Using Problem Topology in Parallelization

Lorie M. Liebrock

CRPC-TR94477-S
September, 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

PhD. Thesis

RICE UNIVERSITY

Using Problem Topology in Parallelization

by

Lorie M. Liebrock

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

Doctor of Philosophy

Houston, Texas

September, 1994

Abstract

Using Problem Topology in Parallelization

by

Lorie M. Liebrock

Problem topology is the key to efficient parallelization support for partially regular applications. Specifically, problem topology provides the information necessary for automatic data distribution and regular application optimization of partially regular applications. Problem topology is the dimensionality, size, and connectivity of the problem. Problem topology has traditionally been used in explicit parallelization of regular problems such as physical simulation applications. In languages such as High Performance Fortran, problems that are regular allow many optimizations not applicable to irregular application codes. Unfortunately, many applications must sacrifice regularity to some extent for computational efficiency. This research focuses on partially regular problems and strives to take advantage of partial regularity in the parallelization and compilation process.

This dissertation uses topology for automatic, natural-topology, data distribution in linearized and composite grid (or multiblock) applications. For linearized applications, Fortran D is extended with logical topology and index array specifications. With the information provided in these specifications, it is shown how regular, linearized applications can be parallelized automatically in Fortran D as regular computations using their natural topology. In composite grid problems, meshes are coupled to form larger, more complex topology structures. Composite grid problems arise in important application areas such as fluid flow simulation, aerodynamic simulation, electric circuit simulation, and nuclear reactor simulation. Such physical phenomenon are inherently parallel and their simulations are computationally intensive. This dissertation presents a programming style and template for writing High Performance Fortran programs for these applications, algorithms for automatic distribution of composite grid applications with mesh configurations included in the input, and a discussion of compiler issues for composite grid problems. The automatically generated distributions for composite grid applications guarantee that all communication associated with the distribution of any given mesh will remain regular and nearest-neighbor in the mapping to processors. This research allows High Performance Fortran compilers to perform regular application optimizations on the codes for this class of partially regular applications. Finally, the research is supported by experimental results, which indicate that substantial performance improvements are possible when topology is used in the parallelization of partially regular applications.

Acknowledgments

I would like to thank my committee for providing insight and guidance for my research.

Throughout the course of this research, countless people have given feedback on my work. I would like to thank the researchers at Los Alamos National Laboratories, Sandia National Laboratories, Idaho National Engineering Laboratories, Mississippi State University, and the Nuclear Regulatory Commission for the insight they provided into their applications. I would particularly like to thank: Dr. Bharat Soni and Paul Craft at Mississippi State University for supplying the fluid flow and aerodynamic problems used in our validation test suite; George Niederauer and Jim Lime at Los Alamos National Laboratories and Joe Kelly and Frank Odar at the Nuclear Regulatory Commission for providing the nuclear reactor configurations used in our validation test suite; and finally Brenda Helminen at Michigan Technological University for reading many drafts of papers and this dissertation and providing immense help in clarifying and improving the writing of these documents.

Two fellow graduate students and their families were kind enough to let me stay with them during my many week-long trips to Houston. For their generosity and patience with the "house guest", I give special thanks to Cliff and Melanie Click and Jerry and Lisa Roth.

Finally, I thank my husband, Darrell L. Hicks, for his emotional support of my desire to get a Ph.D.

Contents

Abstract ii
Acknowledgments iv
List of Illustrations ix

1 Uses of Problem Topology in Parallelization 1
1.1 Linearized Applications and their Parallelization 2
1.1.1 Linearized Applications 2
1.1.2 Related Work 5
1.2 Composite Grid Applications and their Parallelization 5
1.2.1 Composite Grid Applications 5
1.2.2 Related Work 8
1.3 Overview of Dissertation 10

2 Parallelization of Linearized Applications in Fortran D 12
2.1 Theoretical Motivation 15
2.2 The Fortran D Language and Extensions 18
2.2.1 Logical Dimension Specification 20
2.2.2 Index Array Specification 21
2.3 Compilation 22
2.3.1 Strategy 23
2.3.2 Illustration of Strategy 26
2.3.3 Details of Compilation 26
2.3.4 Analysis 27
2.3.5 Propagation 28
2.3.6 Code Generation 28
2.3.7 Relaxation of Restrictions 31
2.3.8 Optimization for Linearized Array Computations 32
2.4 Experimental Performance Results 33
2.4.1 2-d Direct Linearized Address Computation Application . . . 33

2.4.2 3-d Index Array Application 35
2.4.3 PARTI Parallelization of 3-d Index Array Application 36
2.4.4 3-d Index Array Application Results 37
2.5 Chapter Summary 38

3 Composite Grid Problems 40
3.1 Fluid Flow and Aerodynamic Simulations 40
3.2 Water-Cooled Nuclear Reactor Simulations 42
3.3 Using Problem Topology for ICRM Parallelization 48

4 Mapping Algorithm for Large Mesh Composite Grid Problems 57
4.1 Intuition 57
4.2 Automatic Distribution Algorithm 58
4.2.1 Table Generation 61
4.2.2 Global Minimization 64
4.2.3 Coupling Communication Reduction 65
4.2.4 Limitations -vs- Advantages 67
4.2.5 Fortran D Modification/Generation 69
4.3 Algorithm Validation 73
4.3.1 Simulation of Material Flow in an Elbow with Vanes 73
4.3.2 Simulation of Aerodynamics over Fuselage-Inlet-Nozzle of an F15e Aircraft 75
4.3.3 Simulation of Aerodynamics over a Complete F15e Aircraft . . 76
4.4 Chapter Summary 77

5 Mapping Algorithm for Small Mesh Composite Grid Problems 78
5.1 Intuition 78
5.2 Automatic Distribution Algorithm 78
5.2.1 Random Algorithm 79
5.2.2 Load-Balanced Random Algorithm 80
5.2.3 Topoloby-based Small Mesh Algorithm 80
5.2.4 Limitations -vs- Advantages 84

5.3 Algorithm Validation 84
5.3.1 LOFT 3-d Reactor Model 85
5.3.2 LOFT 1-d Reactor Model 87
5.3.3 H.B. Robinson Reactor Model 87
5.3.4 Westinghouse AP600 Reactor Model 90
5.4 Chapter Summary 90

6 Mapping Algorithm and Precompiler Support for Medium and Mixed Size Mesh Composite Grid Problems 93
6.1 Intuition 93
6.2 Automatic Distribution Algorithm 94
6.2.1 Algorithm Details 96
6.2.2 Limitations 102
6.3 Support for High Performance Fortran Programs 103
6.3.1 Original High Performance Fortran Program 103
6.3.2 Transformation and Final High Performance Fortran Program 103
6.3.3 HPF Compiler Support 121
6.4 Algorithm Validation 125
6.4.1 Simulation of Material Flow in an Elbow with Vanes 127
6.4.2 Simulation of Aerodyanmics over Fuselage-Inlet-Nozzle of an F15e Aircraft 127
6.4.3 Simulation of Aerodyanmics over a Complete F15e Aircraft . . 129
6.4.4 3-d LOFT Reactor Model 129
6.4.5 1-d LOFT Reactor Model 129
6.4.6 H.B. Robinson Reactor Model 133
6.4.7 Westinghouse AP600 Reactor Model 133
6.5 Chapter Summary 133

7 Conclusion 137
7.1 Contributions of the Dissertation 137
7.2 Future Research 138
7.2.1 Future Medium Mesh Algorithm Research 138
7.2.2 Future Small Mesh Algorithm Research 140
7.3 Final Remarks 140

Bibliography **141**

A Modeling Parallel Computation **147**

A.1 Introduction 147
A.2 Assumptions 147
A.3 Notation 147
A.4 The Model 148
A.4.1 The Complete Model 148
A.4.2 Communication 149
A.4.3 Computation 151
A.4.4 Indirection 152
A.4.5 Inspector 152
A.5 Model Validation 153
A.5.1 Test Problem: 1-Dimensional Explicit Material Dynamics . . . 153
A.6 Theorems 155
A.6.1 Standard Model Theorems 155
A.6.2 Extended Model Theorems 158
A.7 Appendix Summary 159

Illustrations

2.1 Mesh divided into slices versus sub-grids for communication. 13
2.2 Fortran D Data Decomposition Specifications 19
2.3 Alignment Statements for Linearized Arrays. 20
2.4 Distribution of 64 element (logical 8x8) array x. 21
2.5 Logical Topology Specification. 22
2.6 Example 1: Linearized Array Computation. 23
2.7 Road Map for Linearized Application Modification (Example 1) . . . 24
2.8 Example 1: Linearized Array Computation. 25
2.9 Compilation Process Outline 27
2.10 3-Dimensional Processor Layout 29
2.11 Communication Optimization Timing Comparison 34
2.12 Example 1 Timing Results 34
2.13 Example 1 Cache Effect Timing Results (Mesh size: (1024x244)/P) . 35
2.14 UTCOMP Timing Results for `disper` 38
2.15 UTCOMP Cache Effect Timing Results (Mesh size: (10x24x24)/P) . 38

3.1 Automatically Generated Grids for Flow through an Elbow 41
3.2 F15e Volume Grid Configuration for Fuselage-Inlet-Nozzle 43
3.3 F15e Volume Grid Configuration for Full Aircraft 44
3.4 Operation counts for all component types. 45
3.5 Component and element counts for LOFT 3-d reactor model. 45
3.6 Component and element counts for the LOFT 1-d reactor model. . . 46
3.7 Component and element counts for H.B. Robinson reactor model. . . 47
3.8 Component and element counts for Westinghouse AP600 reactor model. 47
3.9 Plan view of AP600 model. 48
3.10 Isometric view of AP600 reactor vessel model. 49
3.11 AP600 reactor vessel heat structure model. 50

3.12 AP600 reactor coolant loop 1 model. 51
3.13 AP600 safety systems modeling noding diagram. 52
3.14 Template for composite grid HPF programs. 54

4.1 3-d CFD Multiblock Configuration. 61
4.2 2-d Multiblock Configuration. 61
4.3 3-d CFD Tables 62
4.4 2-d Multiblock Tables 63
4.5 3-d CFD Directions and Starting Processors 67
4.6 2-d Multiblock Directions and Starting Processors 67
4.7 Validation results from flow through an elbow with 2 vanes. 74
4.8 Validation results from F15e Fuselage-Inlet-Nozzle aerodynamic simulation. 75
4.9 Validation results from full F15e aerodynamic simulation. 76

5.1 Pseudocode for mapping mesh M onto processor P 82
5.2 Topology-based Distribution Algorithm for Small Meshes 83
5.3 Validation results for 3-d LOFT reactor model. 86
5.4 Validation results for 1-d LOFT reactor model. 88
5.5 Validation results for H.B. Robinson reactor model. 89
5.6 Validation results for Westinghouse AP600 reactor model. 91
5.7 Validation results for Westinghouse AP600 reactor model. 92

6.1 Medium Mesh Packing Algorithm 98
6.2 Original HPF module and main program. 104
6.3 Original HPF example input routines for all meshes. 105
6.4 Original HPF example allocation routines for all meshes. 106
6.5 Original HPF main subroutine. 107
6.6 Original coupling update insert for the main routine. 108
6.7 Original HPF example compute routines for all meshes. 109
6.8 Original HPF input routine for couplings. 110
6.9 Original HPF example update routine for 3d to 2d couplings. 111
6.10 Transformation steps for composite grid HPF programs. 112
6.11 Transformed HPF module and main program. 113

6.12 Transformed HPF example input routine. 114
6.13 Transformed HPF example allocation routines for 2-d meshes. 115
6.14 Transformed HPF example master allocation routine. 116
6.15 Transformed HPF main_routine example. 117
6.16 Transformed HPF main_routine example continued. 118
6.17 Transformed HPF main_routine example continued. 119
6.18 Transformed HPF main_routine example continued. 120
6.19 Transformed HPF example compute routine for 3-d meshes. 121
6.20 Transformed HPF example update routine for 3d ijk to 2d ijk couplings.122
6.21 Transformed HPF example update routine for 3d ijk to 2d small couplings. 123
6.22 Transformed HPF example update routine for 3d small to 2d small couplings. 124
6.23 Validation results from flow through an elbow with 2 vanes. 127
6.24 Validation results from F15e Fuselage-Inlet-Nozzle aerodynamic simulation. 128
6.25 Validation results from full F15e aerodynamic simulation. 130
6.26 Validation results for 3-D LOFT reactor model. 131
6.27 Validation results for 1-D LOFT reactor model. 132
6.28 Validation results for H.B. Robinson reactor model. 134
6.29 Validation results for Westinghouse AP600 reactor model. 135

A.1 Single Mesh Sample Distributions. 153
A.2 Two Mesh Sample Distributions. 154
A.3 Sub-block distribution mapping 159

Chapter 1

Uses of Problem Topology in Parallelization

Problem topology is the key to efficient parallelization support for partially regular applications. Specifically, problem topology provides the information necessary for automatic data distribution and regular application optimization of partially regular applications. Problem topology consists of the dimensionality, size, and connectivity of the problem. In the simulation of physical phenomenon, the dimensionality and connectivity of the phenomenon are a part of the essence of the problem. Researchers in simulation of specific physical phenomenon use their understanding of the topology of the problem in most, if not all, stages of their research. This is reflected in the algorithms they develop and the programs they write. The topology of problems has traditionally been useful in parallelization of regular problems in that it is often taken advantage of explicitly whenever a physical simulation application is parallelized for a distributed memory machine [HB91]. In regular problems, each problem element is directly dependent on only a fixed set of neighbors. In languages such as Fortran D [FHK+90] and High Performance Fortran [KLS+94], problems that are regular allow many communication optimizations including generation of blocked and parallel communications. Unfortunately, many applications must sacrifice regularity to some extent for computational efficiency. Two examples of such sacrifices are: on vector machines, multi-dimensional arrays have traditionally been linearized to increase the length of vectors; and in complex topology problems, such as the simulation of airflow over an aircraft, many regular meshes are connected together to allow use of efficient algorithms without requiring the simulation of inactive space. Such applications exhibit partial regularity. In the past, problems that were not completely regular were treated as irregular for purposes of parallelization. Irregular treatment has typically led to less efficient execution on parallel machines than regular treatment does. In particular, compile time optimizations such as communication blocking and parallelization of communication can not be applied. Strategies that exploit the partial regularity of linearized and composite grid applications are illustrated in this dissertation. The two primary goals of this research are to reduce the amount of work that a user with a partially regular problem must do to parallelize their application and to take advantage of optimizations developed for regular applications on these partially regular applications. This dissertation shows that significant progress has been made toward both of these goals.

In the linearization research, the regularity of the problem is hidden by the linearization of arrays and the use of index arrays. In this case, topology specification is used to recover the obscured topology of the linearized arrays and distribute them accordingly. In order for the application programmer to use this same kind of regular multi-dimensional distribution, the application would have to be delinearized and have all uses of index arrays removed. Hence the user is required to do less work. Further, communication optimization developed for regular applications with regular distributions can be applied to the resulting program. In Section 1.1, regular linearized applications are introduced, contributions to support regular linearized applications are discussed, and a short summary of related work is presented.

For composite grid research, the problem topology includes the connectivity between the meshes. In this case, problem topology is used to automate distribution of the coupled meshes, which, when combined with the automatic High Performance Fortran (HPF) program transformation procedure, allows regular application optimization. In order for the application programmer to obtain the same results, he/she would have to determine the distribution for each mesh and perform the entire transformation procedure, including generation and modification of the appropriate clones of all of the computation and coupling update routines. Here, again, the user is required to do less work and the compiler can perform the optimizations developed for regular applications with regular distributions. In Section 1.2, regular composite grid applications are introduced, research contributions to support composite grid applications are briefly discussed, and a short summary of related work is presented.

1.1 Linearized Applications and their Parallelization

1.1.1 Linearized Applications

When vector processors came to dominate the supercomputer market in the late seventies and early eighties, vector length was the main factor in determining performance on those machines—longer vectors meant better performance, a rule that holds for these machines even today. To increase the length of vector operations, many programmers "linearized" multi-dimensional arrays in vectorized applications; the multiple dimensions of arrays were collapsed to a single dimension.

Many application codes have been linearized to improve performance on vector processors. Some examples of these codes are UTCOMP and UTCHEM from the University of Texas, and reservoir simulation codes such as VIP-COMP, MORE, and ECLIPSE [Ram93]. Another such linearized code is KIVA, for simulation of chemically reacting flow [Mou93]. All of these applications are computationally intensive and were linearized because supercomputing power was necessary to obtain solutions in a reasonable amount of time.

As recently as last year, linearization was still being performed by hand on supercomputer applications. A global ocean model simulation code at Los Alamos National Laboratories has recently been ported from the Thinking Machines CM-2 to the Cray YMP. This porting was done by linearizing all three of the physical dimensions of the arrays and breaking the long resulting vectors over the eight processors. The researchers found that direct inline computation of linearized addresses was faster than using index arrays [Lub93]. Linearization was a recommended technique for vectorization on the early vector processors such as the CDC Cyber 205 and the Hitachi-S9 with integrated array processor [Wol84]. The Los Alamos porting experience indicates that linearization is still in use for modern vector computers.

