synchronization be efficient, but the computation itself on each process must be efficient as well. This chapter describes techniques that dHPF uses to improve the efficiency of each process on an SPMD program execution.

6.1 Integer Set-based Code Generation

Generating SPMD code templates consisting of bounds-reduced loops and conditionals that partition the computation among the available processors for vectors of disjunctive iteration spaces is a complex problem. This is especially true when generating code for iteration spaces that have been partially-replicated along data partitioning boundaries.

While Omega contains a sophisticated procedure for generating code templates for complex iteration spaces [KPR95], it was found that applying Omega’s code generation algorithm to vectors of disjunctive iteration spaces often produced inefficient code templates in which a set of guards terms is repeatedly tested and placeholders for code fragments may repeat many times. Appendix A defines the Omega operations dHPF uses to manipulate integer sets. In dHPF, the following steps are taken to generate high-quality code for complex iteration spaces.

Avoid testing conditions already known to be true. Exploit context information when generating a code template for an iteration space by eliminating any constraints from the iteration space that are universally true, or that the enclosing context makes
true. The intersection of universal known information and the computation partitioning for a scope’s parent is passed as known constraints when generating code for a statement’s iteration space. Omega’s code generation interface accepts one conjunct expressing such known constraints.

Avoid repeatedly testing the same conditions for multiple iteration spaces. To generate code for a set of iteration spaces $I_j, 0 \leq j < n$, first, determine the satisfiability conditions $S_j$ for each iteration space $I_j$ as $S_j = \text{ProjectOnSym}(I_j)$ and compute any satisfiability constraints common to all iteration spaces as $C = \text{Hull}(\bigcup_j S_j)$. If these constraints are non-trivial, simplify each iteration space $I_j$ by removing the common satisfiability constraints to create $I_j'$, where $I_j' = \text{Gist}(I_j, C)$. Next, separately generate code for $C$ and the simplified iteration spaces $I_j'$, for all $j$. Finally, enclose the code generated for all $I_j'$s within a conditional that tests for $C$.

Avoid code replication for disjunctions. When all of the conjuncts in a disjunctive iteration space differ only in satisfiability constraints that are independent of the iteration space range, factor out common satisfiability constraints and enclose the iteration space in a conditional as described in the previous point, then generate a second conditional that tests the disjunction of the remaining satisfiability constraints $D = \bigcup_j \text{Gist}(S_j, C)$ and wrap it around the code generated for the simplified iteration space.

Avoid conditions in guards that repeat tests in loop bounds. Using the strategy described above for computing $S_j$, $S_j$ can contain constraints that will also be enforced by the code generated for the simplified iteration space $I_j$. If $I_j'$ contains a loop with one or more symbolic bounds, then the satisfiability constraints will contain a test comparing the loop’s upper and lower bounds to ensure the iteration set is non-empty. Since an empty loop will not be entered, it is not necessary for the satisfiability conditions to contain this test. The strategy for computing $S_j$ to avoid getting loop bound comparisons as part of the satisfiability conditions was refined. The strategy
for a refined $S_j$, denoted $R_j$ for simplicity, is $R_j = \text{Gist}(S_j, \text{ProjectOnSym}(I_j))$. This computation removes constraints that from $S_j$ that $I_j$ will enforce and creates the simpler satisfiability constraints desired.

6.2 Generating Optimizable Code

The dHPF compiler translates an HPF code into SPMD Fortran code with calls to the MPI message-passing library for communication. Our generated code will achieve high scalar performance only if the Fortran compiler on the target system can generate good object code for the SPMD code produced by dHPF. Obtaining scalar performance with our SPMD code that is competitive with that of the original sequential application proved surprisingly difficult. Below we outline the key problems areas in which we had difficulty and the approaches required to boost SPMD scalar performance to a level closer to the sequential code.

*Use Cray pointer notation for dynamically-allocated arrays.* Rather than linearizing subscript expressions for multi-dimensional data, we found that expressing accesses as multi-dimensional subscripted expressions using Cray pointers exposes such array accesses to the back-end compiler and lets it apply common optimization strategies for them.

*Avoid subscripts containing multiplication.* Array references of the form $a(i - \text{mypid} \times \text{bsize})$ inhibited certain scalar optimizations in our experimental platform. The solution was to store the loop invariant term $\text{mypid} \times \text{bsize}$ in a variable $t\text{.myid}$, so the references would be expressed as $a(i - t\text{.myid})$. This enabled the missing optimizations and simplified the support for load balanced block sizes (see section 4.7). When subscripts contained multiplies, back-end compilers failed to generate memory prefetch instructions. In our generated code, we had to ensure not only that subscripts contained no multiplies, but also that the back-end compiler would not perform forward substitution that violated this constraint. We had to hoist multiplies and wrap them in a call to an identity function that was separately compiled to
achieve the desired effect.

6.3 Overlap Regions vs. Direct-Access Buffers

Overlap regions [Ger90], in which a processor allocates additional storage around the boundary of a data block it owns to store data from neighboring processors, are commonly used by compilers and application developers. The dHPF compiler supports the use of overlap regions for distributed arrays. An HPF2 SHADOW directive specifies the extent of an overlap region for an array on a dimension-by-dimension basis. Figure 6.1 shows HPF source code with an overlap region specified for array a; the SHADOW directive specifies a lower overlap of width one in the second dimension.

    CHPF$ distribute a(*, block) onto P
    CHPF$ shadow a(0, 1:0)
    do j = 2, n
        do i = 1, n
            a(i, j) = a(i, j - 1) + c ! ON_HOME a(i, j)
            end do
        end do
    end do

Figure 6.1: An overlap region on an array in HPF.

Overlap regions are convenient because they enable uniform access to both local and off-processor data, which leads to simple code for partitioned loops. For example, in Figure 6.2 overlap regions make it possible to access a(i, j - 1) for j equal to the processor boundary (my-j_lo).

Figure 6.3 shows a 2D overlap region around a processor’s data tile. Processor 0 is shown shifting a strided rectangular section of data to processor 1. Communication of this data section needs two copies. First, processor 0 packs the data section into a buffer. Next, the receiver (processor 1) unpacks the data from the buffer into
do j = max(2, my_j_lo), min(my_j_lo + j_blocksize, n)
    do i = 1, n
        a(i, j) = a(i, j - 1) + c
    end do
end do

Figure 6.2: SPMD code using an overlap region.

![Overlap Regions](image)

Figure 6.3: Overlap Regions

the overlap region, so its computation can access it in the same manner it accesses on-processor data.

The diagram shows an example with just a right horizontal shift phase, but an application might have a left horizontal shift phase along with bottom and up vertical shift phases, which use the other overlap regions in a similar manner. If we assume a column-major layout, then for each separate phase, the elements in the sections of the overlap region which are not used result in unused lines in the cache.

While overlap regions lead to simple SPMD code, there are three ways that using them can degrade performance. First, any loop accessing an array that has overlap regions allocated, which does not use most of the overlap region in each of the dimensions for which one is provided, suffers from both reduced spatial reuse (values
do j = max(2, my_j_lo), min(my_j_lo + j_blocksize, n)
  do i = 1, n
    if (j - 1 .lt. my_j_lo) then ! REMOTE data
      t1 = buffer_a(i, j - my_j_lo)
    else ! LOCAL data
      t1 = a(i, j - 1)
    end if
    a(i, j) = t1 + c
  end do
end do

Figure 6.4: SPMD code using a direct-access buffer.

in the overlap region may be fetched into cache and not used) and less effective cache utilization (some cache lines may be underutilized because unaccessed overlap regions map to them). Second, if data is received into a message buffer and then copied into overlap regions, there will be two live copies of the data occupying space in the cache. Third, copying the data from a communication buffer into an overlap region can be costly, particularly if the data in the overlap region is non-contiguous.*

To avoid the cache inefficiency that comes from using multi-dimensional overlap regions for single-dimensional shift communication, the dHPF compiler supports accessing remote data directly out of communication buffers. This has the dual advantages of eliminating the unpacking phase on the receiving processor as well as eliminating the need for overlap regions on the receiving processor’s tile. Directly accessing data out of communication buffers, introduces two modes of access for array references: boundary remote data accessed out of the buffer and interior data accessed out of the array.

If the compiler were to generate naive code for data that may reside either in a

*Any data movement in modern machines is costly!
do j = my_j_lo, my_j_lo + 1
    do i = 1, n
        a(i, j) = buffer_a(i, j - my_j_lo) + c
    end do
end do

do j = my_j_lo + 2, min(my_j_lo + j_blocksize, n)
    do i = 1, n
        a(i, j) = a(i - 1, j) + c
    end do
end do

Figure 6.5: Optimized use of a direct-access buffer.

buffer or in a local array, each access would require a conditional test, which would be costly. Figure 6.4 shows naive code for this situation. To avoid the overhead of this approach, the dHPF compiler splits a loop nest that accesses non-local data out of buffers into a loop nest whose iterations may require remote data and those that must access data only from the local array [AMC98].

Loop splitting in this manner eliminates conditionals from the interior section of the loop nest, but may not eliminate conditionals in the non-local section of the loop nest. Figure 6.5 shows a split loop with one iteration space accessing data for array a only out of the local array section, and the other iteration space accessing data out of a buffer. In this case, it was possible to eliminate all conditionals for the non-local reference, because the set of non-local iterations for the statement is a subset of the non-local iterations for the reference. In a more general case, the non-local iteration space may not access exclusively off-processor data and a conditional would be required.
6.3.1 Aggregation and Direct Buffer Access

Direct buffer-access is very useful, especially in combination with communication aggregation. Incoming non-local data for different arrays or disjoint sections of the same array is laid out sequentially in the buffer. dHPF generates code for directly accessing such data by using pointers into each contiguous section in the buffer. With this scheme, the dHPF compiler supports direct access to buffers comprised of unions of constantly-strided rectangular sections.

6.4 Padding for Dynamically-allocated Arrays

Due to its support for symbolic processor counts, dHPF manages storage for each processor using dynamic memory allocation. Each processor allocates a contiguous region of memory for its local portion of the distributed arrays. Since dHPF performs source-to-source compilation, it uses Fortran's standard column-major data layout for the local portions of distributed arrays. Figure 6.6 shows an illustration of the data layout for two 2D data tiles on a processor. The tiles are allocated contiguously in memory, they may belong to the same distributed array or to different arrays, as explained in section 5.6.7.