This practice of linearization improved performance on vector processors at the cost of obscuring the function of the application code and, in many cases, making efficient parallelization more difficult. In particular, when regular problems are linearized their topology becomes obscured. Languages such as Fortran D and High Performance Fortran do not support linearized applications, with the natural multi-dimensional distribution of data as regular application.

Since most, if not all, supercomputers today are vector processors, the science and engineering community will encounter many linearized arrays when converting these codes to parallel systems. In the long term, the best approach would be to "delinearize" the application before attempting parallelization. This would permit the user to specify a natural multi-dimensional decomposition and allow the compiler to provide the regular application performance advantages on scalable parallel systems. In addition, the user could maintain a single source image if the compiler linearized multi-dimensional arrays when compiling to a vector machine (a fairly straightforward technical problem). Unfortunately, as M. Ramé and U. Kremer learned while attempting to delinearize part of UTCOMP, this is not always an easy task. Often the functionality of the code is obscured during linearization and further obscured during subsequent development of the linearized code. Such was the case with UTCOMP [Ram93]. Consideration of UTCOMP lead to this use of topology for parallelization.

To support applications coded in this linearized style, extensions to Fortran D and compiler technology are developed in this dissertation to handle linearized arrays more efficiently. The extensions are based on the notion of problem topology. In particular, the user is allowed to map linearized arrays to parallel processors according to the logical topology of the problem (logical dimensions of the linearized arrays) rather than the single declared dimension of the linearized arrays. The knowledge of the logical dimensions is used during subscript analysis and communication generation to eliminate the need for runtime interpretation to carry out communication.

This work deals only with the so-called regular applications, for which regular communication stencils exist. In a regular communication stencil, each element is directly dependent only on a fixed subset of neighbors in the problem topology. Hence, the compiler only needs to recognize a limited set of linearized index computation patterns for support of direct index computation. However, many of these applications use index arrays. To reduce (in many cases eliminate) reliance on run-time interpretation for index array processing, specifications of special-case index arrays are also developed. These specifications significantly simplify communication analysis for the special-case index arrays. The combination of topology and index array specifications allow regular application optimizations to be performed in the compiler on linearized applications.

The contributions this dissertation makes in the area of parallelization of linearized regular applications are:

- language extensions for topology and index array specifications,
- technology to support linearized applications with multi-dimensional distributions of linearized arrays as regular,
- experiments to obtain results that validate this approach to handling linearized applications.

Chapter 2 will present the details on this linearized application work. Further, the theoretical results presented in the same chapter help to guide the distribution algorithms that are developed in the rest of the thesis.

1.1.2 Related Work

When this research began, there were only two approaches to providing support for parallelization of linearized applications. In both of these approaches, the user is required to provide all of the data layout specifications. One approach is to treat these problems as regular, but with one parallel dimension. This approach takes no advantage of the problem structure, as there is no way to express the natural data distribution and make communication efficient. Fortran D is an example of the languages that allowed such support. This approach is applicable only when all indices are directly computed and no index arrays are used.

The other approach treats these problems as irregular. This approach takes little advantage of the problem structure, as opposed to regular support, to make communication efficient. Here again the user provides the distribution specifications. With the irregular approach, communication analysis is performed at runtime. The `PARTI` system is an example of such support software [SGCS93]. At the time this work was done, the user had to insert all of the calls to the `PARTI` primitives. A detailed discussion of the extensive application modifications that the user was required to perform is presented in Section 2.4.3.

The approach presented in Chapter 2 supports linearized applications by allowing the programmer to specify their distributions in the natural topology, by requiring only minor application modification, and by facilitating regular application optimization in the compiler.

1.2 Composite Grid Applications and their Parallelization

1.2.1 Composite Grid Applications

In composite grid, irregularly coupled regular mesh (ICRM), or multiblock problems, multiple meshes are connected together to form a larger, more complex topology structure. ICRM problems arise in important and computationally intensive application areas such as fluid flow simulation, aerodynamic simulation, and nuclear reactor simulation. Indeed, many composite grid problems require the use of the fastest computers available, even for simplified simulations. For the solution of the grand challenge simulations in these problems, it is clear that parallelization will be necessary. Again, these simulations are computationally intensive and the phenomena being simulated are inherently parallel. Note that not all of the algorithms for simulating these phenomena are inherently parallel, but the phenomena themselves are. Unfortunately, even when the simulation algorithms are inherently parallel, parallel programming languages such as High Performance Fortran and Fortran D provide little explicit support for these problems.

Early in this research, problems were solicited from scientists working on composite grid applications. The selection criteria for accepting data sets was that the topology and program statistics come from real applications and that the specifications could be obtained in a timely manner. Test problem descriptions come from two different application areas. In the first application area, the meshes are all large with essentially the same computation being performed on each element of each mesh. Three test problems from this application area will be presented in Section 3.1. To illustrate the computational complexity of this class of problems, consider the simulation of HOPE. HOPE [Yam90] is a winged vehicle for space transportation called the H-II Orbiting Plane planned by the National Space Development Agency of Japan. To make accurate simulation of re-entry feasible, hypersonic aerodynamics and aerothermodynamic characteristics must be precisely evaluated. Simulation of one model of a HOPE vehicle provides an indication of the number of grid points used in such computations. In one unsymmetrical calculation of the HOPE 63 model, a total of approximately 9,000,000 grid points were used in the 3-dimensional grids. In many ICRM problems from this application area, the mesh configurations are now generated automatically [SW92, SKC88, SGW88, All88, SE87, Tho87, SG92]. All three of the fluid flow and aerodynamic test problems are composed of meshes that were generated automatically. Without automatic distribution, the application programmer would have to decipher the automatically generated mesh configuration and specify the data distribution explicitly. It is essential that these applications be automatically parallelized when the mesh configurations are generated automatically.

In the second application area, the mesh sizes vary greatly and the amount and type of computation varies greatly between different meshes. Four test problems from this application area are presented in Section 3.2. To illustrate the computational complexity of this class of problems, consider the simulation of the AP600. The AP600 is a new reactor design from Westinghouse. To verify the design criteria of the AP600, that it be capable of unsupervised operation for three days, simulations of many test cases need to be run. Since simulation of the AP600 reactor takes approximately 40 times real time using two communicating workstations (one running RELAP-5 and the other running Contain), each test case takes approximately 120 days to run [Kel94]. In these complex topology problems, the configuration is generated by hand and many iterations of the design and simulation testing steps may be needed to obtain a good model for the system being simulated. This iterative process makes it important to have an automatic approach for distribution. Further, although the topology of reactor simulations is static, the computations are not and the distribution algorithms presented could be reapplied periodically at runtime using dynamically collected statistics to redistribute data as the simulation progresses.

Since the problems that this research focuses on are all partially regular, the algorithms presented are required to preserve the problem regularity in each mesh for which a distribution is generated. The reason for this requirement is that it allows the compiler to take advantage of the vast collection of work that has been done on optimizations for regular distributions of regular problems [HKT91a]. If irregular distributions were generated, there would be much less advantage gained via compiler optimization.

Further, in the case of simple stencils, such as the 3-dimensional seven point stencil and a tori architecture, the description of communication that results from the distributions generated can be strengthened. In this case, the distribution for any single mesh generated by any of the automatic distribution algorithms presented here causes only nearest-neighbor communication. Some people have argued that this is not important as the cost of communicating between distant processors is falling. The problem with this argument is that there is not just a single communication to do in these problems. In general, there is a large number of communications associated with each iteration of the algorithm for any distribution of meshes. By requiring that all communication for a single mesh is regular and nearest-neighbor, no contention is introduced in the network when only the communication associated with a single mesh is considered. There may be contention in the network for communication between different meshes, but there will be none for any single mesh, unless the algorithm used is not a regular nearest-neighbor algorithm.

From an understanding of problem topology, with the requirement that regularity and nearest-neighbor communication be maintained, algorithms are developed for automatically determining data distribution for composite grid problems. This approach allows the compiler to apply all of the available optimization techniques for regular applications to these partially regular applications.

The contributions that this dissertation makes in the area of parallelization of composite grid applications are:

- collection of a test suite of topological descriptions of two classes of composite grid applications;
- design of a template and style recommendations for HPF composite grid applications;
- development of a transformation procedure for converting applications, written in the recommended style using the template, into standard HPF programs that can be optimized as regular application programs;
- design and implementation of algorithms, based on grid size relative to the number of processors in use and problem and architecture topology to automatically distribute data for composite grid applications; and
- execution of experiments to obtain computational load balance and communication measures to validate these algorithms and the overall approach.

Chapters 3–6 present details of this work on parallelization of composite grid applications.

1.2.2 Related Work

Problem topology has been used by computational scientists for parallelization of their applications for many years. For most simulations involving regular geometry and spatially limited interactions, domain decomposition or geometric (topology-based) parallelization is the most efficient and natural approach to parallelization. In their chapter on geometrically parallel algorithms, the parallelization approach most stressed in their book [HB91], Heerman and Burkitt state:

> The method of parallelization that we address in this chapter of geometrically distributing the lattice over the available processing elements is an entirely natural one since the processor array then has the same geometry as the system being simulated. It clearly achieves an efficient load balancing if the system is reasonably homogeneous and has only short-range interactions and if each processor has the same fraction of the volume of the original system.

When this research began there were only two approaches to providing support for parallelization of this general class of applications. In both of those approaches, the user provides all of the data layout specifications. One approach is to treat these problems as irregular. This approach takes little advantage of the problem structure, as opposed to regular support, to make communication efficient. Standard Fortran D is an example of the languages in this category. The other approach was to treat each mesh as a regular computation but do communication analysis at runtime. The `PARTI` system is an example of such support software [SGCS93]. Neither of these approaches save the programmer from having to decipher the grid assembly and determine data distributions. Further, neither of these approches allow the compiler to perform regular problem optimizations on composite grid applications.

Recently, a number of algorithms have been developed that could be used to automatically determine data distributions for composite grids. These approaches are mostly variations on graph partitioning. To list just a few of the approaches that might, possibly with modifications, be used here: simulated annealing, neural networks, genetic algorithms, incremental graph partitioning, recursive single tree bisection, recursive dual tree bisection, greedy algorithms, minimum bandwidth algorithms, inertia algorithms, and spectral partitioning [MF94, ZI93, OR94a, OR94b, FL93, Dag93, PSL90]. Many more references may be found in these papers. There are two major problems with the use of these approaches to automatically distribute data in composite grid applications.

- Graph based approaches are inefficient for most of these applications as each mesh element (of which there are often many thousands) becomes a vertex in the graph. Even with graph contraction applied, these approaches are still expensive.

- More importantly, these approaches generate irregular distributions. This implies that the compiler optimizations developed for regular applications can not be used.

One other support development for composite grid applications must be mentioned. Researchers at IBM's T.J. Watson Research Center are extending IBM's scientific database package to support composite grid application parallelization. These extensions allow the user to specify array partitioning into sub-blocks and the package then provides automatic distribution of array sub-blocks to processors in a bin packing manner. The user is also required to insert calls to database package routines to manage data exchange [CN93]. This approach is somewhat similar to that used in PARTI with the exception of the bin packing distribution support. Using a bin packing approach at runtime to distribution of sub-blocks implies the loss of regularity of communication inside of the meshes and that the compiler can not perform regular optimizations.

The automatic distribution algorithms that will be presented in this dissertation do not determine distribution on an element-wise basis. The automatic distribution algorithm runtimes are primarily functions of the number and dimensionality of the meshes and the number of processors. The one exception to this is discussed in Chapter 6 where the runtime of the linear optimization procedure can increase with the mesh sizes. As for the second point, one of my primary goals was to be able to generate data distributions and a program that can be optimized as a regular application at compile time. I have achieved that goal.

The one type of ICRM problem for which automatic distribution has been explored is multigrid. In multigrid computations, fine granularity meshes are placed over subgrids of larger granularity meshes to refine the computation in areas of interest. Thuné has been working on automatic distribution of data structures for multigrid computations [Thu93].

1.3 Overview of Dissertation

Chapter 2 begins with extensions to Fortran D to allow the use of problem topology in parallelization of linearized applications. Following that, the specification of problem topology is used to support the natural topology parallelization of applications with linearized arrays and the use of index arrays.

Chapter 3 presents an introduction to the test problems used in the composite grid application parallelization research. That chapter also provides HPF programming style recommendations and a composite grid application template.

Chapters 4–6 present algorithms to map regular ICRM problems onto parallel computers. Chapter 4 develops an algorithm for automatic data distribution of composite grid problems that have only large meshes. Chapter 4 also discusses automatic modification/generation of Fortran D for this class of problems. Chapter 5 develops algorithms for automatic data distribution in the presence of small meshes. Chapter 6 develops an algorithm for automatic data distribution in the presence of arbitrary (mixed) size meshes. Each of these chapters presents the experimental results obtained by applying the algorithms to some of the test problems. Discussion of compiler issues, discussion of HPF program transformation, and presentation of transformation of examples are defered until Chapter 6. Therefore, Chapter 6 will also: 1) outline transformation of HPF composite grid applications, 2) discuss the HPF compiler technology needed to support the transformed programs, and 3) show excerpts from abstracted applications (written in HPF) before and after transformation.

Chapter 7 concludes the main part of this dissertation with a review of the contributions of the dissertation and a discussion of intended future research.

Appendix A presents the details of the model for parallel computation that was used throughout this research. The model and related theorems provide insight that was applied throughout this research.

Chapter 2

Parallelization of Linearized Applications in Fortran D

Users would like porting to parallel processors to be easy but the result must provide efficient execution to be acceptable. Some applications programmers are considering Fortran D [FHK+90] or similar languages for this conversion. If all of the linearized address computations are done directly (via inline computation such as A(i)=f(A(i+ni))) then the code *can* be parallelized using a 1-dimensional data distribution in Fortran D. However, this might not provide the best performance, as will be illustrated with an example. It is well known [FO84, RAP87] that the computation to communication ratio is one of the most important numbers there is in determining the efficacy of a parallelization. For this reason, the relationship between parallelization topology and the computation to communication ratio for two parallelization topologies, the natural problem topology and a 1-dimensional topology, will be examined.

Define a standard computation/communication stencil to be one in which any given element is dependent only on the element before it and the element after it in each dimension (the 2-dimensional stencil is a five point stencil and the 3-dimensional stencil is a seven point stencil). Now, consider a regular application with large linearized meshes that uses a standard computation/communication stencil. When parallelized in a 1-dimensional fashion, as required for regular support in Fortran D, the same size boundary must be exchanged no matter how many processors are used (with fewer processors than columns). If the same application is parallelized in the natural topology of the problem, the size of the boundaries to be exchanged decreases with increasing number of processors. See Figure 2.1 for a graphical illustration of this well known phenomenon. Since the communication per processor decreases as the computation per processor decreases instead of remaining fixed as it does in the 1-dimensional parallelization, the multi-dimensional parallelization is preferable. For a more precise treatment of the merits of natural topology parallelization, see

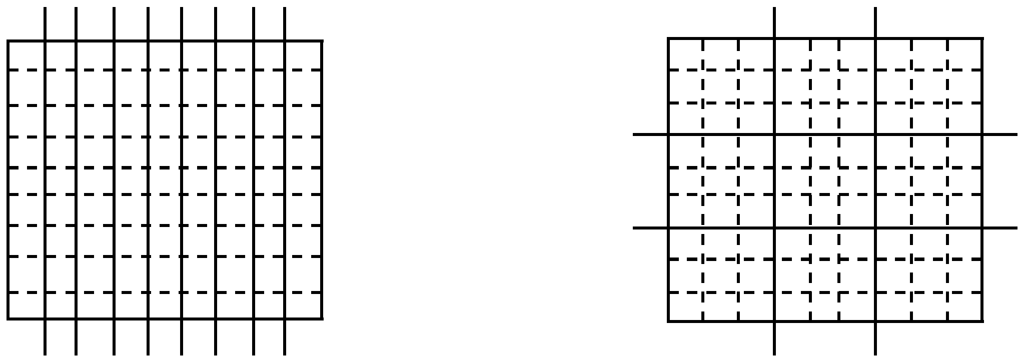


Figure 2.1 Mesh divided into slices versus sub-grids for communication.

Section 2.1. The experimental results in Section 2.4 for both examples agree with these theoretical results.

Thus, it is established that using 1-dimensional data distributions on a problem with a natural multi-dimensional topology is not a good idea. Unfortunately, there is no way in the original Fortran D or in HPF to distribute a linearized array according to its natural topology without treating the computation as an irregular one. With the current definition of Fortran D, the computation can either be distributed one dimensionally and treated as a regular computation or distributed according to a user-defined function and processed via an inspector/executor approach, which requires that communication be determined at run time. Hence, you lose efficiency by not using the proper topology for distribution or you lose efficiency by paying the overhead associated with handling irregular problems.

This loss of efficiency is overcome by extending Fortran D to support parallelization according to the natural problem topology in the presence of linearized arrays. This extension involves two new statements. The LOGICAL DIMENSIONS statement provides information that will allow the Fortran D compiler to optimize the natural topology parallelization as a regular computation without the user having to "delinearize" the arrays in the application. In particular, the LOGICAL DIMENSIONS statement provides the topology description, which is sufficient to determine, at compile time, what communication is necessary when linearized array index computations are computed inline.