To minimize intra-array cache conflicts, dHPF's runtime pads each distributed dimension of an array to an odd length. This reduces the possibility of data cache conflicts between columns of the same tile (in general, hyperplanes belonging to the same tile). Padding plus overlap regions that are unused in a particular loop nest increase the number of unused data locations that can occupy address space also mapped to useful data, leading to less effective spatial and temporal reuse; our experiments show that a reduction in conflict misses from padding offsets this cost.
6.5 Memory Allocation for Communication Buffers

An often overlooked aspect of parallel programs is the interaction between memory allocated for distributed data and memory allocated for communication buffers used for message-passing communication. The amount of memory used for communication buffers depends upon:

- the computation to communication ratio of the application,
- the amount of distributed data allocated per process, and
- the number of processes involved in the parallel program

An application’s dependences and their interaction with the partitioning of the application’s data determine how frequently each process communicates with other processes, as well as determining how frequently the off-processor data will be accessed. The data and communication buffers allocated per process determine the
memory footprint of the application data on each node. The distributed data allocated per process should decrease as the number of processes involved in the parallel program increases (if global data sizes are kept constant). However, the memory footprint of the communication buffers becomes more significant as the number of processes increases because of an increase in the communication-to-computation ratio.

It is clear from these properties that careful management of memory allocated to communication buffers is a high priority to achieve high performance and good scalability in an SPMD parallel program, especially with small data sizes and large numbers of processors. For these reasons, it is necessary to minimize the amount of memory used for communication buffers and avoid overheads in its allocation, as well as to maximize cache utilization when accessing them.

A naive buffer management strategy is to allocate memory dynamically for each communication event and release it as soon as the data it holds has been consumed. This has a high overhead due to the cost of calling runtime functions to perform memory allocation and deallocation. It also does not guarantee that the allocated memory for communication events will be mapped effectively into the cache. In particular, the mapped locations of the communication buffers might vary between different procedures of the SPMD application and might interact differently with globally allocated distributed data for each procedure. It does have the advantage of providing the minimum memory footprint for buffers.

To avoid the problems mentioned above, dHPF uses a different buffer management strategy: communication buffers are allocated using a specialized persistent arena [Han90].

The arena structure persists across procedure calls. Communication buffers are allocated out of this persistent space. If there is not enough available space in the current arena, a new one is allocated. At each clean point (when no allocated buffers are live), the allocation pointer moves back to the head of the arena. If more than
one arena exists at a clean point, all arenas are discarded and a single arena large enough to hold all data requested so far is allocated. This leads to efficient memory parceling when the following procedures request allocations. If at some point there is not enough previously allocated memory to satisfy a procedure’s request, then more dynamic memory is requested from the program heap.

This allocation strategy will eliminate the overhead of allocating/deallocating memory for communication buffers, and the potential cache conflicts due to differing mappings for the allocated buffers. Allocating a buffer involves returning a pointer into the arena and advancing a persistent pointer for the next allocation. Deallocating a buffer involves marking it as unused, no actual heap frees occur.

Figure 6.7 shows the state of memory during the execution of two procedures: A and B. Procedure A asks for some buffer space, which on its first execution is dynamically allocated from the program heap. Procedure A then uses this memory and releases it back to the runtime library; at this point, the buffer is not deallocated, but merely marked as unused. When procedure B executes, the same region in the arena can be used to satisfy its request.

For regular programs with an iterative nature, this arena-based scheme reaches a very efficient steady state for buffer allocation. On the first iteration through the procedures that make up the program, the costs of buffer allocation, management and compaction are paid up front. On subsequent iterations, the arena-based manager has all the heap memory pre-allocated in a contiguous block and needs only to keep track of buffer requests and releases.
Figure 6.7: Communication Buffer Reuse
Chapter 7

Experimental Validation

This chapter presents experimental results obtained by compiling several codes with
the dHPF compiler, which applied the optimization techniques described in the pre-
vioius chapters. Each benchmark is described and analyzed in terms of what features
and techniques were necessary to compile it with dHPF and to obtain high perform-
ance from the generated SPMD code.

7.1 NAS SP and BT Application Benchmarks

The NAS SP and BT application benchmarks [BHS+95] are tightly-coupled computa-
tional fluid dynamics codes that use line-sweep computations to perform Alternating
Direction Implicit (ADI) integration to solve discretized versions of the Navier-Stokes
equation in three dimensions. SP solves scalar penta-diagonal systems, while BT
solves block-tridiagonal systems. Both codes perform an iterative computation. In
each time step, the codes have a loosely synchronous phase followed by tightly-coupled
bi-directional sweeps along each of the three spatial dimensions. These codes have
been widely used to evaluate the performance of parallel systems. Sophisticated
hand-coded parallelizations of these codes developed by NASA provide a yardstick
for evaluating the quality of code produced by parallelizing compilers.

The NAS 2.3-serial versions of the SP and BT benchmarks each consists of more
than 3500 lines of Fortran 77 sequential code (including comments). To these, HPF
data layout directives were added, which account for 2% of the line count. To pre-
pare these codes for use with dHPF, several procedures were manually inlined. For
SP, lhsx and minvr were inlined into the source code for procedure xsolve; lhsy
and pinvr into y.solve; as well as lhsz and tzetar into z.solve. For BT, lhsx, lhsy, and lhsz were inlined into x.solve, y.solve, and z.solve respectively. The purpose of this inlining in SP and BT was to enable the dHPF compiler to globally restructure the local computation of these inlined routines so it could be overlapped with line-sweep communication. The inlining was necessary to eliminate a structural difference between the hand-coded MPI and serial versions that would have required interprocedural loop fusion to achieve automatically. In BT, one additional small inlining step was used to expose interprocedurally carried dependences in the sweep computation to avoid having to place communication within the called routine.

7.2 NAS LU Application Benchmark

LU solves the same 3D Navier-Stokes equation as SP and BT. LU implements the solution by using a Successive Over-Relaxation (SSOR) algorithm which splits the operator of the Navier-Stokes equation into a product of lower-triangular and upper-triangular matrices (see [BHS+95] and [FJY98]).

The algorithm solves five coupled nonlinear partial differential equations, on a 3D logically structured grid, using an implicit pseudo-time marching scheme. It is a challenging application to parallelize effectively due to the potential for generating many small messages between processors.

Computationally, the application is structured by computing the elements of the triangular matrices in the subroutines jacld and jacu respectively. The next step is to solve the lower and upper triangular systems, using subroutines blts and buts. After these steps, the variables are updated, a new right-hand side is computed and the process repeats inside a time-step loop.

The routines jacld, jacu, the updating of the variables and the computation of a new right-hand side are completely data-parallel and require only loosely synchronous execution. blts and buts are parallel by 2D hyperplane in a 3D domain; their execution requires the communication of multiple small messages in a pipelined
computation.

The NAS 2.3-serial version of this benchmark consists of approximately 3700 lines of Fortran 77 source code (including comments). We added HPF data layout and other directives, for a total of 136 lines. This application did not require restructuring for compilation with the dHPF compiler. The addition of the HPF directives was enough.

We use a 2D BLOCK distribution with a symbolic partition onto powers-of-two numbers of processors (1, 2, 4, 8, 16, etc.). When the number of processors \( p \) has an integral square root (4, 16, 64, etc.) then each data dimension is partitioned onto exactly the same number of processors: \( \sqrt{p} \). When \( p \) does not have an integral square root, then the first dimension is partitioned into \textit{twice} the number of tiles as the second. Multipartitioning cannot be applied to this problem because the parallelism is not by-line, instead parallel execution happens along a wavefront determined by a diagonal hyperplane on the 3D domain.

The routines \texttt{bits} and \texttt{buts} exhibit \textit{carried} communication on 2D-BLOCK partitioned data. We have implemented a compile-time option in dHPF that enables the programmer to specify that she wants aggressive communication hoisting similar to that applied to multipartitioned loops. In LU this can be done without reducing available parallelism, because the implementation itself is explicitly handling the computation of \texttt{bits} and \texttt{buts} using wavefront parallelism. In general, however, such aggressive placement could reduce parallelism, for this reason, we leave the decision up to the programmer.

### 7.3 NAS MG Kernel Benchmark

MG solves a discrete Poisson problem by using an iterative V-cycle multigrid algorithm on a 3D grid with periodic boundary conditions (see [BHS+95, FJY98, AAC+95]). Each iteration involves evaluating the residual and applying a correction. The V-cycle starts by computing an approximate solution on the finest grid, computing the residual and then projecting it onto increasingly coarser grids. It then
computes an approximate solution on the coarsest grid by smoothing the residual in the \texttt{psinv} routine. This solution is then interpolated onto the finer grids.

After several such steps, the $V$-cycle completes with a final solution on the finest grid. The operators involved in the $V$-cycle computations represent three-dimensional, nearest-neighbor, 27-point stencils.

The NAS 2.3-serial implementation of this benchmark includes 1840 lines of Fortran 77 source code, including programmer's comments. The dHPF implementation adds 83 lines of directives for data partitioning and explicit communication (\texttt{REFLECT} and \texttt{LOCAL}).

One of the initialization routines, \texttt{zran3}, uses an \textit{irregular} reduction to compute the initial values on one of distributed arrays. dHPF does not support analysis and compilation of code that uses irregular accesses to distributed arrays, for this reason we use a modified version of the \texttt{zran3} routine from the NAS-2.3 MPI version of the NAS MG benchmark. The only modifications to the routine were to the parameter layout to make it compatible with the other dHPF-generated routines of MG, the computation and MPI communication inside the routine were not modified at all.

The data layout in the dHPF version of this application differs significantly from the one in the serial and MPI NAS versions. The NAS implementations use a set of large one-dimensional arrays that are allocated in the main program and then reshaped in Fortran 77-style subroutine calls to three-dimensional grid sections. This scheme is not compatible with the HPF language semantics.

For this reason, we added partial support for some Fortran 90 constructs to dHPF:

- Support for user-defined types with pointer components
- Support for Fortran 90 dynamic memory allocation and deallocation
- Support for HPF distribution of dynamically allocated arrays (although dynamically allocated distributed arrays can only be used as procedure arguments)
• Support for HPF alignment of dynamically allocated arrays (same limitation applies)

• Support for Fortran 90 interfaces with assumed-shape arrays

• Support for distributed assumed-shape arrays (only as arguments to a procedure, dHPF does not support ALIGN or DISTRIBUTE directives on interfaces)

The dHPF implementation of MG uses a vector of pointers to distributed 3D arrays to represent each grid level in the multigrid. The length of the vector corresponds to the number of levels. Figure 7.1 illustrates the memory layout for MG: a vector holds pointers to a 3D array for each level in the grid. Level $i$ has a size of $n_i^3$, where $n_i$ is the size of every dimension, level $i - 1$ has a size of $(\frac{n_i}{2})^3$. Each level is dynamically allocated using Fortran 90 memory allocation primitives and distributed using a 3D BLOCK partition. As for the NAS MPI hand-coded version, the number of processors has to be a power of two and is partitioned as evenly as possible among the three dimensions: i.e., 4 processors are distributed in a $1 \times 2 \times 2$ arrangement, 8 processors are distributed in a $2 \times 2 \times 2$ arrangement. In general, processors are distributed as follows:

\[
\begin{align*}
\text{if } (n \mod 3) > 1 & \text{ then } \bar{P} = \left(\left\lfloor \frac{\log_2 n}{3} \right\rfloor, \left\lfloor \frac{\log_2 n}{3} \right\rfloor, \left\lfloor \frac{\log_2 n}{3} \right\rfloor \right) \\
\text{else } \bar{P} & = \left(\left\lfloor \frac{\log_2 n}{3} \right\rfloor, \left\lfloor \frac{\log_2 n}{3} \right\rfloor, \left\lfloor \frac{\log_2 n}{3} \right\rfloor \right)
\end{align*}
\]  

(7.1)  

(7.2)

Where $n$ is the number of processors and $P$ is the 3D processor array.