The second extension is designed to help optimize accesses via index arrays that are often used in conjunction with linearization. When index arrays are used, the Fortran D compiler cannot compile the code as regular. As the data reference pattern is not analyzable at compile time, the only support for this case is via an inspector/executor strategy [vKK+92, BSS90]. There are a number of problems with this approach. To begin with, current compilers cannot yet generate the inspector/executor automatically. Hence, this conversion must be done by hand. To do this, the user must modify declarations, determine the new loop bounds, figure out how each processor computes the local portion of the index arrays (in global terms) and finally insert calls to localize the index arrays and calls to perform communication. If double indirection is used, then the user must be careful to gather the nonlocal values of the appropriate index array(s) before they are localized (including the newly gathered values). If values from index arrays are used in comparisons, then the user must save a copy of the global version of the index array and replace all uses of index values in comparisons with uses of values from the global version.

A new version of the Fortran D compiler is under development that will be able to generate the inspector and executor automatically [Sal93]. Even if the complete inspector/executor can be generated automatically, there is still extra execution time associated with the resulting parallelization. Communication requirements must be determined at runtime and communication satisfying those requirements must be generated. The overhead associated with determining the communication pattern is proportional to the number of index arrays present. Since the applications under consideration are regular calculations when considered in the natural topology, this is an unnecessary overhead.

To maximize the efficiency of execution of these linearized applications, INDEX ARRAY specifications are proposed. The INDEX ARRAY statement specifies the relationship between indices of arrays and their values. This allows compile time analysis of indirections for optimization as a regular application. Experimental results in Section 2.4.4 for Example 2 compare the performance of using this new approach to using a block-structured version of the `PARTI` inspector/executor approach.

Before details of the Fortran D extensions and their use are presented, theoretical motivations are developed, which will be used throughout the rest of the dissertation. After the extensions are presented, use of the extensions to allow the compiler to support multi-dimensional distributions, and thereby provide more efficient parallelization, will be presented.

2.1 Theoretical Motivation

Here I consider applications with standard computation/communication stencils and prove some theorems that motivate my approach to distribution. These results can be extended for other symmetric stencils.

First, a result is proven for applications with large 2-dimensional meshes using many processors.

> **Theorem 2.1** Let $P = p_i * p_j$ be the number of processors with $p_i, p_j >> 1$. Let the size of the application mesh be m-by-n ($m \leq n$) where $m >> \frac{n}{P}$, P divides m and n, and $\frac{m}{p_i} = \frac{n}{p_j}$. If the application uses a standard 2-dimensional stencil then a sub-block (2-dimensional) partition of the problem onto the processors will yield less communication than a linear (1-dimensional) partitioning.

Proof: Let α be the number of computations per element and β be the number of communications (sends) per element in each direction of each dimension.

The minimum cuts possible with a linear partitioning is $m(P-1)$ and is achieved by cutting into P slices of shape m-by-$\frac{n}{P}$. Since each cut requires two elements to perform sends in one direction of the dimension, the total communication is $C_{LP} = 2\beta m(P-1)$. The total computation is αmn. Hence the communication to computation ratio for linear partitioning is:

$$\frac{2\beta m(P-1)}{\alpha mn} = \frac{2\beta(P-1)}{\alpha n}$$

The minimum cuts possible with a sub-block partitioning is $m(p_j-1)+n(p_i-1)$ and is achieved by cutting into P sub-blocks of shape $\frac{m}{p_i}$-by-$\frac{n}{p_j}$. Since each of the vertical cuts requires two elements to perform sends in one direction of the second dimension, the total vertical communication is $2\beta m(p_j - 1)$. Since each of the horizontal cuts requires two elements to perform sends in one direction of the first dimension, the total horizontal communication is $2\beta n(p_i - 1)$. Hence, the total communication is $C_{SP} = 2\beta(m(p_j-1) + n(p_i-1))$. The total computation is αmn. Hence the communication to computation ratio for sub-block partitioning is:

$$\frac{2\beta(m(p_j-1)+n(p_i-1))}{\alpha mn}$$

This leads to a communication ratio of

$$\frac{C_{LP}}{C_{SP}} = \frac{m(P-1)}{m(p_j-1)+n(p_i-1)},$$

which can be reduced to

$$\frac{C_{LP}}{C_{SP}} = \frac{p_i - \frac{1}{p_j}}{2 - \frac{1}{p_j} - \frac{1}{p_i}}. \tag{2.1}$$

Since both p_i and p_j are much larger than one, the communication to computation ratio is worse for the linear partitioning than for the sub-block partitioning. $\square$

Although the result of Theorem 2.1 is quite general, it may not be obvious to the casual reader just how bad the linear partitioning can be relative to the sub-block partitioning. Corollary 2.1 makes this clear.

Corollary 2.1 If $m = n$ in Theorem 2.1 then the communication ratio is:

$$\frac{C_{LP}}{C_{SP}} = \frac{p_i+1}{2} = \frac{p_j+1}{2} = \frac{\sqrt{P}+1}{2}$$

Proof: Follows from Theorem 2.1.

Next, a similar result is proved for the 3-dimensional case.

Theorem 2.2 Let $P = p_i * p_j * p_k$ be the number of processors with $p_i, p_j, p_k >> 1$. Let the size of the application mesh be l-by-m-by-n ($l \leq m \leq n$)) where P divides l and m and n, and $\frac{l}{p_i} = \frac{m}{p_j} = \frac{n}{p_k}$. If the application uses a standard 3-dimensional stencil then a sub-cube (3-dimensional) partition of the problem onto the processors will yield less communication than a linear partition.

Proof: Let α be the number of computations per element and β be the number of communications (sends) per element in each direction of each dimension.

The minimum cuts possible with a linear partitioning is $lm(P-1)$ and is achieved by cutting into P slices of shape l-by-m-by-$\frac{n}{P}$. Since each cut requires two elements to perform sends in one direction of the dimension, the total communication is $C_{LP} = 2\beta lm(P-1)$. The total computation is αlmn. Hence the communication to computation ratio for linear partitioning is:

$$\frac{2\beta lm(P-1)}{\alpha lmn} = \frac{2\beta(P-1)}{\alpha n}$$

The minimum cuts possible with a sub-cube partitioning is $mn(p_i-1)+ln(p_j-1)+lm(p_k-1)$ and is achieved by cutting into P sub-cubes of shape $\frac{l}{p_i}$-by-$\frac{m}{p_j}$-by-$\frac{n}{p_k}$. Since each of the horizontal cuts requires two elements to perform sends in one direction of the first dimension, the total horizontal communication is $2\beta mn(p_i-1)$. Since each of the vertical cuts requires two elements to perform sends in one direction of the second dimension, the total vertical communication is $2\beta ln(p_j-1)$. Since each of the level cuts requires two elements to perform sends in one direction of the third dimension, the total level communication is $2\beta lm(p_k-1)$. Hence, the total communication is $C_{SP} = 2\beta(mn(p_i-1)+ln(p_j-1)+lm(p_k-1))$. The total computation is αlmn. Hence the communication to computation ratio for sub-cube partitioning is:

$$\frac{2\beta(mn(p_i-1)+ln(p_j-1)+lm(p_k-1))}{\alpha lmn}$$

Then the communication ratio is

$$\frac{C_{LP}}{C_{SP}} = \frac{lm(P-1)}{(mn(p_i-1)+ln(p_j-1)+lm(p_k-1))},$$

which can be reduced to

$$\frac{C_{LP}}{C_{SP}} = \frac{p_i(1-\frac{1}{P})}{\frac{2-\frac{1}{p_j}-\frac{1}{p_k}}{p_i}+\frac{p_i-1}{p_j^2}} \tag{2.2}$$

Since the denominator is less than one and p_i is much larger than one, the communication to computation ratio is much worse for the linear partitioning than for the sub-cube partioning. $\square$

Although the result of Theorem 2.2 is quite general, it may not be obvious just how bad the linear partitioning can be relative to the sub-block partitioning. Corollary 2.2 makes this clear.

Corollary 2.2 If $l = m = n$ in Theorem 2.2 then the communication ratio is:

$$\frac{C_{LP}}{C_{SP}} = \frac{1+P^{\frac{1}{3}}+P^{\frac{2}{3}}}{3}$$

Proof: Follows from Theorem 2.2.

Finally, a result for n-dimensional hypercube meshes will be proven.

Theorem 2.3 For an n-dimensional mesh having s elements in each dimension using the standard n-dimensional stencil, the linear to n-dimensional communication ratio is

$$\frac{1+P^{\frac{1}{n}}+P^{\frac{2}{n}}+...+P^{\frac{n-1}{n}}}{n}.$$

Proof: Let s be the length of the sides of the hypercube and β be the number of communications (sends) per element in each direction of each dimension.

If the hypercube is partitioned linearly, then the total area of the cuts is $s^{n-1}(P-1)$ yielding a total communication of $C_{LP} = 2\beta s^{n-1}(P-1)$.

If the hypercube is partitioned into $q^n = P$ sub-hypercubes, then the total area of the cuts is $ns^{n-1}(q-1) = ns^{n-1}(\sqrt[n]{P}-1)$ yielding a total communication of $C_{LP} = 2\beta ns^{n-1}(\sqrt[n]{P}-1)$.

Hence the communication ratio is

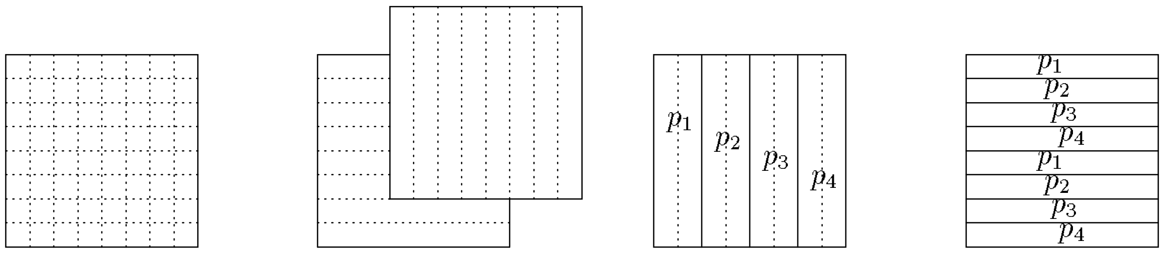
$$\begin{aligned}\frac{C_{LP}}{C_{SP}} &= \frac{2\beta s^{n-1}(P-1)}{2\beta ns^{n-1}(\sqrt[n]{P}-1)} \\ &= \frac{s^{n-1}(P-1)}{ns^{n-1}(\sqrt[n]{P}-1)} \\ &= \frac{1+P^{\frac{1}{n}}+P^{\frac{2}{n}}+\ldots+P^{\frac{n-1}{n}}}{n} \quad \square\end{aligned}$$

Theorem 2.3 shows that as the dimensionality of the problem and the number of processors increases the natural topology parallelization has ever greater advantage over the linear parallelization.

For a concrete illustration, consider using 1024 processors in parallelization of a problem with meshes of size 1024x1024. The linear parallelization would then have 1024 elements per processor (one column of the logically 2-dimensional meshes) with 2048 boundary values to send and receive on each "internal" processor. Note that the two outside (first and last) processors would only send 1024 boundary values, they would not even be computing any results. The sub-block parallelization would also have 1024 elements per processor, but they would form a 32x32 sub-block with 128 boundary values to communicate on "internal" processors. External processors would be computing between 93 and 97 percent as much as internal ones. This example makes it clear that not only is communication per processor reduced by using the natural topology for parallelization but load balancing may also be improved. Hence, the computation to communication ratio is improved by parallelizing according to the natural problem topology. This result is used as motivation throughout the research.

2.2 The Fortran D Language and Extensions

Fortran D supports both *alignment* and *distribution* specifications. A DECOMPOSITION statement specifies an abstract problem or index domain. An ALIGN statement maps array elements onto one or more elements of a decomposition. This provides the minimal information necessary for reducing data movement for the program, given an unlimited number of processors. A DISTRIBUTE statement groups decomposition elements and maps them to the finite resources of the physical machine. Sample data alignment and distributions are shown in Figure 2.2.



```
DECOMPOSITION      REAL A(N,N)          DISTRIBUTE       DISTRIBUTE
    D(N,N)         ALIGN A(I,J)          D(:,BLOCK)        D(CYCLIC,:)
                    with D(J-2,I+3)
```

Figure 2.2 Fortran D Data Decomposition Specifications

Fortran D also supports irregular data distributions and dynamic data decomposition, i.e., changing the alignment to or distribution of a decomposition at any point in the program. To permit a modular programming style, the effects of data decomposition specifications are limited to the scope of the enclosing procedure. However, procedures do inherit the decompositions of their callers. The complete language is described in detail elsewhere [FHK+90].

Note that the original Fortran D language specification does not support alignment of linearized arrays according to their logical topology and that irregular data distributions require runtime support. This work makes it possible to more efficiently compile the class of linearized regular application codes.

In an attempt to provide better support for linearized applications, Fortran D is extended in two ways. Dimension specification provides the basis for loop bounds updates and communication generation. Index array specifications further simplify communication analysis and in many cases eliminate the need for runtime support.

2.2.1 Logical Dimension Specification

The logical dimensions of the linearized dimensions of an index array are provided in the LOGICAL DIMENSIONS statement. This allows the 2-dimensional array `z` to be aligned as if it were declared to be 3-dimensional, i.e.,

```
      real z(ni*nj,np)
C     logical dimensions z((ni,nj),np)
C     align z(i,j,*) with d(j,i).
```

In this example `(ni,nj)` are the logical dimensions for the first declared dimension of the array `z`. The linearized array is then indexed according to the logical dimensions in the ALIGN statement. The array is still indexed elsewhere according to its declaration. Hence, the program body does not have to be rewritten to parallelize the application according to its natural topology.

For a more complete example, the logical layout and distribution of `x` from Figure 2.3 is shown in Figure 2.4, where `nx=ny=8`.

```
      function a()
      parameter ($nprocs = 4)
      real x(nx*ny), y(nx*ny)
C     decomposition d(nx,ny)
C     logical dimensions x((nx,ny)), y((ny,nx))
C     align x with d
C     align y(i,j) with d(j,i)
C     distribute d(block,block)
      ...
      return
```

Figure 2.3 Alignment Statements for Linearized Arrays.

Once a linearized array has been aligned to a DECOMPOSITION of the correct dimensionality, the DECOMPOSITION can be distributed as is most appropriate for the problem. This allows greater flexibility in distribution. For example, in the code fragment of Figure 2.3, the (block,block) distribution of array `x` and the transposed (block,block) distribution of array `y` cannot be accomplished in the current version of High Performance Fortran or as a regular computation in Fortran D.

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Figure 2.4 Distribution of 64 element (logical 8x8) array x.

2.2.2 Index Array Specification

Linearized arrays are typically indexed in one of two ways. The most straightforward indexing method is direct inline computation of linearized indices according to Fortran's standard array storage allocation. The other common indexing approach is to use an index array to store the index of the neighbor, e.g.,

$$\text{west}(12) = 4$$

in array `x` of Figure 2.4.

Index array specifications are added to Fortran D as part of the support mechanism for linearized arrays.

The code in Figure 2.5 specifies that in the logical topology: `A(ind(i,1))` is the element of `A` north of element `A(i)`, `A(ind(i,2))` is the element of `A` at position `i`, `A(ind(i,3))` is the element of `A` south of element `A(i)`, and `A(ind(i,4))` is the element of `A` west of element `A(i)`, `A(ind(i,5))` is either the element of `A` at position `i` or the element of `A` east of element `A(i)`. This multiple use of `ind(i,5)` is allowed for applications such as UTCOMP, which has an index array where the element pointed to depends on the value in another array, but it is always one of a small set of elements. With this type of specification such conditional indirections are supported. `A(ind(i,6))` is the element of `A` south and east of the element of `A` at position `i`.

In general, for a specification $[(\mathtt{i},\mathtt{c}),\mathtt{d}_1,\mathtt{o}_1,\mathtt{d}_2,\mathtt{o}_2,\ldots]$, the `i` indicates the index location to consider where $\mathtt{d}_\mathtt{i}$ has an offset of $\mathtt{o}_\mathtt{i}$. Since `ind(i,4)` is one entry left

```
      parameter (ni = 10, nj = 10)
      real A(ni*nj)
      integer ind(ni*nj,6)
C     decomposition d(ni,nj)
C     logical dimensions A((ni,nj)), ind ((ni,nj),6)
C     index array 2D ind [(i,1),1,-1], [(i,2),1,0],
C                 [(i,3),1,+1], [(i,4),2,-1],
C                 [(i,5),2,+1], [(i,5),2,0],
C                 [(i,6),1,+1,2,+1]
C     align A with d
C     align ind(i,j,k) with d(i,j,*)
C     distribute d(block,block)
      ...
      return
```

Figure 2.5 Logical Topology Specification.

of center in the second dimension, `A(ind(i,4))` refers to `A(i-ni)` in the linearized array where `ni` is the number of elements in the first dimension. The syntax of index array specifications may be modified as applications with more general communication stencils are studied.

2.3 Compilation

This section begins with an outline of the strategy used to compile linearized applications. After that, the compilation strategy is illustrated with an example. Note that the constants are actual sizes, not just declared sizes.

In this work, it is assumed that all of the arrays in a given loop are similarly linearized, aligned (mapped), and distributed. The first example deals with the use of directly computed indices for linearized arrays. The second example deals with index arrays. Finally, only unit (+1 or -1) loop step sizes and step sizes that are multiples of the number of elements in the first dimension are allowed. Relaxation of these restrictions will be discussed after the basic compilation strategy is presented.

The examples used for illustration in this work come from UTCOMP. UTCOMP is a University of Texas code that simulates 3-dimensional miscible gas displacements. This problem of parallelization of linearized applications and UTCOMP were brought to my attention by M. Ramé.

2.3.1 Strategy

As in the example of Figure 2.6, all logical dimension specifications for linearized ar-

```
      program main
      parameter (nip=1, njp=1, n$proc=nip*njp)
      parameter (nim = 1024, njm = 244, nijm = nim * njm)
      real x(nijm), y(nijm), c(nijm), n(nijm), s(nijm), e(nijm), w(nijm)
C     decomposition d(ni,nj)
C     logical dimensions n((ni,nj)),s((ni,nj)),e((ni,nj)),w((ni,nj))
C     logical dimensions x((ni,nj)),y((ni,nj)),c((ni,nj))
C     align x, y, n, s, e, w with d
C     distribute d(block,block)
      read(*,*)ni,nj
      nij=ni*nj
      call compute(ni, nj, nij, x, y, c, n, e, w)
      return

      subroutine compute(ni, nj, nij, x, y, c, n, s, e, w)
      parameter (nim = 1024, njm = 244, nijm = nim * njm)
      real x(nijm), y(nijm), c(nijm), n(nijm), s(nijm), e(nijm), w(nijm)
C     align x, y, n, s, e, w with d
C     distribute d(block,block)
      do i = ni+1,nij-ni
         y(i) = x(i)*c(i) + x(i-1)*n(i) + x(i+1)*s(i) + x(i-ni)*e(i) + x(i+ni)*w(i)
      enddo
      return
```

Figure 2.6 Example 1: Linearized Array Computation.

rays should use variables that are compile time constants or input runtime constants.

These annotations could then be used for index analysis to support communication generation.

To support direct computation of linearized indices, such as via a function call, the compiler would need to be capable of evaluating the index computation to determine ownership of data for communication. The logical dimension specifications support this analysis. This would eliminate the need to execute an inspector. Details on how such index computations may be evaluated by the compiler are available elsewhere [Koe90]. Once such computations are recognized, the compiler can determine whether the indirection implies a communication from the distribution in effect. Communication can be generated at this point because the indices of the source and sink of the dependence are known, as are the processors that store the values. In the cases where communication is necessary, the compiler can generate and insert communication for getting the correct element(s) of the linearized array.

To support the use of index arrays, the index array specification supplies the information necessary for communication generation.

Other considerations for compilation include resizing arrays in declarations and updating loop bounds based on the processor id. Loop bound computations are more complicated than for non-linearized computations as each processor has a set of linearized array indices. This corresponds to one contiguous block of the logical array but it is not a contiguous block of the linearized array.

```
modify declarations
compute loop bounds for each processor
exchange standard pattern phantom boundaries
exchange specialized phantom boundaries
modify loop
    use new loop bounds
    put correct north/south neighbors in temporaries
    modify computation to use temporaries
```

Figure 2.7 Road Map for Linearized Application Modification (Example 1)

```
      subroutine compute(x, y, c, n, s, e, w)
C  modify declarations
      parameter (nip=1, njp=1, n$proc=nip*njp, ni=1024, nj=244, nij=ni*nj)
      parameter (n$dip=ni/nip+1, n$djp=nj/njp+1,n$dij=n$dip*n$djp,n$sp=-(n$dip+n$djp))
      real x(n$sp:n$dij-n$sp), y(n$dij), c(n$dij),n(n$dij), s(n$dij), e(n$dij), w(n$dij)
      integer lb$1, ub$1
      common /buf$s/ buf$1(-n$sp), buf$2(-n$sp),buf$3(-n$sp),buf$4(-n$sp)
      common /proc$c/ my$proc,my$col,my$row,in$p,is$p,ie$p,iw$p,ine$p,inw$p,ise$p,isw$p,
*        n$ip,n$jp,n$di,n$dj,n$i,n$j,in$st,in$nd,is$st,is$nd,ie$st,ie$nd,iw$st,iw$nd,n$s,n$f
C  compute loop bounds for each processor
      lb$1 = i$bnd2(ni+1,1,1)
      ub$1 = i$bnd2(nij-ni,-1,-1)
C  exchange standard pattern phantom boundaries
      call exc$b1(x,n$sp,n$dij-n$sp,101)
C  exchange specialized phantom boundaries
      if (n$jp .gt. 1) then
          if (my$row .eq. 1) then
              call b$f_dt2(x(1),2,n$di,1,1,buf$1)
              call csend(105,buf$1,8*(n$dj-1),nip-1+nip*(my$col-1),my$pid)
              if (my$col .gt. 1) call csend(106,x,8,n$ip-1+n$ip*(my$col-1),my$pid)
              call crecv(107,x(in$st+1),8*(n$dj-1))
              if (my$col .gt. 1) call crecv(108,x(in$st),8)
          elseif (my$row .eq. n$ip) then
              call b$f_dt2(x(1),n$di,n$di,1,n$dj-1,buf$1)
              call csend(107,buf$1,8*(n$dj-1),1-1+n$ip*(my$col-1),my$pid)
              if (my$col .lt. n$jp) call csend(108,x(n$di*n$dj),8,1-1+n$ip*(n$jp+1-1),my$pid)
              call crecv(105,x(is$st),8*(n$dj-1))
              if (my$col .lt. n$jp) call crecv(106,x(is$nd),8)
          endif
      endif
C  modify loop – use new loop bounds
      do i = lb$1, ub$1
C      put correct north/south neighbors in temporaries
          if (1 .eq. mod(i,n$di)) xim$1 = x(in$st+i$n-1)
          if (1 .ne. mod(i,n$di)) xim$1 = x(i-1)
          if (0 .eq. mod(i,n$di)) xip$1 = x(is$st+i$n-1)
          if (0 .eq. mod(i,n$di)) xip$1 = x(i+1)
C      modify computation to use temporaries
          y(i) = x(i)*c(i) + xi$1*n(i) + xip1*s(i) + x(i - n$di)*e(i) + x(i + n$di)*w(i)
      enddo
      end
```

Figure 2.8 Example 1: Linearized Array Computation.

2.3.2 Illustration of Strategy

The compilation strategy is now illustrated with an example. Figure 2.6 shows the Fortran D source code for a linearized computation similar to some of the direct linearized address computation code in UTCOMP. This example is a 2-dimensional code similar to a part of one of the 3-dimensional routines from UTCOMP.

For good performance in an application implementation, the user or an automatic alignment and distribution system [LHKD91, BFKK90] would align each array to minimize communication. In the example of Figure 2.6, all of the arrays are perfectly aligned with decomposition `D`.

Code modification for this simple example is outlined in Figure 2.7. This outline serves as a roadmap for the code, shown in Figure 2.8, that was generated via hand compilation. A setup routine was called in the main program to initialize the variables in the common block `proc$c`.

The function `b$f_dt2`, called as

```
call  b$f_dt2(x(1),il,ih,jl,jh,buf),
```

copies from the `n$di` by `n$dj` array `x` the sub-array `x(il:ih,jl:jh)` into `buf`.

A single function is written to compute lower and upper bounds of loops based on linearized 2-dimensional arrays distributed in two dimensions. First, the bounds computation function determines the location, in the logical 2-dimensional array, of the input bound. Then, the first location that the calling processor owns in the direction that the loop iterates is determined via the known mapping of elements to processors. This same basic procedure can be followed for linearized `n`-dimensional arrays distributed in `n` dimensions.

One of the disadvantages of linearization of arrays is that communication overlap arrays are more complicated. For efficiency, starting and ending indices of overlaps are computed in the setup routine then used throughout the program to access boundary values. One advantage to this approach is that, with careful packaging during buffering for sends, most communications do not require unpackaging for receives. Boundary messages can be received directly into their linearized storage.

2.3.3 Details of Compilation

There are three stages in the compilation process: analysis, interprocedural propagation, and code generation. Each of these stages will be examined in detail. Each stage is discussed in terms of the needed modifications/additions to the Rice Fortran D compiler [HKT91b]. An outline of the compilation process is shown in Figure 2.9.

- Analysis
    - collect index information
    - note potential index/indirection arrays
    - note potential multiple indirections
    - store logical dimensions of linearized arrays
    - store index array specifications
- Propagation
    - propagate specifications down the call chain
    - propagate index array notes up the call chain
- Code Generation
    - modify declarations
    - insert call to setup routine
    - modify initialization of index arrays
    - update loop step sizes and loop bounds
    - modify direct linearized address computations
    - modify logical statements using values of index arrays
    - generate communication

Figure 2.9 Compilation Process Outline

2.3.4 Analysis

Index information is collected on a per loop basis for later communication generation. Information is also collected about the use of potential index/indirection arrays. Multiple indirection candidates must be noted in the symbol table so that boundary indices can be computed if necessary.

Specifications for logical dimensions of arrays and specification of index arrays need to be collected along with the alignment and distribution information collected in the Rice Fortran D compiler. Actual dimension size names are collected for each logically distributed dimension of the linearized arrays. This information is available directly from the LOGICAL DIMENSIONS statements in symbolic form.

Logical tests involving values from potential index arrays are marked for later processing.

2.3.5 Propagation

All of the specification information collected during analysis (e.g., logical dimensions of arrays, index arrays, and dimension sizes) is propagated down the call chain. If different index array specifications are associated with the same dummy parameter in a subroutine, the subroutine must be cloned for each different specification. In this case, clone names must be propagated up the call chain to allow selection of the correct clone.

2.3.6 Code Generation

In each routine, array declarations for all distributed linearized arrays are modified to be the largest size needed by any processor plus the size needed for storing the phantom boundary values.

Immediately after the initialization of the actual dimension sizes (assignment or input), a call to a setup routine with the sizes as parameters is inserted. The setup routine identifies the processor, its location in the logical processor array, the processor identifiers of all logical neighbors in the logical processor array, and the offsets for storage of boundary values.

In Figure 2.10, the neighboring processors are indicated in a 3-dimensional logical processor array for a seven point stencil application. The diagram also indicates what boundary allocation is necessary. For each of the boundaries, starting and ending allocation addresses are computed in the setup routine. These addresses are then used to simplify and improve the efficiency of communication.

If index arrays are used, then their computation must be modified so that boundary addresses are used. This is straightforward when there is at most one indirection in any data access and there are only references to indices of the form i-1, i, or i+1 in any dimension. Otherwise multiple layers of phantom boundaries must be supplied. When double indirection is used, the boundary elements of the index array must be computed and filled in. Fortunately, the more complicated cases having many compound indirections and/or distant elements as neighbors are not the norm for these regular computations. The second example, in the routine called `disper`, does use

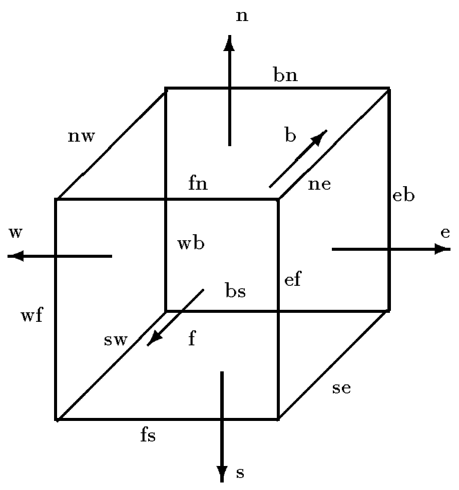


Figure 2.10 3-Dimensional Processor Layout

one form of double indirection. Note that the double indirection discussion implies relaxation of the "owner computes rule" for index arrays.

Loop modification begins with step and bounds updates. Consider the simplest case, where the step is one. To find a processor's lower bound, its first index (in the original array indexing) greater than or equal to the original lower bound is found and translated into the (smaller) processor's array index space. Since the mapping of elements to processors is known and fixed, this not difficult. Let δr and δc be the number of rows and columns that a processor owns. Basically, in the 2-dimensional case, to find the lower bound on any processor, assuming the step size is positive, the following steps are performed:

- Determine the row and column of the bound in the original array.
- If the bound's column is greater than the last column the processor owns a portion of, then set lbr to $\delta r + 1$ and lbc to $\delta c + 1$.
- Else if the bound's column is less than the first column the processor owns a portion of, then set lbr to 1 and lbc to 1.
- Else

  - Set lbc to the bound's column minus the index of the first column the processor owns a portion of plus one.
  - If the bound's row is greater than the last row that the processor owns a portion of, then add 1 to lbc and set lbr to 1.
  - Else if the bound's row is less than the first row that the processor owns a portion of, then set lbr to 1.
  - Else set lbr to the bound's row minus the index of the first row that the processor own's a portion of plus one.

- Compute the new lower bound as $(lbc - 1) * \delta c + lbr$.

The process for the upper bound is similar. Indeed one routine was written and $-1 * sign(step)$ was passed for the step to compute the upper bound. Non unit steps complicate the computation. Fortunately, typical non-unit steps are fairly straightforward. In particular, probably the most common non-unit step is the size of the first dimension. In this case, bounds are found as above but the step size is modified to be the number of elements in the first dimension of the sub-array owned by the processor. In the completely general case, a different starting position could be needed for each logical column of the sub-matrix owned by each processor. After bounds have been computed, the body of the loop is considered. If linearized addresses are computed directly in the loop, as were seen in the first example, a case structure must be built that moves a neighbor value from either normal neighbor storage or phantom boundary storage in the processor's sub-array to a temporary. The temporary is then used in place of the variable with its direct linearized address computation. If index arrays are used, this modification is not needed because the index array values have already been modified to handle the boundaries. This may give an execution time advantage to parallelization of linearized applications that use index arrays for indirection rather than direct computation of linearized addresses.

Logical statements that involve tests of actual index array values require modification. An example of such usage is:

```
iwst = ind(i,4)
⋮
if (iwst .ne.  0)  A(i) = f(A(iwst))
```

where 0 is out of bounds for `A`. Because `A(iwst)` may now be an internal boundary value, this computation must be modified to something like

```
iwst = ind(i,4)
⋮
if (iwst .ne.  A$0)  A(i) = f(A(iwst))
```

where `A$0` is a special (invalid) index value that is also used during initialization in place of zeros. If such uses can not be deciphered at compile time, runtime checks must be added for correctness or the user must supply further information.

The final code modification involves communication generation. With direct linearized address computation, communication generation involves a simple modification to the Rice Fortran D compiler. The only difference is in recognizing and using the higher dimensional nature of the arrays and their indexing. For example, recognition of a reference to `A(i-ni)` as a reference to the element to the west of element `i` in the logically 2-dimensional (or higher) array `A` implies communication is only needed in a west to east pattern along the boundary faces in the processor array.

With the use of index arrays, communication generation is even simpler as a reference to `A(ind(i,4))` is specified to be a reference to the element to the west of `A(i)`. Hence, the index array specifications provide an upper bound on the communication needed for an indirection through an index array. This is an upper bound due to entries such as

```
(ind2(i),.,ind2(i)),
```

which indicates that `ind2(i)` refers to either element `i-1` or element `i+1` in the appropriate dimension. In the cases where such multiple entries occur, communication must be generated at compile time to satisfy all entries or runtime support must be provided to determine what communication is necessary and perform it. In regular applications there will only be one communication direction associated with most index array usage. With this exception, communication optimization is essentially the same as that needed in any good Fortran D compiler [HKT91a].

2.3.7 Relaxation of Restrictions

Certain restrictions were placed on the codes that were considered for compilation in this work. This section discusses how to handle codes that do not adhere to those restrictions.

Using Arrays with Different Linearizations

If different arrays have been linearized in different ways, this multiplicity must be accommodated. One way to handle this would be to call a setup routine for each type of linearization and allocate common blocks for each. In bounds computations and generated communication, information would be selected from the appropriate common block. For communication between arrays with different linearizations, the analysis is essentially the same analysis done in the Rice Fortran D compiler for arrays with different distributions [HKT91c].

Arbitrary Step Sizes

To support arbitrary step sizes, the step size is first computed for the sub-blocks. A starting and ending position must be computed for each sub-column on each processor. Finally, the loop must be modified to iterate over the columns with a nested loop that iterates from that sub-column's starting to ending positions by the sub-block step size.

2.3.8 Optimization for Linearized Array Computations

Sometimes linearization for vectorization introduces extraneous computations. This is usually due to computing boundary values in the same manner (as part of the vector computation) as interior points. The boundary conditions are then recomputed correctly after the vector-based computation. Typically, this also implies non-productive communication if the code is parallelized based on the linearized code. A good compiler should recognize that the boundary values are stored twice but not used between the stores. Once this has been recognized, the compiler can eliminate the useless first computation. The compiler should then recognize that the communication associated with the first computation is also superfluous and may be eliminated as well. Even if the extraneous computations are not eliminated, the communication may still be eliminated and maintain correctness (with proper initialization).