Before any computation begins in the main program, the processor array is determined at runtime, the distributed data for the grid levels is allocated for each base array and dHPF runtime distribution and alignment descriptors are created for each distributed grid level. After this, procedure calls involving distributed data are handled through the use of Fortran 90 assumed-shape arrays and interfaces for the distributed data. All routines using distributed data deal with only one or two levels.
of the grid at a time, for this reason they do not need to use the vector holding pointers to the grid for each level. The only routine that uses these vectors directly is \texttt{mg3p}, this routine basically functions as a helper to the main program and does not by itself use distributed data.

Routines that deal with only one level of the grid at a time use a natural HPF alignment directive (\texttt{ALIGN array(i,j,k) with template(i,j,k)}) with an appropriately-sized template (the current grid level size is always a parameter for each routine). The interpolation and coarsening routines that deal with two adjacent levels of the grid at the same time, require special handling due to the relation between the levels: both levels can be aligned to a single template with the size of the finest level or each level can be naturally aligned to a template with its own size.

Figure 7.2 shows the HPF directives that specify a single template alignment for two different grid levels. Figure 7.3 shows the HPF directives for separate alignment for each level. In both cases, \texttt{n1}, \texttt{n2} and \texttt{n3} represent the size of the finest grid level and \texttt{m1}, \texttt{m2} and \texttt{m3} represent the size of the coarsest grid level ($m_i = n_i/2$). In the single template case, \texttt{dHPF} is able to generate any necessary communication for both grid levels automatically. In the multiple template case, \texttt{dHPF} cannot cur-
double precision fine_level(n1, n2, n3)
double precision coarse_level(m1, m2, m3)

template templ(n1, n2, n3)

align fine_level(i, j, k) with templ(i, j, k)
align coarse_level(i, j, k) with templ(2*i, 2*j, 2*k)

Figure 7.2: Single template alignment for multiple grid levels

double precision fine_level(n1, n2, n3)
double precision coarse_level(m1, m2, m3)

template finetempl(n1, n2, n3)
template coarsetempl(m1, m2, m3)

align fine_level(i, j, k) with finetempl(i, j, k)
align coarse_level(i, j, k) with coarsetempl(i, j, k)

Figure 7.3: Multiple template alignment for MG grid levels
rently generate communication for expressions involving arrays aligned onto different templates, for this reason communication has to be explicitly implemented using the HPF/JA directives REFLECT and LOCAL.

Currently, the dHPF version of NAS MG is implemented using the two-template approach with explicit communication using REFLECT directives to communicate boundary data.

7.4 SPEC95 tomcatv benchmark

The SPEC95 tomcatv is part of the Standard Performance Evaluation Corporation’s (SPEC) 1995 set of floating point benchmarks. It is a relatively simple benchmark written in approximately 200 lines of Fortran 77 code [Rei95]. The benchmark generates a two-dimensional boundary-fitted coordinate system around general geometric domains. The size of the generated mesh is $514^2$.

The dHPF-parallelized implementation of tomcatv uses two-dimensional multipartitioning to achieve full parallelism with line-style 1D dependences, very similar to those in line sweeps. We distributed seven arrays using the 2D multipartitioning and declared shadow regions as necessary. We enclosed some of the tomcatv loop nests in a surrounding one-trip loop to enable communication coalescing and scheduling for loop nests using the same arrays.

We used the standard benchmark size of $514^2$ for our experiments and we also executed the benchmark with a size of $1028^2$, to compare scalability with a different computation-to-communication ratio.

Communication generation and computation partitioning selection are fully automatic and under dHPF’s control for this benchmark.

7.5 SGI Experimental Framework

The initial experiments were performed on a dedicated SGI Origin 2000 parallel computer with 128 R10000 250MHz processors. Each processor possesses 4MB of L2
cache, 32KB of L1 data cache and 32KB of L1 instruction cache. All Fortran code generated by source-to-source compilation with dHPF along with the reference hand-coded versions of the NAS SP and BT benchmarks (version 2.3) were compiled using the SGI MIPSpro compilers and linked with the SGI MPI native library. In the experiments, SP and BT executions were measured using both class A ($64^3$) and class B ($102^3$) problem sizes.

7.5.1 Performance Comparison

This section provides a comparison of the resulting performance of four different versions of the NAS SP and BT benchmarks:

- NAS SP & BT Fortran 77 MPI hand-coded version, implemented by the NASA Ames Research Laboratory
- NAS SP & BT (multipartitioned) compiled with dHPF from sequential sources
- NAS SP & BT (2D block) compiled with dHPF from sequential sources
- NAS SP & BT (1D block, transpose) compiled with the Portland Group’s pgHPF from their HPF sources

The Portland Group’s (PGI) versions of the NAS SP and BT benchmarks were obtained directly from the PGI WWW site [PGI98].

PGI’s HPF versions of these codes use a 1D BLOCK data distribution and perform full transposes of the principal arrays between the sweep phases of the computation. Since their compiler does not support array redistribution, their implementation uses two copies of two large 4D arrays, where the copies are related by a transpose. The PGI versions of the NAS SP and BT computations were written from scratch to avoid the limitations of the PGI compiler. A discussion of this topic can be found elsewhere [AJMCY98].

These code versions were run on 1–64 processors for the class ’A’ ($64^3$) problem size and 1–81 processors for the larger class ’B’ ($102^3$) problem size. This range of
Figure 7.4: Parallel efficiency for NAS SP (class 'B').

Problem sizes and processor counts is intended to show the performance of different variants of the benchmarks for a range of per-processor data sizes and communication-to-computation ratios.

Figures 7.4 and 7.5 compare the efficiency of four different parallelizations of the NAS SP benchmark for the class 'B' and 'A' problem sizes respectively. Figures 7.6 and 7.7 present the efficiency comparisons for the NAS BT benchmark's class 'B' and 'A' problem sizes.

For each parallelization \( \rho \), the efficiency metric is computed as \( \frac{t_s}{P \times t_p(P, \rho)} \). In this equation, \( t_s \) is the execution time of the original sequential version implemented by the NAS group at the NASA Ames Research Laboratory; \( P \) is the number of processors; \( t_p(P, \rho) \) is the time for the parallel execution on \( P \) processors using parallelization \( \rho \). Using this metric, perfect speedup would appear as a horizontal line at efficiency 1.0. The use of efficiency rather than speedup or execution time as the comparison
Figure 7.5: Parallel efficiency for NAS SP (class 'A').

Figure 7.6: Parallel efficiency for NAS BT (class 'B').
Figure 7.7: Parallel efficiency for NAS BT (class 'A').

metric, enables the accurate gauging of the relative performance of multiple versions across the entire range of processor counts.

The graphs show that the efficiency of the hand-coded MPI-based parallelizations based on a multipartitioned data distribution is excellent, yielding an average parallel efficiency of 1.20 for SP class 'A', 0.94 for SP class 'B', 0.97 for BT class 'A' and 1.02 for BT class 'B'. Thus, the hand-coded versions achieve nearly linear speedup.

The dHPF-generated multipartitioned code achieves similar parallel efficiency and near-linear speedup for most processor counts, demonstrating the effectiveness of dHPF’s compilation and optimization technologies. Its average efficiency across the range of processors is 1.11 for SP class 'A', 0.99 for SP class 'B', 0.94 for BT class 'A' and 1.04 for BT class 'B'. The dHPF compiler achieves this level of performance for code generated for a symbolic number of processors, whereas the hand-coded MPI implementations have the number of processors compiled directly into them.
The remaining gaps between the performance of the dHPF-generated code and that of the hand-coded MPI can be attributed to two factors. First, the dHPF-generated code has a higher secondary cache miss rate on large numbers of processors due to interference between code and data in the unified L2 cache of the Origin 2000. The memory footprint of dHPF’s generated code is substantially larger than that of the hand-coded version due to its generality. Second, on 81 processors the class 'B' benchmarks suffer from load imbalance on tiles along some of the surfaces of multipartitioned arrays. dHPF used a fixed block size (namely, \( \lceil \frac{s}{t} \rceil \), where \( s \) is the extent of the dimension and \( t \) is the number of tiles in that dimension) for tiles along a dimension; using this scheme, tiles at the rightmost end may have fewer elements. For the class 'B' benchmarks on 81 processors, this leads to partially-imbalanced computation with 8 tiles of size 12 and a tile of size 6 at the right boundary. In contrast, the tiling used by the hand-coded MPI ensures each processor has either \( \lfloor \frac{s}{t} \rfloor \) or \( \lceil \frac{s}{t} \rceil \) elements, which leads to more even load balance.

The performance of the dHPF-generated 2D block partitioning using coarse-grain pipelining (CGP) [AJMCY98] is respectable, with an average efficiency of 0.80 for SP class 'A' and 0.65 for class 'B'. Due to a compiler limitation, the CGP version of SP uses overlap regions for storing off-processor data; this places it at a disadvantage with respect to the multipartitioned version, which uses its data directly out of communication buffers. Even if this limitation were eliminated, it would not boost the performance to the level achieved by the multipartitioned codes, due to the inherent serialization in the CGP scheme. Due to a limitation in dHPF’s dependence analysis, it was not possible to obtain results for BT using this partitioning scheme.

The comparison includes the efficiency of the version of the NAS SP and BT benchmarks written by PGI which use 1D block distributions with transposes between sweep phases. These codes, compiled with version 2.4 of the pg/hpf compiler, achieve an average efficiency of 0.69 for SP class 'A', 0.53 for SP class 'B', 0.87 for BT class 'A' and 0.99 for BT class 'B'. Despite good average efficiency for BT, Figures 7.6 and 7.7
show that the PGI version has considerably lower scalability than the dHPF and MPI versions. The full array transposes cause communication volume proportional to the full 3D array volume, whereas communication volume for multipartitioning is proportional to the surface area of the partitioned tiles.