In the example of Figure 2.6, only the interior points loop has been shown. In the complete implementation, there would be boundary updates following the loop shown. The computations for elements 9, 16, 17, etc. would all be recomputed as boundary values without ever having been used. Hence their computation in this loop is superfluous.

2.4 Experimental Performance Results

In order to demonstrate the performance achievable when using the techniques described in this chapter, two example applications were hand translated. The codes were then compiled with release 3.3 of the Intel iPSC/860 Fortran compiler.

Timing experiments were run on the Intel iPSC/860 at Rice University. The machine has 32 processors, each with 8Mbytes of memory and a vector processor. Actual run times are presented for both of the example application codes. All of the times presented are the minimum of at least three executions. Timings are in seconds and are rounded to five significant digits.

2.4.1 2-d Direct Linearized Address Computation Application

The original (sequential linearized) compute routine for this example is shown in Figure 2.6. This application was parallelized in three ways. First, the approach described in this chapter was used to achieve the natural 2-dimensional parallelization. Next, the procedure, described in Section 2.3.8, is used to optimize the communication of this 2-dimensional parallelization. Finally, the Fortran D compiler was used to generate a linear (1-dimensional) parallelization, the only type of regular parallelization possible with the original definition of Fortran D. In all cases, the meshes (each of which was 1024x244) were divided evenly across the processors.

Upon computing efficiencies for this example, it was found that the new parallel code ran at better than 100 percent efficiency. Some time was spent tracking down the cause of this anomaly. The actual runtime of the code generated for this application is highly dependent on the precise memory allocation used. The declarations were changed (and nothing else) in the original program to be the same as those in the parallel program. The original sequential code ran in 1.0050 seconds while the "reallocated" code ran in 0.53127 seconds. This illustrates how much difference memory allocation can make on small, simple, programs. On large programs with many data structures, changes in memory allocation do not appear to make significant differences in runtime. For all speedup comparisons, the timings of the "reallocated" code are used. Hence comparisons are made against the "best" known sequential code.

The timings for the first two parallelizations are shown in Table 2.11. These timings provide an example of the type of performance improvement possible when the communication optimization procedure in Section 2.3.8 is performed on a small simple application.

Num. Procs	Unoptimized	Optimized	unopt./opt.
2	2.5932E-01	2.4425E-01	1.06
4	1.2532E-01	1.1585E-01	1.08
8	6.4001E-02	5.9694E-02	1.07
16	4.0701E-02	3.1243E-02	1.30
32	3.7446E-02	1.6700E-02	2.24

Figure 2.11 Communication Optimization Timing Comparison

In the following comparisons with Fortran D, the optimized code is used. Table 2.12 presents runtimes and speedup factors comparing the results of my approach to the runtime of the reallocated sequential code. These results illustrate that the techniques described here are capable of providing an improvement in performance even for such a simple computation as Example 1.

	1-d Fortran D		Logical 2-d	
Num. Procs.	runtime	speedup	runtime	speedup
2	5.5696E-01	0.95	2.4425E-01	2.18
4	2.7976E-01	1.90	1.1585E-01	4.59
8	1.2224E-01	4.35	5.9694E-02	8.90
16	4.7520E-02	11.18	3.1243E-02	17.00
32	2.3309E-02	22.79	1.6700E-02	31.81

Figure 2.12 Example 1 Timing Results

Next, consider the superlinear speedups in Table 2.12. The Intel i860 has a 2 way, set associative, 8Kbyte cache with a write-back policy. This cache is the cause of the superlinear speedups. In this trivial example, there are only a few arrays and a very simple memory access pattern. The cache makes a tremendous difference for this application. To illustrate this the sequential program was run on the same size problem that was run on each of the parallel processors. Since the original problem had 1024x244 meshes, size (1024x244)/P meshes for P = 2, 4, 8, 16, and 32 were used. Table 2.13 illustrates runtime efficiencies with cache effect factored out.

P	Sequential	1-d Fortran D			Logical 2-d		
	runtime	runtime	speedup	eff.	runtime	speedup	eff.
2	1.7928E-01	5.5696E-01	0.64	32.2	2.4425E-01	1.47	73.4
4	8.5883E-02	2.7976E-01	1.23	30.1	1.1585E-01	2.97	74.1
8	4.3315E-02	1.2224E-01	2.83	35.4	5.9694E-02	5.80	72.6
16	2.1925E-02	4.7520E-02	7.38	46.1	3.1243E-02	11.23	70.2
32	1.0182E-02	2.3309E-02	13.98	43.7	1.6700E-02	19.51	61.0

Figure 2.13 Example 1 Cache Effect Timing Results (Mesh size: (1024x244)/P)

Since it appears that there could be a memory allocation imbalance between the Fortran D code and the logically 2-dimensional code, it can not be stated categorically that this procedure outperforms Fortran D on this application. It can be said that an old (a preliminary compilation procedure was used to generate it) version of the code, with similar memory allocation performance to the Fortran D code, also outperformed the Fortran D code. Next a more realistic application provides a better look at performance results.

2.4.2 3-d Index Array Application

The second example consists of three routines from UTCOMP. Two of the routines are index array computations, one of which must be modified to support double indirection usage. The third routine, `disper`, which computes the dispersion term for UTCOMP, is over one thousand lines long, uses index arrays including double indirection, and is representative of the code throughout UTCOMP [Ram92].

This code was not parallelized using standard Fortran D as that would require runtime support. The requirement for runtime support is due to the indirections that use index arrays. This support is not available in the current version of the Rice Fortran D project. The results of a hand parallelization using the inspector/executor approach for regular problems supported by the `PARTI` routines are used for comparison. This parallelization will be discussed first, in Section 2.4.3, then I will return to a discussion of the performance results in Section 2.4.4.

2.4.3 PARTI Parallelization of 3-d Index Array Application

In the original treatment of applications using the PARTI routines, the data element to processor mapping was determined by having the user set up an array on each processor with global indices of the elements the processor owned. The inspector then built a distributed translation table that described the mapping of global indices to processors and the offsets in the array. This approach is still used for irregular problems that do not have standard distributions. For irregular problems with standard distributions, the indices are now "dereferenced" via a function. The elimination of the distributed translation table provides up to a factor of five improvement in runtime of the inspector [Sal93]. Standard distributions (supplied as part of PARTI) are BLOCK and CYCLIC. Note that the BLOCK distribution will provide the linear distribution that was discussed in conjunction with the Fortran D results. The user may also modify the PARTI routines to support other regular distributions. To begin with, only the application modifications necessary to use the standard distributions are discussed.

First, consider the inspector. This is a new addition to the application code that must be written to support the initialization of the PARTI data structures. For simple programs, using only single indirections, it is only necessary to call a routine that localizes index array values (converts the global array indices to indices for the specific processor) and constructs a communication schedule for the index array(s). This is easy to do, but disper uses double indirection. When double indirection is used, the inspector is more complicated. For a double indirection like $\mathtt{A}(\alpha(\beta(\mathtt{i})))$, where α and β are index arrays, after β is localized and a schedule for β-induced communication has been computed, the off processor values of α that will be referenced are collected (via a call to a gather routine). Next, localization and communication schedule generation is performed for α including the gathered elements. But the application is still not quite done. The values of index arrays are used in comparisons in disper. To handle this properly, a copy of the index arrays (before they were localized) must be kept to use in the comparisons.

Now, consider what needs to be done in the executor portion of the code. Firts, declarations must be modified so that storage is declared only for the portion of each array that a processors owns. Index array computations must be modified so that each processor computes only its set of index values (in terms of the global index space). The program must be analyzed and calls inserted at the appropriate places to perform communication. Loop bounds must be correctly computed for each processor based on the portion of the arrays that the processor owns. Logical comparisons must be modified to use the values of the global index arrays (recall that they were saved by the inspector).

Finally, the application is ready to run using a `PARTI` supplied regular distribution. In the comparisons of Section 2.4.4, the standard block distribution was used and is refered to as "`PARTI/1-d`".

In order to run the application using its natural topology parallelization in `PARTI`, the user must modify one of the `PARTI` routines. In particular, the user must write code that, when given a vector of indices in the global index space, will generate vectors of owning processor numbers and offsets. The routine was modified to support a 3-dimensional distribution of the linearized arrays. In the comparisons of Section 2.4.4, this code is refered to as "`PARTI/3-d`".

Work is being done on a compiler to automatically generate inspector/executor programs for a restricted type of source program. The version of that compiler in progress should be able to handle the double indirection and/or the logical comparisons with global index values [Sal93].

Also, note that I am not claiming that the approach presented can be used to solve all of the problems that can be solved using `PARTI` routines. The `PARTI` system is used for comparison because it is capable of handling the problems under consideration (as well as much more general problems, with more user work).

2.4.4 3-d Index Array Application Results

I now return to the runtime results for UTCOMP. The runtime for the original sequential code is 109.76 seconds.

As expected, the natural topology parallelization using `PARTI` significantly outperforms the linear parallelization using `PARTI`. The results in Table 2.14 illustrate this.

An efficiency experiment was done to see what part of the apparent good performance in Table 2.14 was due to cache effect. Table 2.15 compares the timings for the sequential program on the mesh sizes (10x24x24)/P for P = 2, 4, 8 16, and 32 to the timings for the parallel programs. This table shows that the time for running a problem of size (10x24x24)/P is near 1/P times the time for running a problem of size 10x24x24. It is clear that there is much less cache effect in this application than was present in the first example.

	PARTI/1-d		PARTI/3-d		Logical 3-d	
Num. Procs.	runtime	speedup	runtime	speedup	runtime	speedup
2	69.339	1.58	60.271	1.82	57.206	1.92
4	34.868	3.15	30.186	3.64	28.597	3.84
8	17.070	6.43	15.134	7.25	14.268	7.69
16	8.3085	13.21	7.5702	14.5	7.1471	15.36
32	4.2477	25.84	3.8696	28.36	3.6997	29.67

Figure 2.14 UTCOMP Timing Results for `disper`

From the timing results, this new compilation approach slightly outperforms the results achieved by the best hand-coded use of the `PARTI` routines. The difference in performance combined with the difference in the amount of work that the user must do makes this a preferable approach to parallelization of linearized applications.

	Sequential	PARTI/1-d		PARTI/3-d		Logical 3-d	
P	runtime	runtime	eff.	runtime	eff.	runtime	eff.
2	54.106	69.339	78.0	60.271	89.8	57.206	94.6
4	26.654	34.867	76.4	30.186	88.3	28.597	93.2
8	12.935	17.070	75.8	15.134	85.5	14.268	90.7
16	6.2814	8.3085	75.6	7.5702	83.0	7.1471	87.9
32	3.0237	4.2477	71.2	3.8696	78.1	3.6997	81.7

Figure 2.15 UTCOMP Cache Effect Timing Results (Mesh size: (10x24x24)/P)

2.5 Chapter Summary

Neither Fortran D nor High Performance Fortran provide regular support for the natural topology parallelization of linearized applications codes. I have presented extensions to Fortran D that permit specification of the logical dimensions of linearized arrays and the use of index arrays to specify regular communication in linearized array references, along with an approach to compiling the resulting programs. Hand simulation of the compilation algorithm shows that runtime support is not necessary for many linearized applications even if indirection is used.

Better performance and, in comparison to the inspector/executor approach, reduced manual recoding effort make this new approach preferable to using the original Fortran D, HPF, or inspector/executor approaches for regular linearized applications.

Chapter 3

Composite Grid Problems

This chapter introduces: 1) the problems that are used to test the automatic distribution algorithms for composite grid applications, 2) a program template for composite grid programs, and 3) a style guideline for development of composite grid programs.

The test problems are first presented. Early in this research, problems were solicited from scientists working on composite grid problems. The selection criteria was that the topology and program statistics come from real applications and that the specifications could be obtained in a timely manner. The ICRM problem descriptions come from two different application areas. Recently, new data sets have been obtained that will be used in the continuation of this research.

The discussion of these problems is concluded with a discussion of programming issues for parallelization. This discussion includes recommendations on programming style for these applications in HPF and a program template.

3.1 Fluid Flow and Aerodynamic Simulations

In these applications, the meshes are large and the computation being done is the same for every element of every mesh. A simple example of a large mesh ICRM problem is the simulation of the material flow through an elbow with two parallel vanes inside. Hugh Thornburg at Mississippi State University generated coupled meshes for this simulation, which are shown in Figure 3.1. There are five 3-dimensional meshes ranging in size from 44,772 to 70,520 elements with a total of 275,356 elements and six couplings in this problem.

Perhaps one of the best known application areas for composite grids is aerodynamics. In aerodynamic simulations, different grids are used to resolve flow in the space surrounding the fuselage, wings, foreplane, pylons, etc. [Yam90, Eri87, SB87, SE87]. In recent years, it has become possible to automatically generate the grids used in some ICRM problems [SW92, SKC88, SGW88, All88, SE87, Tho87, SG92]. Shaw

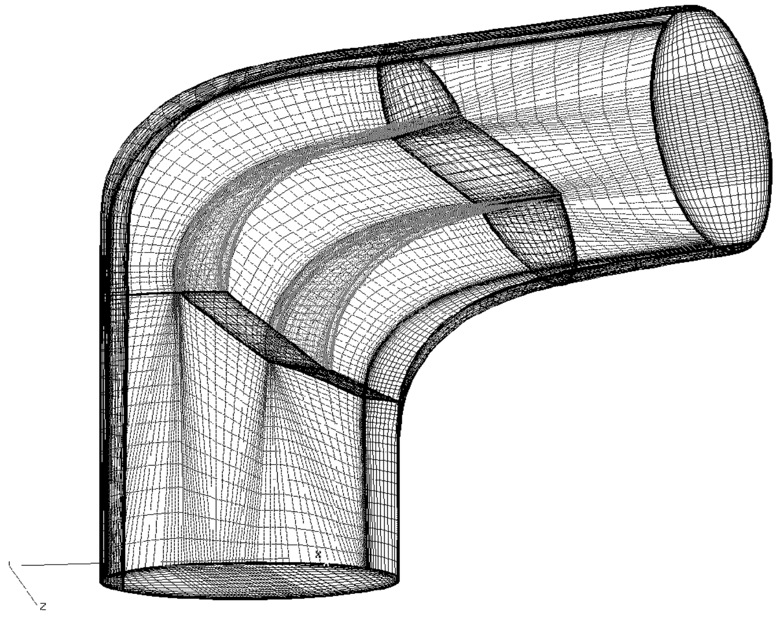


Figure 3.1 Automatically Generated Grids for Flow through an Elbow

and Weatherill [SW92] provide an introduction to the problem of automatic grid generation and illustrates difficulties associated with multiblock problems:

> The fundamental problem that is encountered in the construction of structured, body-conforming meshes for general aerodynamic configurations is that each component of the configuration has its own natural type of grid topology and that these topologies are usually incompatible with each other. In other words, in attempting to discretize the flow domain around an arbitrary set of 2-dimensional shapes, or some complex 3-dimensional shape, a direct conflict arises between the maintenance of a globally structured grid and the preservation of a grid which naturally aligns itself with the local geometric features of a configuration.
>
> This inconsistency has motivated the development of a general category of mesh construction techniques know as multiblock or composite grid generation. Here, the single set of curvilinear coordinates, inherent to a globally structured grid, is replaced by an arbitrary number of coordinate sets that interface node to node with each other at notional boundaries within the physical domain. Returning to the mapping concept, the approach can be viewed as the decomposition of the flow domain into subregions, which are referred to as blocks, each of which is transformed into its own unit cube in computational space. Global structure

> within a grid is sacrificed, but the concept proves the flexibility of connections required to construct a grid whose topological structure, local to each component of a complex configuration, is compatible with the particular geometric characteristics of the component. All points that are connected by a common coordinate system can be directly referenced with respect to each other, but bear no direct structured relationship to any of the grid points that lie in coordinate-sets in other blocks.
>
> Thus, the multiblock technique necessitates information to be defined describing how the blocks connect together...

Both of the aerodynamic test problems that are used for distribution algorithm validation were generated semi-automatically. These examples of grids for such aerodynamic simulations are shown in Figure 3.2 and Figure 3.3. The grid descriptions for both of these examples were provided by Paul Craft working with Dr. Brahat Soni at Mississippi State University. The Fuselage-Inlet-Nozzle problem has 10 meshes ranging in size from 540 to 179,820 elements with a total of 713,766 elements and 25 couplings. In the full F15e simulation there are 32 meshes ranging in size from 540 to 230,688 elements with a total of 1,269,845 elements with 106 couplings. When grids are generated automatically, it is particularly important that distribution is done automatically. The applications programmer must not be required to decipher the automatically generated grid assembly and parallelize the simulation code for a specific configuration by hand.

3.2 Water-Cooled Nuclear Reactor Simulations

In these simulations, not only does the size of the meshes vary greatly but the computation performed varies with the mesh. There are thirteen different types of components associated with reactor simulations: ACCUM, BREAK, FILL, PIPE, PLENUM, PRIZER, PUMP, ROD(or SLAB), STGEN, TEE, TURB, VALVE and VESSEL. Table 3.4 shows the approximate number of additions/multiplications, and divisions* for each element of 1-dimensional, 2-dimensional(ROD), and 3-dimensional (VESSEL) components [JWL94]. Note that these values are an average over all the elements of each component type for a specific problem. The actual values also vary

*These figures were obtained from information collected using the CRAY Hardware Performance Monitor by Susan Woodruff using data sets generated by Jim Lime at Los Alamos National Laboratories.

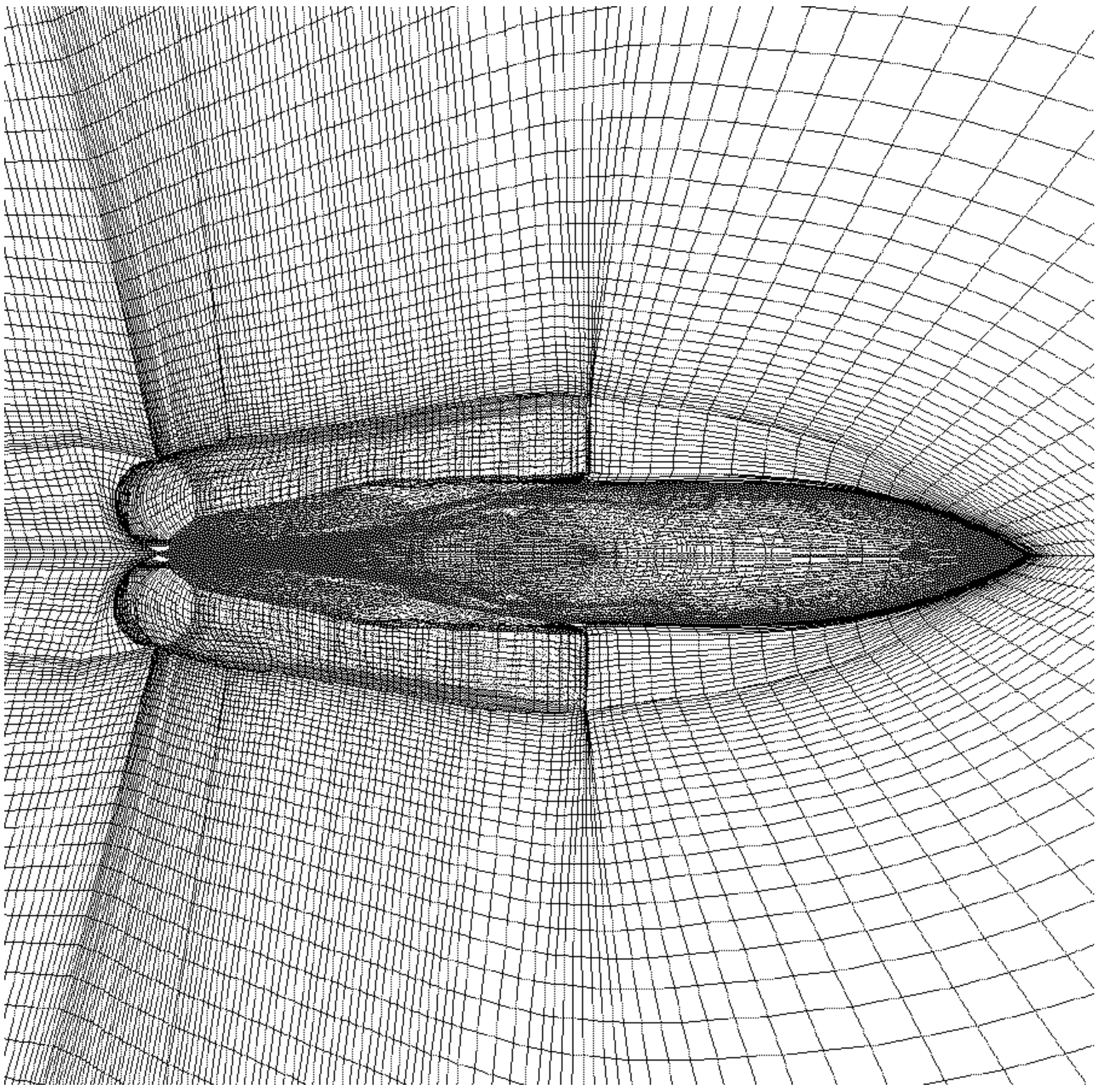


Figure 3.2 F15e Volume Grid Configuration for Fuselage-Inlet-Nozzle

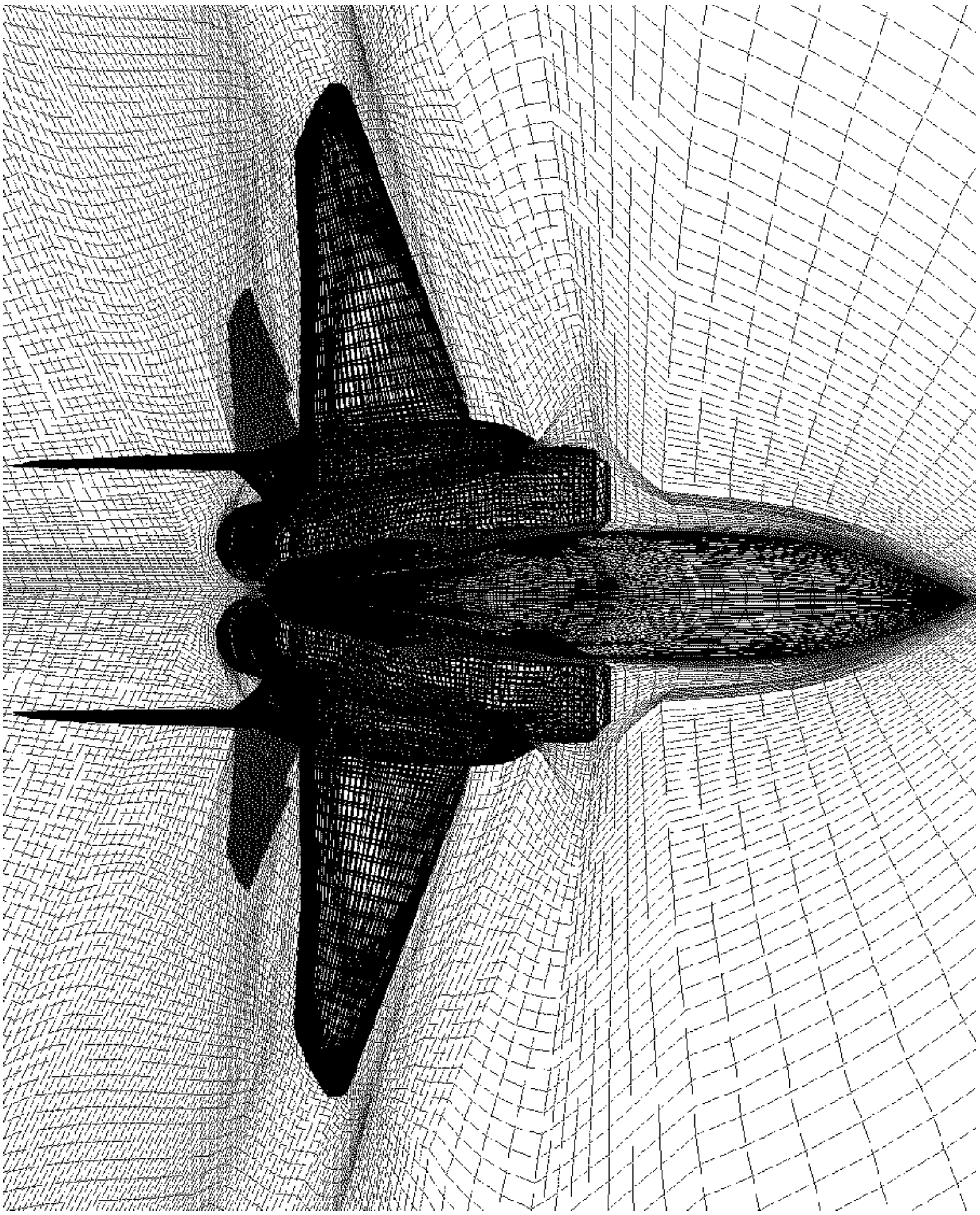


Figure 3.3 F15e Volume Grid Configuration for Full Aircraft

Component Type	# Add/Mults	# Divides
1-D	11,798	704
2-D	64,427	888
3-D	117,057	1073

Figure 3.4 Operation counts for all component types.

by component type. In nuclear reactor simulations, different grids are used for each of the reactor vessels, pipes, pumps, etc. [BSL85].

The first two reactor test problems come from the LOFT model. The LOFT model is a small model that has been used to illustrate concepts in the Los Alamos TRAC manuals. The LOFT reactor model nodalization came from the TRAC manual [LMS93]. There are two versions of this model. The only difference between the versions is that the second version replaces the 3-d reactor vessel with a set of 1-d components and changes the heat structures.

The 3-d version of this model has 28 components (192 3-d cells and 128 1-d cells) and 11 heat structures. Table 3.5 shows the number of each type of component and the number of elements for each type of component.

Component Type	# Components	# Elements
BREAK	1	1
FILL	2	2
PIPE	4	9
PRIZER	2	8
PUMP	2	4
SLABS	11	88
STGEN	1	23
TEE	11	68
VALVE	4	13
VESSEL	1	192

Figure 3.5 Component and element counts for LOFT 3-d reactor model.

The 1-d version of this model has 40 components (169 1-d cells) and 1 heat structure. Table 3.6 shows the number of each type of component and the number of elements for each type of component.

Component Type	# Components	# Elements
BREAK	1	1
FILL	3	3
PIPE	5	12
PUMP	2	4
PRIZER	2	8
SLAB	1	4
STGEN	1	23
TEE	22	105
VALVE	4	13

Figure 3.6 Component and element counts for the LOFT 1-d reactor model.

The next test problem comes from the H.B. Robinson reactor model. Like the LOFT model, the H.B. Robinson model is a small model that has been used to illustrate concepts in the Los Alamos TRAC manuals. The H.B. Robinson reactor model nodalization came from the TRAC manual [BSL85]. This model has 100 components (144 3-d cells and 433 1-d cells) and 21 heat structures. Table 3.7 shows the number of each type of component and the number of elements for each type of component.

The final reactor test problem comes from the Westinghouse AP600 reactor model. The Westinghouse AP600 reactor model nodalization was developed by Jim Lime at Los Alamos National Laboratories with support from the Nuclear Regulatory Commission [LB94]. This model has 173 hydro components (1060 3-d cells and 865 1-d cells) and 47 heat structures. Table 3.8 shows the number of each type of component and the number of elements for each type of component.

These test problems are used in the validation experiments for the small and mixed mesh distribution algorithms. To provide an indication of the complexity of the topology(dimensionality, size, and connectivity) of these problems, Figures 3.9–3.13 show five of the diagrams from the Westinghouse AP600 advanced reactor design. The AP600 design has two loops with one hot leg, one steam generator, two reactor

Component Type	# Components	# Elements
ACCUM	3	12
BREAK	1	1
FILL	27	27
PIPE	17	142
PLENUM	2	8
PUMP	2	6
SLAB	21	126
TEE	32	188
VALVE	15	49
VESSEL	1	144

Figure 3.7 Component and element counts for H.B. Robinson reactor model.

Component Type	# Components	# Elements
ACCUM	2	10
BREAK	10	10
FILL	11	11
PIPE	70	385
PLENUM	3	3
PUMP	4	20
SLAB	47	917
TEE	41	256
VALVE	29	170
VESSEL	3	1060

Figure 3.8 Component and element counts for Westinghouse AP600 reactor model.

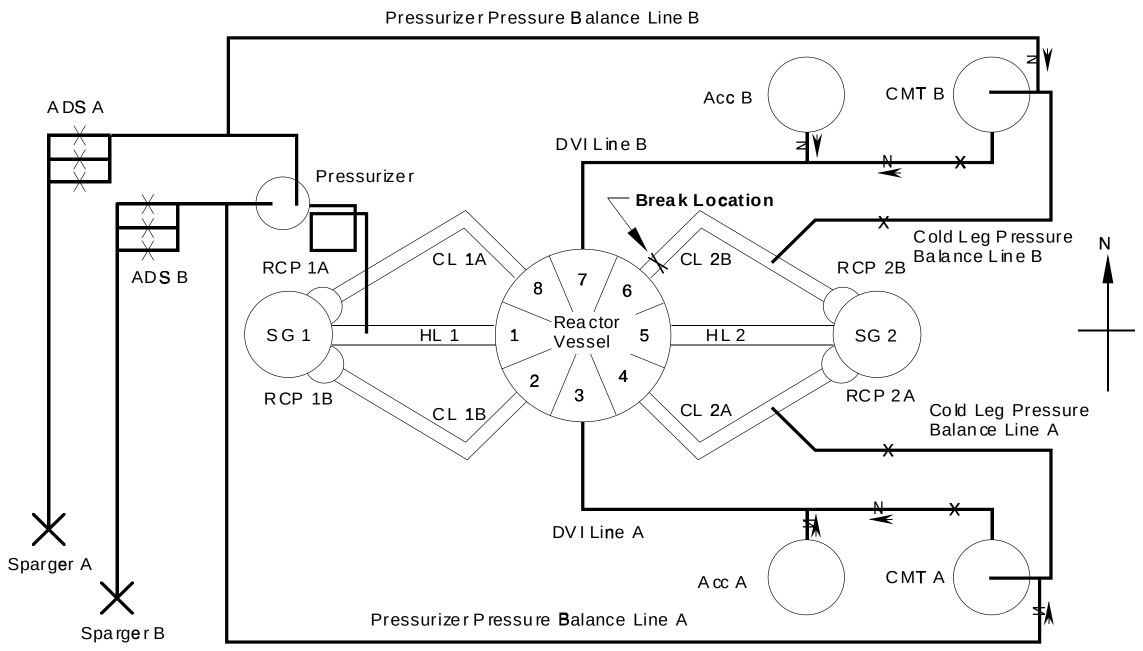


Figure 3.9 Plan view of AP600 model.

pumps, and two cold legs in each loop. Figure 3.9 shows the overall plan of the reactor cooling system, automatic depressurization system, and the passive safety injection system. Figure 3.10 illustrates the relationship of the components in the reactor vessel model. Figure 3.11 shows the relationship of the heat structures to the rest of the components in the reactor vessel model. Figure 3.12 illustrates the relationship of the components in the first coolant loop of the AP600 model. Figure 3.13 illustrates the relationship of the components in the safety systems of the model. The complexity of the AP600 topology illustrates the need for automatic distribution.

3.3 Using Problem Topology for ICRM Parallelization

The composite grid applications considered here have regular computations with regular communication patterns inside of each mesh, but arbitrary connections between meshes. For efficient parallelization in high level languages, such as Fortran D or High Performance Fortran, the mapping of the various meshes to processors must be specified by the user. With this information, the compiler generates a machine dependent parallel program. The work performed by the compiler includes: storage

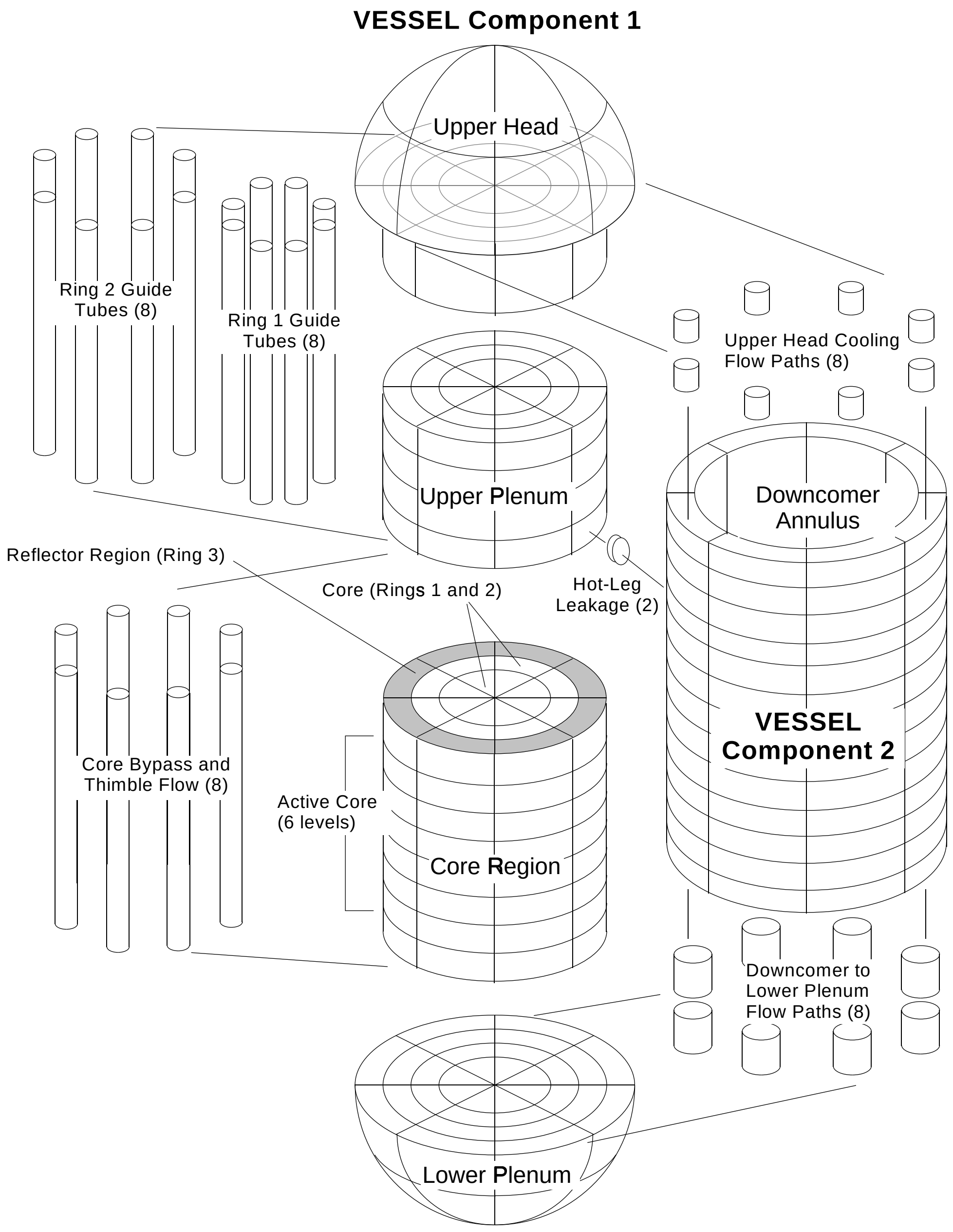


Figure 3.10 Isometric view of AP600 reactor vessel model.