7.5.2 Data Partitioning Support

The previous section showed considerable variations in performance between codes using different partitioning strategies. Here, we quantitatively evaluate the impact of the partitioning strategy on overall execution time and communication volume. The data partitioning strategy employed in a parallel code influences an application’s communication pattern, communication schedule, communication volume and local node performance. The data partitioning choice is a key determinant of an application’s overall performance.

Here, we use the SP and BT benchmarks to compare the impact of three partitioning choices suitable for tightly-coupled line sweep applications. Tables 7.1 and 7.2 present ratios that compare the execution time and communication volume of the 2D-block partitioned versions of the codes and PGI’s 1D-block partitioned versions of the codes with respect to dHPF’s multipartitioned versions. Ratios greater than one indicate higher costs for the partitioning alternatives compared to multipartitioning.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP ‘A’ CGP</td>
<td>1.45</td>
<td>1.23</td>
</tr>
<tr>
<td>Comm. Vol. SP ‘A’ CGP</td>
<td>1.09</td>
<td>1.08</td>
</tr>
<tr>
<td>Time SP ‘B’ CGP</td>
<td>1.76</td>
<td>1.58</td>
</tr>
<tr>
<td>Comm. Vol. SP ‘B’ CGP</td>
<td>1.09</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Table 7.1: 2D-block CGP vs. multipartitioning.

From Table 7.1, we see that the relative increase in communication volume with respect to multipartitioning is modest for both problem sizes. The increase in ex-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP 'A' PGI</td>
<td>1.40</td>
<td>1.61</td>
</tr>
<tr>
<td>Comm. Vol. SP 'A' PGI</td>
<td>3.29</td>
<td>3.27</td>
</tr>
<tr>
<td>Time SP 'B' PGI</td>
<td>1.91</td>
<td>1.98</td>
</tr>
<tr>
<td>Comm. Vol. SP 'B' PGI</td>
<td>4.28</td>
<td>3.28</td>
</tr>
<tr>
<td>Time BT 'A' PGI</td>
<td>0.85</td>
<td>1.34</td>
</tr>
<tr>
<td>Comm. Vol. BT 'A' PGI</td>
<td>3.11</td>
<td>3.16</td>
</tr>
<tr>
<td>Time BT 'B' PGI</td>
<td>0.95</td>
<td>1.69</td>
</tr>
<tr>
<td>Comm. Vol. BT 'B' PGI</td>
<td>4.00</td>
<td>3.13</td>
</tr>
</tbody>
</table>

Table 7.2: 1D-block transpose (PGI) vs. multipartitioning.

execution time is not directly proportional to the increase in communication volume. Previous studies have shown that the principal factor driving the increase in execution time is that the coarse-grain pipelining version incurs more serialization than multipartitioning [CMCS01].

From Table 7.2, it is clear that the communication volume of PGI's 1D-block transpose implementation is a factor of 3–4 larger than that of dHPF's multipartitioned version. For smaller numbers of processors, this strategy is reasonably competitive, but for larger numbers of processors, efficiency degrades. Without more detailed analysis of PGI's generated code, the precise reasons for differences in scalability and performance in response to data and processor scaling are not clear.

### 7.5.3 Partial Computation Replication

Table 7.3 compares the relative performance of versions with and without partial replication of computation. We present the ratios of execution time and communication volume between the dHPF-generated multipartitioned versions of SP and BT without partial computation replication and those with partial replication. Ratios greater than one indicate the cost of not using partial replication. The partially replicated versions use both compiler-derived and explicitly-specified replication using the extended **ON HOME** directive. The results in Table 7.3 show that without partial
replication of computation, both execution time and communication volume increase significantly.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP 'A'</td>
<td>1.18</td>
<td>1.36</td>
</tr>
<tr>
<td>Comm. Vol. SP 'A'</td>
<td>1.33</td>
<td>1.36</td>
</tr>
<tr>
<td>Time SP 'B'</td>
<td>1.12</td>
<td>1.21</td>
</tr>
<tr>
<td>Comm. Vol. SP 'B'</td>
<td>1.33</td>
<td>1.35</td>
</tr>
<tr>
<td>Time BT 'A'</td>
<td>1.35</td>
<td>1.58</td>
</tr>
<tr>
<td>Comm. Vol. BT 'A'</td>
<td>2.94</td>
<td>2.96</td>
</tr>
<tr>
<td>Time BT 'B'</td>
<td>1.13</td>
<td>1.62</td>
</tr>
<tr>
<td>Comm. Vol. BT 'B'</td>
<td>2.97</td>
<td>2.97</td>
</tr>
</tbody>
</table>

Table 7.3: Impact of partial computation replication

This optimization contributes significantly to scalability: its relative improvement in execution time is more significant when communication time is not dominated by computation; for this reason, we observe a higher impact for the smaller class 'A' problem size on a large number of processors. It is worth noting that our experiments were executed on a tightly-coupled parallel computer with a high-bandwidth low-latency interconnect; this reduces the performance penalty associated with higher message volume.

7.5.4 Interprocedural Communication Elimination

We present results on the impact of using the HPF/JA directives to eliminate communication across procedures on the selected benchmarks. Table 7.4 presents ratios comparing the execution time and communication volume of dHPF-generated multi-partitioned versions of SP and BT without using the HPF/JA directives to eliminate communication across procedures with respect to the versions that use them. Ratios greater than one indicate increases in execution time and communication volume. Without the directives, communication volume and execution time are roughly 10–20% higher.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Time SP 'A'</td>
<td>1.05</td>
<td>1.15</td>
</tr>
<tr>
<td>Comm. Vol. SP 'A'</td>
<td>1.12</td>
<td>1.13</td>
</tr>
<tr>
<td>'Time SP 'B'</td>
<td>1.05</td>
<td>1.08</td>
</tr>
<tr>
<td>Comm. Vol. SP 'B'</td>
<td>1.11</td>
<td>1.12</td>
</tr>
<tr>
<td>'Time BT 'A'</td>
<td>1.12</td>
<td>1.09</td>
</tr>
<tr>
<td>Comm. Vol. BT 'A'</td>
<td>1.18</td>
<td>1.18</td>
</tr>
<tr>
<td>'Time BT 'B'</td>
<td>1.06</td>
<td>1.20</td>
</tr>
<tr>
<td>Comm. Vol. BT 'B'</td>
<td>1.18</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Table 7.4: Impact of using the HPF/JA directives to eliminate communication interprocedurally.

### 7.5.5 Normalized Coalescing

Table 7.5 presents the relative performance of codes where coalescing was applied with and without normalization. (Coalescing without normalization may overlook opportunities for eliminating redundancy.) We present the ratios of execution time and communication volume between the dHPF-generated multipartitioned versions of SP and BT with un-normalized coalescing and those with normalized coalescing. Ratios greater than one indicate the overhead of un-normalized coalescing compared to normalized coalescing. The results show that without normalization, both execution time and MPI communication volume increase.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Time SP 'A'</td>
<td>1.41</td>
<td>2.10</td>
</tr>
<tr>
<td>Comm. Vol. SP 'A'</td>
<td>1.70</td>
<td>1.71</td>
</tr>
<tr>
<td>'Time SP 'B'</td>
<td>1.28</td>
<td>1.65</td>
</tr>
<tr>
<td>Comm. Vol. SP 'B'</td>
<td>1.71</td>
<td>1.71</td>
</tr>
<tr>
<td>'Time BT 'A'</td>
<td>1.02</td>
<td>1.24</td>
</tr>
<tr>
<td>Comm. Vol. BT 'A'</td>
<td>1.32</td>
<td>1.32</td>
</tr>
<tr>
<td>'Time BT 'B'</td>
<td>1.04</td>
<td>1.08</td>
</tr>
<tr>
<td>Comm. Vol. BT 'B'</td>
<td>1.32</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 7.5: Impact of normalized coalescing.
We found that the reduction in execution time due to normalized coalescing is mostly due to reduced communication volume in the critical tightly-coupled line sweep routines ($x_\text{solve}$, $y_\text{solve}$, $z_\text{solve}$) of SP and BT. Loosely coupled communication is also reduced, but its impact is not as significant on machines with a high-bandwidth, low-latency interconnect such as the Origin 2000. The greater execution time of the un-normalized version for SP cannot be attributed completely to the extra communication volume, because un-normalized coalescing produces sets which are not rectangular sections; this leads to less efficient management of communicated data, as explained in more detail in Section 6.3.

After both partial replication of computation and normalized coalescing, we found that the communication volume for the dHPF-generated code for SP was within 1% of the volume for the hand-coded versions. For BT, we found that the volume in the dHPF-generated code was actually lower by 5% (class B) and 17% (class A) than that of the hand-coded versions.

### 7.5.6 Multi-variable Aggregation

Table 7.6 presents ratios for execution time and communication frequency that compare the cost of of non-aggregated versions of the codes with respect to aggregated versions. We present the ratios of execution time and communication frequency between the dHPF-generated multipartitioned versions of SP and BT without aggregation and those with aggregation. Ratios larger than one indicate an increase in cost without aggregation. While aggregation reduces message frequency considerably for both SP and BT in all cases, the impact of aggregation on execution time for BT is small because of its high computation to communication ratio. However, the latency reduction that aggregation yields eventually becomes important when scaling a computation to a large enough number of processors so that communication time becomes significant with respect to computation. SP has a higher communication/computation ratio and thus aggregation has a larger impact. For SP, as we drop
back from the larger class 'B' to the smaller class 'A' problem size, and/or increase
the number of processors, the execution time impact of aggregation increases. For
the 64 processor execution of SP using the class 'A' problem size, aggregation cuts
execution time by 10%.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP 'A'</td>
<td>1.02</td>
<td>1.10</td>
</tr>
<tr>
<td>Comm. Freq. SP 'A'</td>
<td>2.12</td>
<td>2.31</td>
</tr>
<tr>
<td>Time SP 'B'</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>Comm. Freq. SP 'B'</td>
<td>2.12</td>
<td>2.31</td>
</tr>
<tr>
<td>Time BT 'A'</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>Comm. Freq. BT 'A'</td>
<td>1.37</td>
<td>1.43</td>
</tr>
<tr>
<td>Time BT 'B'</td>
<td>1.03</td>
<td>1.02</td>
</tr>
<tr>
<td>Comm. Freq. BT 'B'</td>
<td>1.37</td>
<td>1.43</td>
</tr>
</tbody>
</table>

Table 7.6: Impact of aggregation.