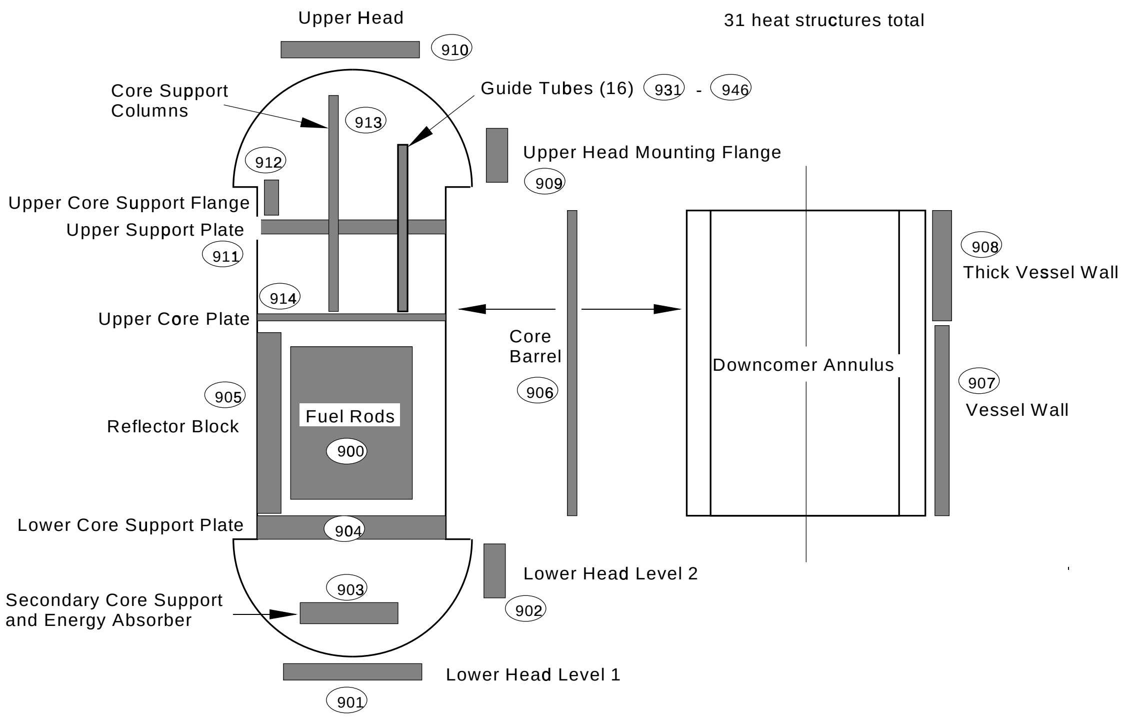


Figure 3.11 AP600 reactor vessel heat structure model.

allocation modification, loop bounds updates, and communication generation. This work uses the regularity of the meshes along with the connectivity between meshes to eliminate the need for user specification of distribution of data. It is important to note that the definition of the High Performance Fortran language does not require the compiler to do interprocedural analysis. Therefore, each subroutine must explicitly specify the distribution for all distributed data structures. This further implies that cloning of routines would be necessary for each type of distribution that a parameter may have. Therefore, for High Performance Fortran, this work eliminates the need for user provided distribution information in subroutines and, hence, cloning.

Automatic distribution begins with the classification of the meshes in ICRM problems according to their size relative to the number of processors being used in the application parallelization. Large meshes can be efficiently distributed over all the processors. Small meshes have to be packed together to provide enough work for a single processor. Medium size meshes are too large to put on a single processor, but too small to distribute over all of the processors. Hence, medium size meshes

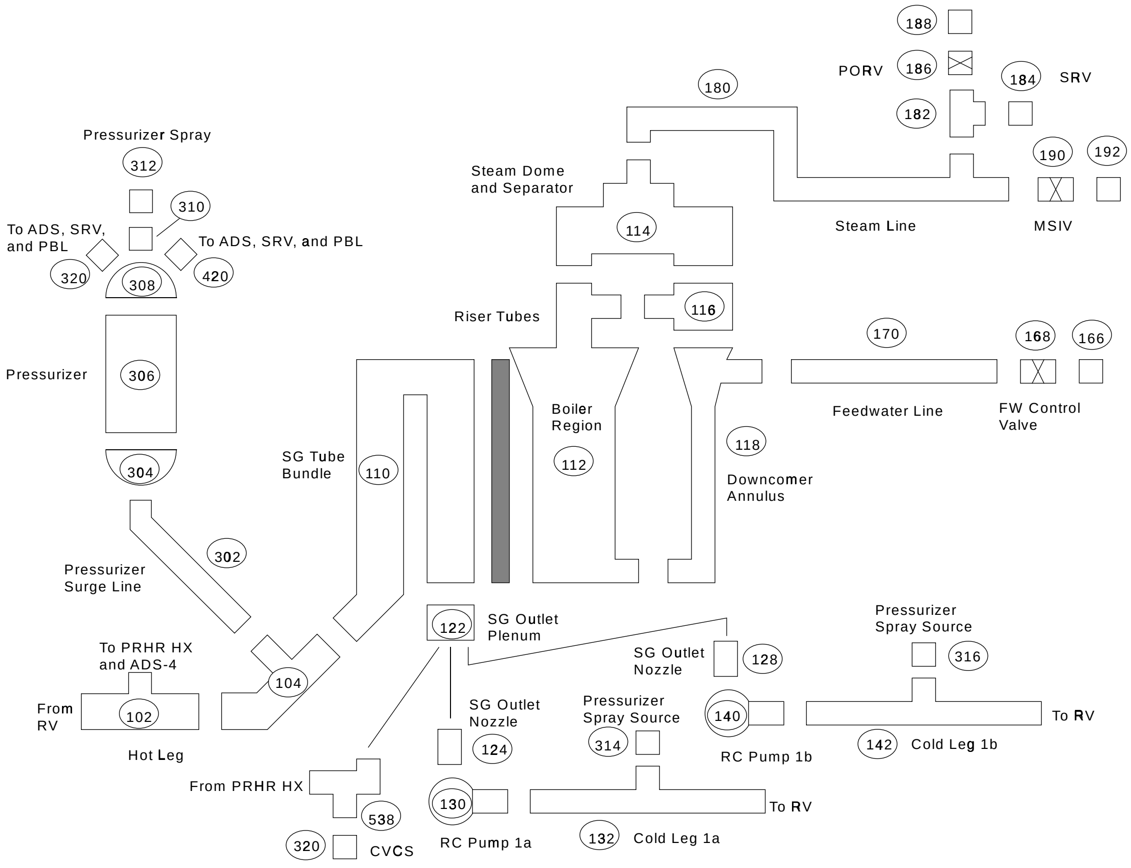


Figure 3.12 AP600 reactor coolant loop 1 model.

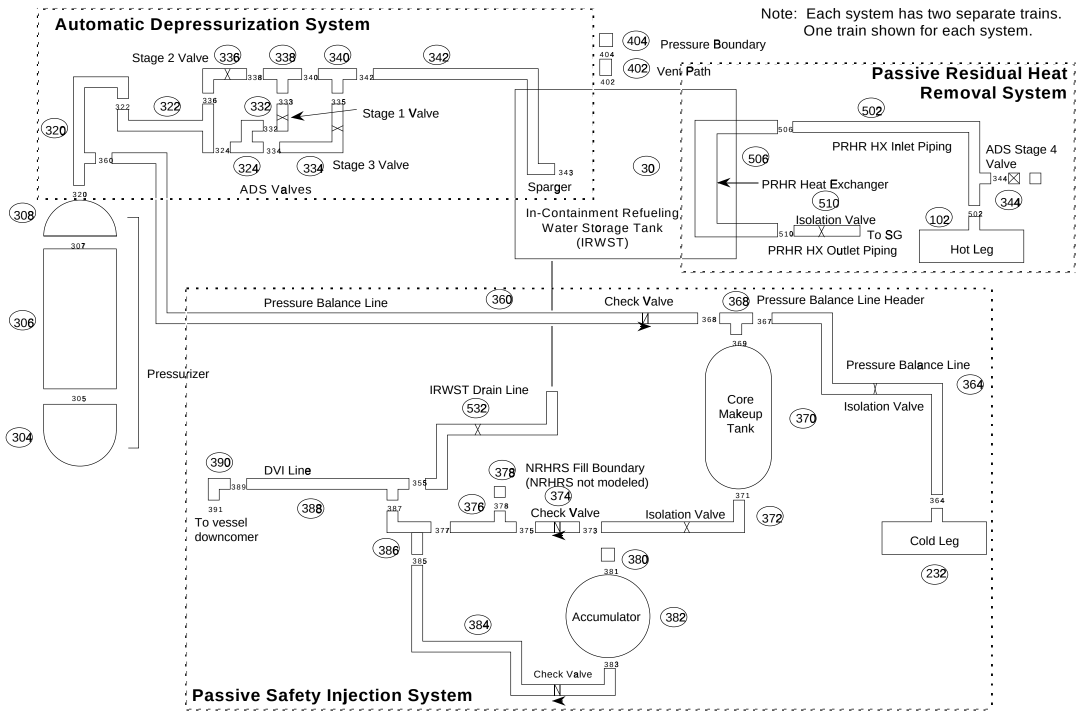


Figure 3.13 AP600 safety systems modeling noding diagram.

must be distributed over a subset of the processors. An algorithm for distribution of each of these mesh classes is presented. To simplify the development of the algorithms, I focus on one mesh size classification at a time. In Chapter 4, problems with only large meshes are considered. In Chapter 5, problems with small meshes are considered. The chapter also discusses how to automatically distribute problems with both large and small meshes. Finally, Chapter 6 develops an algorithm for automatic distribution of medium size meshes, discusses how to automatically distribute problems with all sizes of mesh, and describes the transformation procedure. My automatic distribution system is composed of the transformation procedure that will be discussed in Section 6.3.2 and the automatic distribution algorithms that will be described in Chapters 4–6. In order to use my automatic distribution system, the application program must be written in a specific form.

Programming Style

This section outlines a programming style that is natural to ICRM applications such as aerodynamic or nuclear reactor simulations, and does not inhibit dependence analysis. The guiding principles in determining the programming style that should be used for these applications are:

- the programming style must be natural for the application to reduce development and support costs;
- the programming style must lead to machine independent user programs [Kel94];
- the programming style must be relatively easy to analyze to reduce the cost of compilation and increase the the level of performance the compiler can provide;
- the program structure that the programmer uses should be similar to the program structure that precompiler generates to better support debugging;
- the precompilation should be machine independent to allow its use with the machine dependent compiler for any parallel processor.

This style is described in terms of HPF [KLS+94]. Indeed, from this chapter on, High Performance Fortran is used for all of the code examples. A template for HPF composite grid applications is shown in Figure 3.14.

The template shows the "contains" nesting that is required in the user programs to ensure that the number of processors and mesh sizes are constant at the level that alignment and distribution take place. This is a requirement of the HPF language. Hence the programmer writes a routine, `subroutine main`, that does everything except for the input, which is done in the main program before calling `subroutine main`. By using this two level approach, the need for recompilation based on the input is eliminated and recompilation is only needed when changes are made to the application.

Two of the most important features of a modern language, for the current discussion, are dynamic memory allocation and user-defined data types. The user-defined data types allow all of the arrays and scalars for a given mesh to be grouped into one data structure with the actual storage for the arrays being allocatable at runtime according to the input. Further, an allocatable array of such structures can be used to represent all of the meshes of a given type, i.e., all meshes with the same basic computation (the same component type in reactor simulations) or, more likely, all meshes of the same dimensionality (as used in the abstracted HPF program presented

```
      module composite
C     -- user defined data types go here
C     -- global declarations go here
      contains
      subroutine main
C     -- calls to allocation routines based on input go here
C     -- non-runtime-constant input goes here
C     -- initialization goes here
C     -- all computation goes here
C     -- output goes here
      contains
C     all subroutines except runtime constant input go here
      end subroutine main
C     runtime constant input routines go here
      end module composite

      program icrm
      use module composite
C     -- calls to runtime constant input routines go here
      call main()
      end program icrm
```

Figure 3.14 Template for composite grid HPF programs.

in Chapter 6). For example, in many aerodynamics simulations there would be one allocatable array of structures for all of the meshes surrounding the aircraft as the same basic computation is performed for each mesh. This implies that the program has one allocatable array of structures for each type of mesh or dimensionality. The basic ideas for data structure creation are:

- arrays for each mesh are dynamically allocated;
- all arrays and scalars for each mesh type are grouped into a user defined data structure;
- arrays of user defined structures are allocated dynamically;
- all arrays in user defined structures are to be distributed;
- all arrays in the same user defined structure are distributed the same way; and
- there is an array of user defined data structures for each type (or dimensionality) of mesh.

Associated with each type of mesh there is also a compute routine that is called with each structure on every time step. In addition there must be a routine for internal boundary data exchange for each pair of coupled mesh types. These routines need to be written using a structured programming style; e.g., via the use of `select case` not computed `goto`s.

A few notes about the structure of user defined data types and associated storage allocation. First, all calls for routines that read data that is to be distributed must occur in the main program. The actual allocation for data distribution must occur in `subroutine main`. Further, since HPF does not allow distribution of elements of user defined data structures, the programmer is required to use a pointer to allocate all arrays that are to be distributed and then set the structure array to point to the correct memory. The program could be allowed to use allocatable arrays in the user defined data structure as well, but note that these arrays must be allocatable because the sizes are not available until runtime. Hence the input, allocation, and placement in the program can server as flags to notify the precompiler that these are to be distributed data structures. Then the entire structure for each mesh is passed to any subroutine. The elements of a structure could be passed instead, but that requires more work from the user, and from the compiler, without providing obvious advantages.

Given a program written in this style and an understanding of how the input file relates to the allocation of these structures, the analysis necessary for execution of the automatic distribution algorithm is similar to the interprocedural analysis performed by the Fortran D compiler at Rice University [HHKT92]. This will be discussed further in Chapter 6. Further, this allows the precompiler to be machine independent. The only thing that must be done when the input changes is run the distribution algorithm on the new data set to generate the distribution information in the supplementary input file.

Chapter 4

Mapping Algorithm for Large Mesh Composite Grid Problems

In this chapter, the automatic distribution of ICRM problems that have only large meshes is discussed. The three applications problems shown in Figures 3.1, 3.2, and 3.3 will be used as test problems in validation of the large mesh distribution algorithm developed here.

4.1 Intuition

For any mesh in an ICRM problem to be considered large, the mesh must have enough elements so that distribution across all processors is feasible. This classification provides the basic premise for large mesh distribution. All large meshes will be distributed over all processors. To provide a uniform framework for distribution, the number of dimensions that will be parallel is set to the minimum number of dimensions in any large mesh. This is the number of dimensions that will be distributed in a block fashion for every large mesh.

Many architectures provide communication support for more than one configuration with the specified number of dimensions and the product of the number of processors in the various dimensions equal to the total number of processors to be used. Call this set of configurations the *standard processor configurations* (SPCs).

For each large mesh, the model, described in Appendix A, is used to predict the runtime for one iteration or time-step of computation on the mesh for each mapping of mesh dimensions to processor dimensions. The dimension alignment mapping with the minimum predicted runtime is selected for each standard processor configuration. Next, select the standard processor configuration that minimizes the total predicted runtime. This provides preliminary distribution specification. The meshes could all be mapped to the selected standard processor configuration in a block fashion according to the dimension alignment found above.