### 7.5.7 Processor Set Constraints

Table 7.7 presents results showing the impact of processor constraints on the SP
and BT benchmarks. The table shows the relative performance of versions of the
codes without processor set constraints, compared to the constrained versions. Ra-
tios greater that one indicate the runtime overhead of not using processor constraints.
The metrics presented are execution time, mispredicted branches and L2 instruction
cache misses. These metrics were chosen because additional cache misses and mis-
predicted branches show the runtime effects of code complexity. Without processor
set constraints, the generated code is considerably more complex with many more
conditionals and loops.

Adding processor set constraints has a small impact in execution time; its main
impact is reducing the size of the code and the number of processor loops that have
to be evaluated. This reduces the number of instruction cache misses that have
to be taken as well as the number of mispredicted branches (in some cases). The
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP 'A'</td>
<td>1.01</td>
<td>1.04</td>
</tr>
<tr>
<td>Mis. Branches SP 'A'</td>
<td>0.99</td>
<td>1.01</td>
</tr>
<tr>
<td>L2 I-Cache Miss. SP 'A'</td>
<td>1.16</td>
<td>1.48</td>
</tr>
<tr>
<td>Time SP 'B'</td>
<td>1.03</td>
<td>1.07</td>
</tr>
<tr>
<td>Mis. Branches SP 'B'</td>
<td>0.92</td>
<td>0.99</td>
</tr>
<tr>
<td>L2 I-Cache Miss. SP 'A'</td>
<td>1.05</td>
<td>1.13</td>
</tr>
<tr>
<td>Time BT 'A'</td>
<td>0.98</td>
<td>1.02</td>
</tr>
<tr>
<td>Mis. Branches BT 'A'</td>
<td>1.12</td>
<td>1.10</td>
</tr>
<tr>
<td>L2 I-Cache Miss. SP 'A'</td>
<td>1.09</td>
<td>1.21</td>
</tr>
<tr>
<td>Time BT 'B'</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Mis. Branches BT 'B'</td>
<td>0.93</td>
<td>0.99</td>
</tr>
<tr>
<td>L2 I-Cache Miss. SP 'A'</td>
<td>1.04</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 7.7: Impact of processor set constraints.

The main effect of this optimization is that it speeds up compilation, since adding the processor constraints converts the processing of multidimensional communication sets into processing of single dimensional sets [CMCS01, CMC00].

### 7.5.8 Direct Access Buffers

We compare the efficiency of using direct-access buffers and overlap regions for the selected benchmarks. Table 7.8 presents ratios that compare the execution time and L2 cache misses for versions using overlap regions with respect to versions using direct access buffers. We present the ratios of execution time and L2 data cache misses between the dHPF-generated multipartitioned versions of SP and BT without direct buffer access and those with it. Ratios greater than one indicate the overhead of using overlap regions instead of buffers. Both the execution time and L2 cache misses increase without direct access buffers.

Direct-access buffers play a significant role in reducing L2 data cache misses by avoiding extra copies into shadow regions and reducing the memory footprint of large multidimensional arrays. In particular, they are very important for the tightly-
coupled line sweep phases of the SP and BT benchmarks and the \texttt{lhs} and \texttt{rhs} arrays they use.

7.5.9 Tile-based Layout for Multipartitioned Arrays

Table 7.9 shows the comparative performance of dHPF-generated multipartitioned versions of SP and BT using the regular layout with contiguous tiles for each array, relative to versions using the interleaved tile layout. We present the ratios of execution time and L2 data cache misses. Ratios greater than one indicate the cost of using the contiguous layout compared to the interleaved layout.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP 'A'</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>L2 Misses SP 'A'</td>
<td>1.02</td>
<td>1.01</td>
</tr>
<tr>
<td>Time SP 'B'</td>
<td>1.02</td>
<td>1.00</td>
</tr>
<tr>
<td>L2 Misses SP 'B'</td>
<td>1.01</td>
<td>0.81</td>
</tr>
<tr>
<td>Time BT 'A'</td>
<td>1.04</td>
<td>1.32</td>
</tr>
<tr>
<td>L2 Misses BT 'A'</td>
<td>1.00</td>
<td>1.06</td>
</tr>
<tr>
<td>Time BT 'B'</td>
<td>1.11</td>
<td>1.07</td>
</tr>
<tr>
<td>L2 Misses BT 'B'</td>
<td>1.01</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Table 7.9 : Impact of tile-based for layout for SP and BT.

The table shows that the simultaneously-used tile based layout is a significant
optimization reducing execution time by an average of 7% across all versions. This is due to a reduction of expensive L2 data cache misses. The only case in which it increases run time is for SP class A on 16 processors, but by only 1%. The number of L2 misses for SP class B on 64 processors, is anomalous because an *increase* in L2 misses (using the interleaved layout) did not increase execution time proportionally. The reason behind this anomaly requires more investigation of the behavior of other related metrics (L1 data misses, TLB misses) on this particular configuration.

### 7.5.10 Padding for Dynamically Allocated Arrays

Table 7.10 shows the comparative performance of dHPF-generated multipartitioned versions of SP and BT without padding, relative to versions *with* dynamically allocated array padding. We present the ratios of execution time, L1 data cache misses and L2 data cache misses. Ratios greater than one indicate the cost incurred when not using padding.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>16 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time SP 'A'</td>
<td>1.75</td>
<td>1.02</td>
</tr>
<tr>
<td>L1 Misses SP 'A'</td>
<td>2.65</td>
<td>1.05</td>
</tr>
<tr>
<td>L2 Misses SP 'A'</td>
<td>0.93</td>
<td>0.79</td>
</tr>
<tr>
<td>Time SP 'B'</td>
<td>0.97</td>
<td>0.98</td>
</tr>
<tr>
<td>L1 Misses SP 'B'</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>L2 Misses SP 'B'</td>
<td>0.93</td>
<td>0.91</td>
</tr>
<tr>
<td>Time BT 'A'</td>
<td>0.96</td>
<td>0.95</td>
</tr>
<tr>
<td>L1 Misses BT 'A'</td>
<td>1.02</td>
<td>1.00</td>
</tr>
<tr>
<td>L2 Misses BT 'A'</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>Time BT 'B'</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>L1 Misses BT 'B'</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>L2 Misses BT 'B'</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 7.10: Impact of array padding on SP and BT.

The results in table 7.10 show that padding only significantly decreases execution time for the SP class A benchmark on 16 processors. The improvement comes from
a 165% decrease in L1 data cache misses using padding. Performance improves for SP class A because of the reduction of L1 misses despite the fact that L2 data cache misses increase by 7% for 16 processors and by 26% for 64 processors. The SP benchmark uses data tiles that have one innermost non-distributed dimension; this causes logically contiguous elements in the other dimensions to be farther apart than would be the case if this local dimension did not exist. This increases the potential for cache conflicts between adjacent hyperplanes used in dimensional line sweeps. BT uses an outermost local dimension so it does not exhibit this problem.

For SP class B and BT, the results in table 7.10 show that using padding increases L1 and, in particular, L2 cache misses by having unused elements in the cache lines. This is not offset by a reduction in conflict misses.

Overall, these results show that dHPF needs a more sophisticated algorithm to determine the amount of padding needed for particular arrays instead of the simple heuristic we are currently using. In spite of this, dHPF’s padding produces very good results for SP class A and does not incur heavy overhead in the other cases (slowdowns of no more than 5%), where it is ineffective.

7.5.11 Generalized Multipartitioning

The dHPF compiler for High Performance Fortran supports code generation for generalized multipartitionings (see section 5.2.2).

Communication aggregation is trickier than for diagonal multipartitionings since generalized multipartitionings have multiple tiles per hyper-rectangular slab, but it is possible because generalized multipartitionings also possess the neighbor property described in section 5.2.2. Communication caused by loop-carried dependences should not be performed on a tile-by-tile basis either. Instead, communication should be vectorized for all tiles within a hyper-rectangular slab along the partitioned dimension.

Results from applying generalized multipartitioning in the context of a compiler-based parallelization of the NAS SP computational fluid dynamics application bench-
<table>
<thead>
<tr>
<th># CPUs</th>
<th>hand-coded</th>
<th>dHPF</th>
<th>% diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.95</td>
<td>0.91</td>
<td>3.84</td>
</tr>
<tr>
<td>2</td>
<td>1.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.96</td>
<td>2.93</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>5.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>7.95</td>
<td>8.04</td>
<td>-1.14</td>
</tr>
<tr>
<td>12</td>
<td>11.80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16.64</td>
<td>16.25</td>
<td>2.34</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>18.54</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>19.03</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>22.25</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>24.32</td>
<td>11.38</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>32.22</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>38.83</td>
<td>-0.97</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>39.78</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td></td>
<td>51.49</td>
<td>-6.46</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>47.35</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>59.84</td>
<td>22.02</td>
</tr>
<tr>
<td>72</td>
<td></td>
<td>66.96</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td>70.63</td>
<td>13.23</td>
</tr>
</tbody>
</table>

Table 7.11: Comparison of hand-coded and dHPF speedups for NAS SP (class B).

mark [BHS+95, CMCS01] for the class B problem size of $10^2$ are presented.

These experiments were performed on a SGI Origin 2000 with 128 250MHz R10000 CPUs, each CPU has 32KB of L1 instruction cache, 32KB of L1 data cache and an unified, two-way set associative L2 cache of 4MB.

Table 7.11 compares the performance of the hand-coded MPI version of the SP benchmark with an MPI version generated by the dHPF compiler.* The hand-coded version uses 3D diagonal multipartitioning and thus can only be run on a perfect square number of processors. The dHPF-generated code uses generalized multipartitioning which enables the code to be run on arbitrary numbers of processors. As

*All speedups presented are relative to the original sequential version of the code.
Table 7.11 shows, the performance of the dHPF-generated code is quite close to (and sometimes exceeds) the performance of the hand-coded MPI for numbers of processors that are perfect squares. When the number of processors is a perfect square, the generalized multipartitionings used by the dHPF-generated code are exactly diagonal multipartitionings. These measurements show that the implementation of generalized multipartitionings is efficient in the case of diagonal multipartitionings, in which each processor has one tile per hyperplane of the partitioning. Both the hand-coded and dHPF-generated versions of SP deliver roughly linear speedup on numbers of processors that are perfect squares.

In the measurements taken of the dHPF-generated code for numbers of processors that are not perfect squares, generalized multipartitionings deliver near linear speedup in these cases as well. The cases shown that make use of generalized multipartitioning are ones in which the factors of the number of processors are small primes. Performance would be reduced for numbers of processors that are prime or have large prime factors because computation would be divided into a large number of phases and communication volume grows in proportion to the number of phases. For this experiment, the code generated by dHPF could not exploit generalized multipartitionings when the block size on any processor falls below the shift width associated with communication operations, which happens when a dimension is partitioned many times (as occurs with large primes and prime factors). This limitation prevented experiments with generalized multipartitionings using the 102³ problem size of the SP benchmark on numbers of processors that are large primes or have large prime factors.†

Overall, these experiments show that generalized multipartitionings are of practical as well as theoretical interest and can be used to efficiently parallelize applications using multipartitioning in a wider range of cases.

†To be perfectly clear, this limitation applies only to code generated by the dHPF compiler; the technique of generalized multipartitioning itself is completely general.
7.6 HP Experimental Framework

This set of experiments was performed on a dedicated Hewlett-Packard Alphaserver
cluster with 750 ES45 nodes at the Pittsburgh Supercomputing Center (lemieux@psc.edu).
Each node contains four 1GHz Alpha processors running Tru64 UNIX, with a total
of 4GB of memory. The nodes in the cluster are connected through a Quadrics high-
performance network which possesses remote DMA capabilities.

All Fortran code generated by source-to-source compilation with dHPF along with
the reference hand-coded versions of the benchmarks were compiled using the HP
Alpha Fortran 90 compiler and linked with the Quadrics-adapted MPI library. In
the experiments, the NAS benchmark executions for SP, BT and LU were measured
using class A (64³), class B (102³), and class C (162³) problem sizes. For the NAS
kernel benchmark MG we used the class A (256³, four iterations) and class B (256³,
twenty iterations) problem sizes.

Due to the significant capabilities of the HP cluster, it was possible to execute
experiments using the largest problem size available in the NAS-2.3 benchmarks.
Executing these three benchmarks using these three problem sizes in conjunction
with executing them using up to 100 processors (class 'C'), shows how the dHPF-
generated code behaves as the computation-to-communication ratios change, as well
as showing the different tradeoffs that occur when data sizes per processor change.

7.6.1 NAS SP

The dHPF versions of SP were compiled using all applicable optimizations described
in the previous chapters. These optimizations are enumerated in table 7.12.

Figures 7.8, 7.9 and 7.10 compare the efficiency of the NAS SP class 'A', 'B' and
'C' problem sizes respectively. Each figure presents the efficiencies of both the hand-
coded MPI version and the dHPF-generated version (see section 7.5.1 for a detailed
explanation of how parallel efficiencies are computed).

The average efficiency (across all numbers of processors) for the hand-coded ver-
Figure 7.8: Parallel efficiency for NAS SP (HP, class 'A').

Figure 7.9: Parallel efficiency for NAS SP (HP, class 'B').
<table>
<thead>
<tr>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized coalescing for simple and compact communication sets</td>
</tr>
<tr>
<td>Processor set constraints for simplified processor set enumeration</td>
</tr>
<tr>
<td>Load-balanced multipartitioning support using variable sized tiles</td>
</tr>
<tr>
<td>Carried communication hoisting for multipartitioned arrays</td>
</tr>
<tr>
<td>Special carried communication scheduling for multipartitioned loops</td>
</tr>
<tr>
<td>Communication aggregation for multiple tiles from the same array and also for multiple arrays</td>
</tr>
<tr>
<td>Non-owner computes computation partitionings for partially replicating computation to eliminate communication</td>
</tr>
<tr>
<td>Interprocedural communication elimination through the use of the extended HPF/IA directives LOCAL and extended ON HOME</td>
</tr>
<tr>
<td>Use of direct-access buffers for carried communication in the tightly-coupled line sweep loops</td>
</tr>
<tr>
<td>Tile-based layout for multipartitioned arrays</td>
</tr>
<tr>
<td>Arena-based communication buffer allocation for more effective cache utilization</td>
</tr>
<tr>
<td>Cray pointer representation for distributed arrays</td>
</tr>
</tbody>
</table>

Table 7.12: dHPF optimizations applied to NAS SP

sion of SP class 'A' is 0.97, while the average efficiency of the dHPF-generated version of SP class 'A' is 0.70. For class 'B', the hand-coded average efficiency is 0.94 and the dHPF-generated average efficiency is 0.74. For class 'C', the hand-coded average efficiency decreases to 0.86 and the dHPF-generated average efficiency is 0.73.

The gap between the dHPF-generated efficiency and the hand-coded efficiency is relatively constant across processor counts, indicating that the parallel characteristics of both codes are very similar: both have the same communication volumes, schedules and frequencies and are computing the same volumes of data. It is important to point out that the dHPF-generated runs were carried using the same executable image for each processor configuration, while the hand-coded runs were done using multiple executables, each compiled with a hard-coded processor configuration in it.

The reasons for this constant gap between the efficiencies of the hand-coded and dHPF-generated version is due to conservative compiler assumptions on the part of
the HP Alpha Fortran 90 compiler: careful analysis of the generated assembly code for critical routines within SP revealed that the static number of memory and integer operations within computational loops of a line sweep is very high compared to the assembly code resulting from compiling the MPI hand-coded version. It appears that the Fortran 90 compiler assumes that all Cray pointers might be aliased to each other and to other variables and for this reasons emits extra loads and stores to handle the ambiguity.

The code that dHPF generates is able to handle any number of processors with a single executable image, for this reason it is highly dependent on symbolic variables that hold the number of processors, the tile sizes in each dimensions and the tile starting coordinates for each multipartitioned tile. This large number of critical variables increases the register pressure in loop nests due to the need to keep them available to perform effective loop-invariant array address arithmetic hoisting. This also affects
the scalar efficiency of the dHPF-generated code because loops will certainly have higher register pressure than equivalent loops in the hand-coded version, where most previously mentioned values are compile-time constants.

In comparison to the previously discussed results obtained for the SGI machine, the dHPF-generated efficiencies are lower due to the different back-end compiler used. The dHPF-generated code used in both experiments is very similar, except for the introduction of load-balancing for multipartitionings in the newer sets of experiments on the HP cluster. The relative decrease in efficiency at the high-end of the processor configurations observed in the SGI experiments, has been corrected through the use of the load-balancing partitions: the efficiency curve is approximately constant and follows the same “shape” of the hand-coded efficiency.

7.6.2 NAS BT

The dHPF versions of BT were compiled using all applicable optimizations described in the previous chapters. These optimizations are the same ones that were applied to SP and are enumerated in table 7.12. The main difference is that the use of HPF/JA directives is limited to LOCAL and extended ON HOME for SP, while the dHPF implementation of NAS BT requires the use of Reflect in the routine compute_rhs to guarantee that the shadow regions of the array u are correctly updated before computing the three dimensional sweeps. In SP, the required data is in auxiliary arrays which were computed using partial computation replication in the routine compute_rhs.

Figures 7.11, 7.12 and 7.13 compare the efficiency of the NAS BT class 'A', 'B' and 'C' problem sizes respectively. Each figure presents the efficiencies of both the hand-coded MPI version and the dHPF-generated version.

The average efficiency for the hand-coded version of BT class 'A' is 1.17, which indicates that this code was capable of achieving superlinear speedups. The average efficiency for the dHPF-generated version of BT class 'A' is 0.78. For class 'B',
Figure 7.11: Parallel efficiency for NAS BT (HP, class 'A').

Figure 7.12: Parallel efficiency for NAS BT (HP, class 'B').
Figure 7.13: Parallel efficiency for NAS BT (HP, class 'C').

average efficiencies of 1.02 (hand-coded) and 0.73 (dHPF-generated) were achieved. The hand-coded version of BT class 'C' has an average efficiency of 1.01 and the dHPF-generated version has an average efficiency of 0.80. It is important to note that due to memory constraints on a single node of the HP cluster, it was not possible to run either the NAS2.3-serial version of BT class 'C' or the 1-processor execution of the hand-coded and dHPF-generated versions of BT class 'C'. For this reason, speedups and efficiency were computed with the hand-coded 4-processor version of BT class 'C' as a basis and assuming an speedup of 3.70 for it (this is an extrapolation from the speedups of the hand-coded 4-processor versions of BT class 'A' and 'B').

The lower average efficiencies of the dHPF-generated versions of BT are due to the same reasons discussed for NAS SP. The load-balancing partitions also enable the dHPF-generated code to maintain an almost constant efficiency for all processor configurations, without a decrease at the high-end.
We performed a small experiment to verify how the weaknesses in the back-end HP Fortran 90 compiler affect performance. We hand-modified the `compute_rhs` routine in NAS BT (class 'C') to access its distributed arrays through symbolically-sized formal parameters instead of through global symbolically-sized Cray pointers. We chose this routine because it is the one with the heaviest computation for BT. We linked the hand-modified version of `compute_rhs` with the rest of the dHPF-generated code using the same compiler flags that we used for the Cray pointer version. We ran the resulting code on 36 processors, the overall time decreased from 302 seconds, for the class 'C' Cray pointer-based execution, to 289 seconds, for the same application using formal parameters for the distributed arrays in `compute_rhs`. The main difference is that the HP Fortran 90 compiler generates code that assumes there is no aliasing between the parameters, in contrast to the pointer-based code where it assumes that every pointer can be aliased to any other variable, thus generating slower code.

### 7.6.3 NAS LU

The dHPF versions of LU were compiled using all applicable optimizations described in the previous chapters. These optimizations are enumerated in table 7.13.

Figures 7.14, 7.15 and 7.16 compare the efficiency of the NAS LU class 'A', 'B' and 'C' problem sizes respectively. Each figure presents the efficiencies of both the hand-coded MPI version and the dHPF-generated version.

The average efficiency for the hand-coded version of LU class 'A' is 1.31, due to the extreme superlinearity of its execution on small numbers of processors. The average efficiency for the dHPF-generated version of LU class 'A' is 0.86. For class 'B', average efficiencies of 0.99 (hand-coded) and 0.63 (dHPF-generated) were achieved. The hand-coded version of LU class 'C' has an average efficiency of 1.03 and the dHPF-generated version has an average efficiency of 0.63.

The lower average efficiencies of the dHPF-generated versions of LU are due to the
Figure 7.14: Parallel efficiency for NAS LU (HP, class 'A').

Figure 7.15: Parallel efficiency for NAS LU (HP, class 'B').
<table>
<thead>
<tr>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized coalescing for simple and compact communication sets</td>
</tr>
<tr>
<td>Processor set constraints for simplified processor set enumeration</td>
</tr>
<tr>
<td>Aggressive communication hoisting for BLOCK partitioned arrays</td>
</tr>
<tr>
<td>Communication aggregation for multiple arrays</td>
</tr>
<tr>
<td>Non-owner computes computation partitionings for partially replicating</td>
</tr>
<tr>
<td>computation to eliminate communication</td>
</tr>
<tr>
<td>Interprocedural communication elimination through the use of the extended</td>
</tr>
<tr>
<td>HPF/JA directives <strong>LOCAL</strong>, extended <strong>ON HOME</strong> and <strong>REFLECT</strong></td>
</tr>
<tr>
<td>Arena-based communication buffer allocation for more effective</td>
</tr>
<tr>
<td>cache utilization</td>
</tr>
<tr>
<td>Cray pointer representation for distributed arrays</td>
</tr>
</tbody>
</table>

Table 7.13: dHPF optimizations applied to NAS LU

The wavefront algorithm inherent to the implementation of LU also shows its problems, specially in the smaller problem sizes where the computation-to-communication ratio is low and the partial serialization limits the scalability of the code. For larger problem sizes, such as class 'C', the algorithm is more scalable, at least for the particular processor configurations on which the experiments were conducted. On such problem sizes, the difference in efficiency between the hand-coded version and the dHPF-generated code is approximately the same as that observed for SP and BT, and it is due to the same factors.

Figure 7.17 shows the efficiency of the LU class 'A' implementation on a smaller scale SGI Origin 2000 system with a total of 16 processors. The average efficiency for the hand-coded version is 1.11 and the average efficiency for the dHPF-generated version is 0.97. The efficiency of the 1-processor version of the dHPF code is 0.93, compared to an efficiency of 1.03 for the hand-coded 1-processor version.

These results illustrate the weaknesses in the handling of the dHPF generated code by the HP Alpha Fortran 90 backend compiler as compared to the SGI compiler.
Figure 7.16: Parallel efficiency for NAS LU (HP, class 'C').

Figure 7.17: Parallel efficiency for NAS LU (small scale SGI, class 'A').
### 7.6.4 NAS MG

The dHPF implementations of MG were compiled using all applicable optimizations described in the previous chapters. These optimizations are enumerated in table 7.14.

<table>
<thead>
<tr>
<th>Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized coalescing for simple and compact communication sets</td>
<td></td>
</tr>
<tr>
<td>Processor set constraints for simplified processor set enumeration</td>
<td></td>
</tr>
<tr>
<td>Non-owner computes computation partitionings for partially replicating</td>
<td></td>
</tr>
<tr>
<td>computation to eliminate communication</td>
<td></td>
</tr>
<tr>
<td>Interprocedural communication elimination through the use of the extended</td>
<td></td>
</tr>
<tr>
<td>HPF/JA directives <strong>LOCAL</strong>, extended <strong>ON HOME</strong> and <strong>REFLECT</strong></td>
<td></td>
</tr>
<tr>
<td>Compact &amp; simple communication events through the use of <strong>REFLECT</strong> and</td>
<td></td>
</tr>
<tr>
<td><strong>LOCAL</strong></td>
<td></td>
</tr>
<tr>
<td>Arena-based communication buffer allocation for more effective</td>
<td></td>
</tr>
<tr>
<td>cache utilization</td>
<td></td>
</tr>
<tr>
<td>Fortran 90 assumed-shaped pointer representation for distributed arrays</td>
<td></td>
</tr>
<tr>
<td>Fortran 90 partial user-defined type support</td>
<td></td>
</tr>
<tr>
<td>Fortran 90 interface support for assumed-shape distributed arrays</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.14: dHPF optimizations applied to NAS MG

Figures 7.18 and 7.19 compare the efficiency of the NAS MG class 'A' and 'B' problem sizes respectively. Each figure presents the efficiencies of both the hand-coded MPI version and the dHPF-generated version.

There are several limitations in the current dHPF implementation of NAS MG that degrade its performance compared to the hand-coded MPI version. The main limitations are in the code generation for the interpolation and coarsening routines, which use two adjacent levels of the multigrid simultaneously. Other routines which use a single level at a time can be compiled in a very similar manner as to that of the routines in SP, BT and LU: computation partitioning and communication set construction and generation are relatively similar in complexity to vectorized communication in these other benchmarks, due to the distributed arrays in those routines being aligned naturally to a similar-sized template.
Figure 7.18: Parallel efficiency for NAS MG (HP, class 'A').

Figure 7.19: Parallel efficiency for NAS MG (HP, class 'B').
The interpolation and coarsening routines, on the other hand, *need* to operate on two adjacent levels of the grid simultaneously. As described previously in section 7.3, our implementation uses *two* separate templates with an adequate size for each grid level for the routines rprj3 and interp.

This scheme with two templates causes dHPF to generate efficient-looking code for rprj3, the coarsening routine, which uses data from the fine level of the grid to define new values on the coarse level of the grid. This routine requires loosely synchronous communication for the fine level only. The computation partitionings for its single loop nest depend only on the coarse grid, thus producing simpler code that does not look very different from the code of the routines that deal with a single grid level, such as psinv and resid.

For interp, the two template scheme does not help very much. This routine also only requires loosely synchronous communication, but the computation partitionings required for its loop nests are very complex. The CPs are expressed in terms of the coarse grid, which is being used to define new interpolated points in the fine grid. For example, the expression

\[
u(2*i1-d1,2*i2-d2,2*i3-t3) = \\
u(2*i1-d1,2*i2-d2,2*i3-t3) + 0.5D0 * (z(i1,i2,i3+1) + z(i1,i2,i3))
\]

which computes a point on the fine grid (\(u\)), based on its previous value and an average of adjacent points on the coarse grid (\(z\)). This statement is going to be computed either ON HOME \(u(2*i1-d1,2*i2-d2,2*i3-t3)\) or ON HOME \(z(i1, i2, i3)\). In either case, the relation between the two templates and the two arrays is not known at compile time so the compiler introduces many conditionals that check for possible alignment relations between the two templates and arrays to correctly realize the computation partitioning. These conditionals, by necessity, use integer division which is very costly in our experimental platforms.
The alternative option of using a single template with different alignments for the coarse and fine grids, also does not generate efficient code because of the need to apply the *inverse alignment function* in several places when realizing computation partitionings. The alignment function maps points on an array to points on the distributed template; the inverse alignment function maps points on the distributed template back to points in the array and is used to relate the reduced loop indices, which operate in template space, back to reduced array indices. The single template solution requires an alignment for the coarse grid as presented in figure 7.2, which uses an alignment \( \tilde{A} : (i, j, k) \rightarrow (2i, 2j, 2k) \), that induces an inverse alignment function which also requires many conditional tests, involving integer division, in the current dHPF-generated code.

These two options for generating the code for the interpolation routine are not adequate, since both of them generate highly complex code to realize computation partitionings. In turn, this makes the back-end compilation of this routine very complicated due to the large size of its dHPF-generated source representation (2MB), as well as the deep nesting of conditionals inside computational loops with complex bounds.

The dHPF implementation also uses the HPF/JA-inspired directive `REFLECT` to realize the loosely synchronous communication for `interp` and `rproj3`. Limitations in the analysis for communication involving either imperfectly aligned arrays or two different templates, limit the applicability of the automatic compiler-based communication generation. The expression of communication using only `REFLECT` is not necessarily a good option, because the programmer is responsible for ensuring its correctness and its efficiency (with respect to communicating only needed data and not more).

We executed a 1-processor run of the class 'A' dHPF-generated code for only 1 iteration (instead of the full 4 iterations), and used hardware counters on our small scale SGI Origin, to determine where most of the processor cycles are being spent.
We also ran the NAS2.3-serial class 'A' version of MG under hardware counters for 1 iteration as a comparison point.

The dHPF-generated version spends 57% of its time executing the integer division routine used to check for valid alignments between differently sized arrays and templates in the routine interp. This overhead is not present in the hand-coded serial version and it also accounts for the differences seen between the MPI parallel version and the dHPF-generated version running on more than one processor.

Figure 7.20 shows the efficiency of NAS MG class 'A' running on our small scale Origin. The efficiency of the dHPF-generated version is higher on this version than on the Alpha cluster, due to differences in the back-end compiler handling of the dHPF-generated code, as discussed previously. The scalar overhead due to the use of integer division in the interpolation routine also lowers the efficiency on this platform significantly.
Options for Improving Efficiency

The use of integer division in the interpolation routine is a result of the way dHPF handles code generation for computation partitionings that are either derived from a non-perfectly aligned array or from HOME references which are expressed as constant factors of induction variables: i.e. ON HOME u(2*i1-d1,2*i2-d2,2*i3-t3).

The symbolic block sizes used by dHPF to support single-image execution on different numbers of processors also contribute to create imprecise set representation within dHPF’s core that cause it to generate inefficient code with too many checks.

The solution to these problems requires a change in the way the set representations for computation partitionings and communication sets are constructed and composed inside the dHPF integer-set core. Part of the solution is to introduce external constraints, similar to the processor set constraints described in section 4.3, these constraints reduce the space of possible symbolic members of a particular integer set, and thus eliminate conditions that might have to be checked at runtime.

Another part of the solution is to reduce the cost of the checks themselves. Currently, most of these checks are to verify if a particular symbolic expression or variable is divisible by a constant integer factor. These checks could be replaced by appropriate logical bit operations with a much lower runtime cost.

7.6.5 SPEC95 tomcav

The dHPF versions of tomcav were compiled using all applicable optimizations described in the previous chapters. These optimizations are enumerated in table 7.15.

Figures 7.21 and 7.22 compare the efficiency of the official SPEC95 tomcav problem size and our version with a problem size of 1028², respectively. Each figure presents the parallel efficiency computed with respect to the sequential version of tomcav.

The parallel efficiency for tomcav degrades rapidly for larger number of processors, due to the poor computation-to-communication ratio of this older 2D bench-
Figure 7.21: Parallel efficiency for the SPEC95 tomcatv benchmark (HP, 514^2 (small) size).

Figure 7.22: Parallel efficiency for the SPEC95 tomcatv benchmark (HP, 1028^2 (large) size).
<table>
<thead>
<tr>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized coalescing for simple and compact communication sets</td>
</tr>
<tr>
<td>Processor set constraints for simplified processor set enumeration</td>
</tr>
<tr>
<td>Load-balanced multipartitioning support using variable sized tiles</td>
</tr>
<tr>
<td>Carried communication hoisting for multipartitioned arrays</td>
</tr>
<tr>
<td>Special carried communication scheduling for multipartitioned loops</td>
</tr>
<tr>
<td>Communication aggregation for multiple tiles from the same array</td>
</tr>
<tr>
<td>and also for multiple arrays</td>
</tr>
<tr>
<td>Tile-based layout for multipartitioned arrays</td>
</tr>
<tr>
<td>Arena-based communication buffer allocation for more effective</td>
</tr>
<tr>
<td>cache utilization</td>
</tr>
<tr>
<td>Cray pointer representation for distributed arrays</td>
</tr>
</tbody>
</table>

Table 7.15: dHPF optimizations applied to SPEC95 tomcatv

mark. This benchmark consists of only 200 lines of Fortran code in a single routine and is not necessarily representative of more modern applications, as the NAS benchmarks are.

Pack et al. [PNZ+02] reported similar speedups to our 1028² version on 8 processors of a shared memory Cray T3D, although it is not clear what problem size they were executing.
Chapter 8

Conclusions

This dissertation describes and analyzes the impact of several novel compilation techniques for data-parallel languages such as HPF. The overall objective of this work has been to achieve high performance and high scalability with compiler-generated code, competitive with hand-written code for a broad class of regular applications.

To achieve this objective, several major flaws and weaknesses in HPF compilers’ analysis and code generation were identified: poor scalar performance for each process in an SPMD execution, inefficient communication (poor schedules, unneeded & redundant data), dependence on runtime resolution even for statically analyzable codes, and inadequate data partitionings for tightly-coupled codes.

The research leading to this dissertation included designing several novel compilation techniques and compiler optimizations, as well as adapting well-known techniques to work in a cohesive and integrated manner in the framework of a symbolic data-parallel compiler with support for tightly-coupled codes. These techniques overcome the weaknesses discussed above:

- Support for specialized data partitioning schemes (multipartitioning) for tightly-coupled codes
- Generalization of multipartitioning to support any number of processors
- Communication vectorization for tightly-coupled codes that reduces communication frequency
- Communication scheduling to hide latency, through the overlapping of non-dependent computation and the use of non-blocking primitives
• Advanced static analysis for communication coalescing and simplification

• Advanced static analysis to support effective integer-set based formulation of data-parallel code generation problems

• Efficient communication buffer management for multi-procedure SPMD programs

• A load-balancing scheme for block-style partitionings that enables high-end scalability for large numbers of processors

• Generation of data-parallel executables that can run & scale effectively from one to hundreds of processors

The experimental results presented in chapter 7 validate the thesis of this dissertation: it is possible to generate high performance, scalable code for complex, regular data-parallel applications. Compiler-generated code for the NAS SP, BT and LU application benchmarks exhibited very high performance, closely tracking that of professionally written hand-coded MPI versions of these benchmarks. Large-scale scalability was also achieved: compiler generated code for these applications was executed on up to 100 processors with high parallel efficiency.

These results were achieved using a combination of deep static analysis techniques such as integer-set based manipulations to simplify computation and communication, dependence-based communication placement, value-number based communication coalescing, etc. Another important factor was exposing more control to the programmer through the use of the HPF/JA-based directives, and carefully balancing what is expressed in the generated code and what is done within the dHPF runtime library. Paying attention to all details of an SPMD parallel program was critical to achieve high performance.

However, the results for the NAS MG kernel benchmark show that the techniques designed and implemented in this dissertation, are necessary but not sufficient to
produce high performance, scalable code for all regular data-parallel applications. The low performance of the dHPF-generated version of NAS MG is related to the strided alignments between partitioned grids corresponding to different levels that induce dHPF to use more complex integer sets and mappings than those used for the other benchmarks.

    dHPF relies on the respective platform scalar compiler to generate object code. The experiments conducted in the course of this dissertation show that there is significant difference in the quality of the object code produced by these backend compilers for dHPF-generated sources.

    In general, scalar compilers have some trouble dealing with automatically generated source code and they do not provide enough controls to fine-tune and tailor the object code generation process. These problems explain the relative performance gap remaining between the hand-coded and the compiler-generated benchmarks. The support for a single executable image running on any number of processors has important consequences with respect to the backend “compilability” of dHPF-generated code: loop bounds and array sizes are going to be expressed as unknown symbolic variables and distributed array indexing is more complex. These factors seem to cause problems for the backend compilers in comparison to the compile-time constant-based code for the hand-coded versions of the same benchmarks.

8.1 Future Work

8.1.1 Short & Medium Term Issues

There is still a need to improve and extend the techniques proposed in this thesis to produce higher-performance code for applications that require complex alignments or computation partitionings such as NAS MG. Changes will have to be made to the integer-set based formulations of the dHPF core to better accommodate these kinds of codes. Also, the code generation strategy for sets involving complex alignments will have to be revised to produce lighter weight conditional tests and loop bounds
in the SPMD output code.

There is also the need to produce code that is more amenable to backend compiler optimization: dHPF’s current strategy uses several Fortran constructs that are not handled well by all Fortran-to-machine code compilers, such as Cray pointers and symbolic variables for array sizes and loop bounds. Some of these constructs come from the generality of the code that dHPF produces, i.e. a single parallel application that can run on any number of processors. It might be necessary to restrict such generality (by compiling for a fixed number of processors) in the interest of higher performance; such restriction might also help in the compilation of codes such as NAS MG.

While the NAS application benchmarks are relatively large, they are not full end-user applications: they do not consider I/O operations or interaction with other non-HPF software, also they do not use more modern Fortran 90 features such as modules and interfaces. Such extensions are necessary to deal with production-level applications that users run at scientific computing centers. It is possible to extend dHPF to support many of these features. It would be of particular interest to investigate the necessary techniques to support multi-language and multi-paradigm programming for HPF applications. That is, to provide support for the interaction of HPF procedures and programs with procedures and modules written in other paradigms such as Fortran 90 + MPI, Co-Array Fortran [CDEC03], or C++. This would make the implementation of realistic user applications in HPF more feasible, due to the fact that many of them use such multi-paradigm models.

Another future objective is to investigate the generalized block (GEN_BLOCK) data partitionings incorporated into HPF 2.0 [For97], which enable the application programmer to specify a block-style distribution with blocks of different sizes on the processors. There are significant challenges in analyzing and generating code for such a general data distribution. It should be possible to extend our work on variable-sized blocks for load balance to support the GEN_BLOCK distribution.
8.1.2 Longer Term Issues

Currently, there is widespread dissatisfaction with the state of programming models for parallel computers. The most commonly used programming model, low-level message passing using the MPI library, is complex and requires very careful tuning to achieve high performance. This causes applications developed using a message-passing model to be difficult to maintain and modify. Many high-level design decisions have to be made in the early stages of development and have to be expressed at a low level in the code itself.

There is a clear need for higher-level programming models, such as data-parallel versions of domain-specific languages for mathematics and statistics. There is a clear direct applicability of data-parallel compilation techniques such as the ones presented in this dissertation to such models.

The techniques presented in this dissertation are broadly applicable to problems requiring data-parallel building blocks, such as computational methods that require fault-tolerant, petaflops level computing.

8.2 Final Remarks

This dissertation has improved the applicability of data-parallel HPF compilers to a broader class of applications, as well as to larger, more complex codes. This has been done through the careful application of compilation techniques, optimizations, language extensions and runtime support to produce scalable, high performance compiler-generated codes for data-parallel applications. The methodology employed in this research focused on analyzing the sources of inefficiency on compiler-generated codes and attacking the source of those inefficiencies at the proper level. This provides a plan of action for future research on parallelizing compilers.
Appendix A

Definitions of Omega set operations

- Hull: If \( r_1 \) is a general integer set, \( \text{hull}(r_1) \) computes a single convex set that contains all elements in \( r_1 \), but often other elements as well.

- Gist: If \( r_1 = \{ x_1 \to y_1 \mid f_1(x_1, y_1) \} \) and \( r_2 = \{ x_2 \to y_2 \mid f_2(x_2, y_2) \} \) then the result of applying \( \text{Gist}(r_1, r_2) \) is \( \{ x \leftarrow y \mid f(x, y) \} \). Where \( f \) is a Presburger formula such that \( \forall x, y \ f(x, y) \land f_2(x, y) \iff f_1(x, y) \).

- Project\_On\_Sym: If \( r = \{ x \leftarrow y \mid f(x, y) \} \) then
  \[
  \text{Project\_On\_Sym}(r) = \{ \{ \} \exists x, y \text{ s.t. } f(x, y) \}.
  \]
Appendix B

Dependence Definitions

Definitions taken from [AK02].

- **Data Dependence**: There is a data dependence from statement $S_1$ to statement $S_2$ if and only if (1) both statements access the same memory location and at least one of them stores into it, and (2) there is feasible run-time execution path from $S_1$ to $S_2$.

- **Loop Dependence**: There exists a dependence from statement $S_1$ to statement $S_2$ in a common nest of loops if and only if there exist two iteration vectors $i$ and $j$ for the nest, such that (1) $i < j$ or $i = j$ and there is a path from $S_1$ to $S_2$ in the body of the loop, (2) statement $S_1$ accesses memory location $M$ on iteration $j$, and (3) one of these accesses is a write.

- **Dependence Distance Vectors**: Suppose that there is a dependence from statement $S_1$ on iteration $i$ of a loop nest and statement $S_2$ on iteration $j$; then the dependence distance vector $d(i, j)$ is defined as a vector of length $n$ such that $d(i, j)_k = j_k - i_k$.

- **Dependence Direction Vectors**: Suppose that there is a dependence from statement $S_1$ on iteration $i$ of a loop nest of $n$ loops and statement $S_2$ on iteration $j$; then the dependence direction vector $D(i, j)$ is defined as a vector of length $n$ such that

$$D(i, j)_k = \begin{cases} 
\text{"<"} & \text{if } d(i, j)_k > 0 \\
\text{"="} & \text{if } d(i, j)_k = 0 \\
\text{">"} & \text{if } d(i, j)_k < 0 
\end{cases}$$
- **Loop-Carried Dependence**: Statement $S_2$ has a loop-carried dependence on statement $S_1$ if and only if $S_1$ references location $M$ on iteration $i$, $S_2$ references location $M$ on iteration $j$, and $d(i, j) > 0$ (that is, $D(i, j)$ contains a “<” as its leftmost non-“=” component).

- **Loop-Independent Dependence**: Statement $S_2$ has a loop-independent dependence on statement $S_1$ if and only if there exist two iteration vectors $i$ and $j$ such that (1) statement $S_1$ refers to memory location $M$ on iteration $i$, $S_2$ refers to $M$ on iteration $j$, and $i = j$; and (2) there is a control flow path from $S_1$ to $S_2$ within the iteration.
Bibliography


Albuquerque, NM, June 1993.


[CDFMC01] Daniel Chavarria-Miranda, Alain Darte, Robert Fowler, and John


[CMCS01] Daniel Chavarria-Miranda, John Mellor-Crummey, and Trushar Sarang. Data-parallel compiler support for multipartitioning. In European Conference on Parallel Computing (Euro-Par), Manchester, United King-
dom, August 2001.


[KRC00] M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed-


ACM International Conference on Supercomputing, Barcelona, Spain, June 1995. ACM.


[PNZ+02] Yunheung Paek, Angeles Navarro, Emilio Zapata, Jay Hoeflinger, and David Padua. An advanced compiler framework for non-cache-coherent