Up to this point, no use of the coupling between meshes has been made. Now, the couplings are used to reduce the amount of communication between coupled meshes that is imposed by the distribution. The basic idea for offset alignment is to align the centers of the coupled array sections for the coupling with the most expensive communication associated with each mesh. This results in full distribution specifications.

4.2 Automatic Distribution Algorithm

Minimally, for this algorithm to be used, in addition to the coupling specifications, measures of the following are needed:

- the amount of computation per element in each mesh,
- the amount of communication in each dimension of each large mesh, and
- the amount of communication between each coupled pair of elements in each coupling between meshes.

For most large mesh applications, the nature of the computation is the same on all meshes; the nature of the communication is the same in each dimension; and the coupling cost is the same for each pair of coupled elements. An estimate of these statistics could be provided by the user, but that alone does not eliminate the communication analysis needed in a Fortran D or HPF compiler or a runtime system such as PARTI. To save the user as much work as possible, most of the analysis is moved into the compiler.

This algorithm is to be used only for ICRM applications in which:

- all meshes are large enough to be efficiently distributed over all processors, and
- computations inside of each mesh are regular. (This implies regular communication for each mesh after distribution.)

The distributions generated are targeted to a torus based communications topology. This seems reasonable because most available machines either are tori or can have tori efficiently embedded in the machine topology. For these experiments, application characteristics were provided by the application developers.

I now begin with an overview of the automatic distribution algorithm and then consider each step of the algorithm in more detail. Fortran D modification/generation is also presented in this chapter. The chapter will conclude with a discussion of the limitations and advantages of this approach.

For efficient parallelization of ICRM applications, each mesh should be distributed according to its dimensionality (recall the results of Chapter 2). On the other hand, compilation of the resulting code is easier if the program has uniform memory allocation. Let n be the dimensionality of the mesh with the fewest number of dimensions. All meshes are distributed over all of the processors in an n-dimensional processor topology. Some of the dimensions of higher dimensional meshes will be serialized. In this manner, the algorithm balances the tradeoffs between the computation to communication ratio considerations relating to using the natural topology parallelization for every mesh and the memory allocation issues in SPMD Fortran codes.

The user provides mesh and coupling specifications along with how many processors to use on the target machine. These will normally be runtime input.

The automatic distribution algorithm must take into consideration the user provided information, the program, and the topology of the target machine. Since the number of processors to use on the target machine is fixed, there are only a fixed number of possible m-dimensional processor configurations that can be used, where $1 \leq m \leq$ *maximum processor dimensionality*. This set is limited to only those configurations with at most n dimensions (recall n is the number of dimensions in the mesh with fewest dimensions) and is called the *standard processor configurations*. The final mapping of decompositions will be to the standard processor configuration with the smallest total predicted runtime.

The automatic distribution algorithm performs the following steps.

1. Analysis: The program is analyzed to determine the approximate computation associated with each decomposition, the approximate communication associated with each dimension of each decomposition, and the approximate communication associated with each pair of coupled decompositions. This analysis is part of the precompilation stage in processing. The analysis only needs to be redone when changes are made to the application program. Alternatively, for applications where these computation and communication measures are a single set of numbers, the measures can be input by the user.

2. Table Generation: The best mapping and predicted runtime for each decomposition on each standard processor configuration is found independently, ignoring coupling communication cost. While an exhaustive search could be used to find

the best mapping to each standard processor configuration, my implementation uses heuristics to limit the search space.

3. Global Minimization: The standard processor configuration that produces the minimum total runtime for all decompositions is found, ignoring coupling communication cost. To find the best standard processor configuration, the predicted runtimes for each standard processor configuration are summed over all of the decompositions and the one with the minimum total predicted runtime is selected.

4. Coupling Communication Reduction: The decompositions are aligned with respect to each other in the distribution to reduce coupling communication cost. The heuristic used is to reduce the maximum coupling communication cost for each decomposition via transposing, shifting, or folding around the torus. This is done by aligning the coupled subsections of coupled dimensions of different decompositions.

5. HPF Program Modification: Perform cloning of routines for each possible dimension alignment order, declare variables for alignment order and offsets, add input statements for alignment order and offsets, and insert TEMPLATE, ALIGN, and DISTRIBUTE statements. This will be discussed in Chapter 6.

The distribution decision procedure for selecting a data distribution uses the computational model presented in Appendix A. The model provides expected runtimes based on application and machine parameters as well as the selected data distribution. Any model could be used that will predict approximate runtimes of different mappings on the target machine.

For the presentation of algorithmic details, denote variables associated with a particular decomposition using a *decomposition name* dot *variable name* notation, e.g., $D_m.size[i]$ is the number of elements in the i^{th} dimension of decomposition D_m. For illustrative purposes, two simple composite grid problems are presented. The 3-d CFD configuration[†] is shown in Figure 4.1. This problem consists of two meshes, A and B, each of which are 40x40x40 with one coupled face. The 2-d Multiblock configuration is shown in Figure 4.2. This problem involves a computation associated

[†]This was the test problem used by J. Saltz [CCSR92] to show the feasibility of the PARTI multiblock approach to parallelization of ICRM problems.

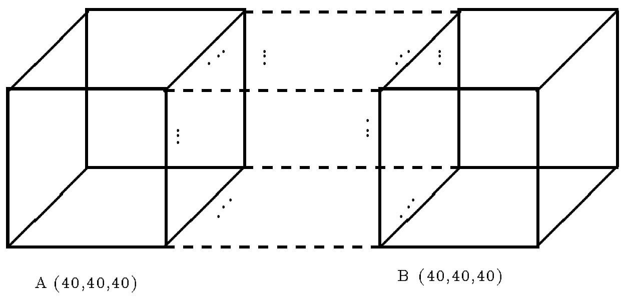


Figure 4.1 3-d CFD Multiblock Configuration.

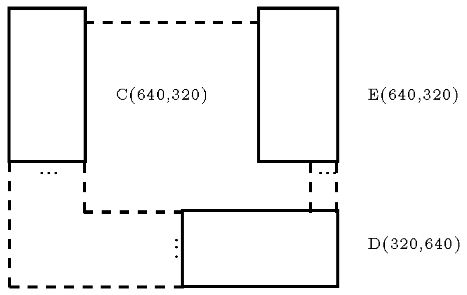


Figure 4.2 2-d Multiblock Configuration.

with the trio of coupled 2-dimensional grids. This problem illustrates some features of the distribution algorithm that are not evident with the first example.

4.2.1 Table Generation

In this first stage, for each standard processor configuration, the model could be evaluated for all mappings of the decomposition's dimensions onto the dimensions of each standard processor configuration. This is an exhaustive search for the best mapping of a decomposition onto each standard processor configuration.

A table of entries is built, with one entry per standard processor configuration. Each entry consists of an assignment of decomposition dimensions to processor dimensions and a predicted runtime. Only one decomposition dimension may be assigned to any processor dimension and all unassigned dimensions are sequentialized.

For each standard processor configuration, all assignments of decomposition dimensions to processor dimensions can be tried and the one with the best predicted runtime is stored. When two mappings produce the same predicted time, one is selected arbitrarily.

In the worst case, for each standard processor configuration, the algorithm considers all partitions of the decomposition's dimensions, d_d, into the processor's dimensions, p_d. There are $\frac{d_d!}{(d_d-p_d)!}$ such partitions.

Since Fortran allows at most seven dimensions in arrays, d_d and p_d are less than or equal to seven. From this, for *all* problems there are most at 7! partitions for any standard processor configuration. More typically, $d_d \leq 4$ and $p_d \leq 3$ so that the number of partitions is less than or equal to 24.

The total number of standard processor configurations with p_d or fewer dimensions is:

$$\begin{aligned}&\eta_{p_d}(n) = \eta_{p_d-1}(n) + \eta_{p_d}(n - p_d)\\&\eta_{p_d}(0) = 1\\&\eta_1(n) = 1\end{aligned}$$

where 2^n is the number of processors. The total number of standard processor configurations is the same as the number of partitions of an integer, n, into p_d or fewer summands [Com74, NZ80]. Consider a machine with 2^{14} processors that is configurable in up to seven dimensions. For such a machine, there would be 105 standard processor configurations. As a more realistic example, for a machine with 2^{14} processors that can be configured with one, two, or three dimensions, the number of standard processor configurations is 24. For this machine, there are 40 evaluations of the model for each decomposition when all meshes are 3-dimensional. Hence, even when all possible mappings are evaluated, there is a constant upper bound on the number of evaluations to be done for each mesh.

Decomposition A		
SPC	Time	Map
(32,1,1)	3483	(2,1,3)
(16,2,1)	2571	(2,3,1)
(8,4,1)	2128	(2,3,1)
(8,2,2)	2119	(2,1,3)
(4,4,2)	2113	(2,3,1)

Decomposition B		
SPC	Time	Map
(32,1,1)	3483	(2,1,3)
(16,2,1)	2571	(2,3,1)
(8,4,1)	2128	(2,3,1)
(8,2,2)	2119	(2,1,3)
(4,4,2)	2113	(2,3,1)

Figure 4.3 3-d CFD Tables

Decomposition C		
SPC	Time	Map
(32,1)	7144	(1,2)
(16,2)	7126	(1,2)
(8,4)	7124	(1,2)

Decomposition D		
SPC	Time	Map
(32,1)	7144	(2,1)
(16,2)	7126	(2,1)
(8,4)	7124	(2,1)

Decomposition E		
SPC	Time	Map
(32,1)	7144	(1,2)
(16,2)	7126	(1,2)
(8,4)	7124	(1,2)

Figure 4.4 2-d Multiblock Tables

The tables generated for the 3-d CFD and 2-d Multiblock examples are shown in Figures 4.3 and 4.4 for the case when 32 processors are used. In the tables, each standard processor configuration (SPC) is specified by the number of elements in each dimension. The mapping (MAP) specifies which dimension of the decomposition is mapped to the corresponding dimension of the standard processor configuration. The predicted runtime (Time) is presented in second. For example, (2,1) specifies that the second dimension of the decomposition is mapped to the first processor dimension and vice versa. In the 3-d CFD problem, the number of bytes of communication is higher in the first dimension than in the other two. This makes it more favorable to use fewer processors in the first dimension than in the others. This leads to a heuristic for reducing the number of mappings that are considered.

Heuristic 1 When one dimension of a decomposition has more communication than the other dimensions, if the standard processor configuration has fewer processors in one dimension than the other dimensions, then map the decomposition dimension having greater communication to the processor dimension with fewer processors.

In the 2-d Multiblock problem, the number of bytes of communication is the same in each dimension of each mesh. This makes the number of elements to be communicated be the only communication factor that influences the choice of mapping. Surface to volume effects [LK94b] imply that each communication "edge" should be the same length. This leads to the second heuristic for reducing the number of evaluations of the model.

Heuristic 2 When every dimension of a decomposition has the same amount of communication, if the standard processor configuration has fewer processors in one dimension than the other dimensions, then map the decomposition dimension having the fewest elements to the processor dimension with the fewest processors.

Finally, consider the case when two dimensions of a mesh are the same size and have the same amount of communication, e.g., the second and third dimension in the 3-d CFD example. This consideration leads to the final heuristic.

Heuristic 3 When two dimensions of a mesh have the same size and communication, for each pair of mappings with these dimensions interchanged only one of the mappings needs to be evaluated. Pairs of equivalent dimensions are recorded for possible use during coupling communication reduction.

In practice, the heuristics are applied only when the number of elements in each dimension of the decomposition is divisible by the number of processors in each dimension. With this qualification, the heuristics usually produce the same results as the exhaustive search. The only exception to this is caused by the use of the last heuristic, where the heuristic provides better performance because it allows interchange and the exhaustive search does not.

4.2.2 Global Minimization

In the global minimization step, the standard processor configuration that will produce the best runtime for the entire problem is selected. To do this, a sum over the predicted runtimes in the decomposition tables for each SPC is found and the standard processor configuration with minimum total predicted runtime is selected. At this point, the optimal mapping has been found if there is no communication between the different decompositions.

Next, create a new standard processor configuration table with each entry initialized to zero. Iterating over all of the standard processor configurations, for each decomposition's table, add the decomposition's predicted runtime to the total runtime for the standard processor configuration. The minimum value in the total runtime table is indexed by the best standard processor configuration.

This stage of the algorithm takes time on the order of the number of standard processor configurations times the number of decompositions. This step of the procedure puts the greatest accuracy requirements on the model because predicted runtimes are added. For a discussion of the model of parallel computation used in this research, refer to Appendix A.

For the 3-d CFD problem, the best standard processor configuration using 32 processors is (4,4,2) and for the 2-d Multiblock problem it is (8,4).

4.2.3 Coupling Communication Reduction

The purpose of coupling communication reduction is to use the knowledge of topology to reduce communication costs associated with coupling. The heuristic used attempts to minimize the maximum cost coupling communication for each decomposition subject to the dimension mapping already selected. This is done by aligning the centers of the coupled subranges of dimensions and setting directions (increasing or decreasing) to be the same for each processor dimension. Further, if interchanging two equivalent dimensions will reduce coupling cost, the interchange is performed.

For each dimension of every decomposition, a starting processor index and a direction is selected. The processor index (between one and the number of processors in the processor dimension that the decomposition's dimension is mapped to) specifies which processor contains the first block of elements for the given dimension of the decomposition. The direction, (+1 or -1), specifies whether the elements in the decomposition's dimension will be mapped in increasing or decreasing order.

Begin by selecting the decomposition with the single highest coupling communication cost and call it D_1. Set the processor index for D_1 to 1 and the direction to increasing in every parallel dimension. D_1 is now completely mapped to the best standard processor configuration. Make D_1 the only element in the set MAPPED. Create a max heap of all couplings involving D_1 with order determined by the coupling costs.

The following steps are repeated until all decompositions are in the set MAPPED.

1. Delete the maximum element from the coupling heap.

2. If both decompositions in the coupling are in MAPPED, then return to step 1. Otherwise, call the unmapped decomposition in the coupling D_u and call the mapped decomposition D_m.

3. If interchanging equivalent dimensions of D_u decreases coupling cost, then do it.

4. Consider each pair of dimensions, where dimension $D_m.i$ is the dimension of D_m coupled to dimension k of D_u and the dimensions *are* both mapped to

the same processor dimension. Related to coupling specification for dimension i of D_m are $D_m.start, D_m.end$, and $D_m.stride$. Related to dimension k of D_u are $D_u.start, D_u.end$, and $D_u.stride$. Since D_m is already mapped, the $D_m.direction$ and $D_m.start\_proc$ for each dimension are already defined. To map dimension k of D_u, its direction and starting processor must be found.

- The direction for dimension k of D_u is the same as that of dimension i of D_m if the stride for the coupling for each decomposition is either increasing or decreasing. If one decomposition's stride is increasing and the other decomposition's stride is decreasing, then the direction for D_u is the opposite of the direction for D_m.
- Determine the offset of the processor that owns the middle entry of the coupling in dimension i of D_m, call it $D_m.Poffset$.
- Determine the offset of the processor that owns the middle entry of the coupling in dimension k of D_u, call it $D_u.Poffset$ (computed as if D_u was mapped beginning in processor 1).
- Determine the processor that should own the first element of dimension k of D_u. It is $D_m.start\_proc[D_m.i]$ + $D_m.Poffset$ - $D_u.Poffset$.

5. For each pair of coupled dimensions that are *not* mapped to the same processor dimension, the direction and starting processor are found for D_u as follows, where k is the dimension of D_u being mapped.

   - The direction for D_u in dimension k is positive (this is arbitrary).
   - Find the dimension of D_m that is mapped to the same processor dimension as dimension k of D_u, label it $D_m.i$.
   - Let j be the index in the coupling specification that involves dimension $D_m.i$ of D_m.
   - Determine the offset of the processor that owns the middle element of the j^{th} coupling entry for D_m, call it $D_m.Poffset$.
   - Determine the offset of the processor that owns the middle entry of the coupling in dimension k of D_u, call it $D_u.Poffset$ (computed as if D_u was mapped beginning in processor 1).
   - Determine the processor that should own the first element of dimension k of D_u. It is $D_m.start\_proc[D_m.i]$ + $D_m.Poffset$ - $D_u.Poffset$.

6. For each coupling involving D_u, which has not already been put into the heap, insert it.

7. Add D_u to the set MAPPED.

Decomposition	Direction	Starting Processor
A	(1,1,1)	(1,1,1)
B	(1,1,1)	(1,1,1)

Figure 4.5 3-d CFD Directions and Starting Processors

Decomposition	Direction	Starting Processor
C	(1,1,1)	(8,1,1)
D	(1,1,1)	(1,1,1)
E	(1,1,1)	(1,1,4)

Figure 4.6 2-d Multiblock Directions and Starting Processors

The time for this stage of the algorithm is bounded by the number of distributed dimensions multiplied by $c\ log\ c$, where c is the number of couplings in the system $(c \geq\ number\ of\ meshes - 1)$.

Figures 4.5 and 4.6 show the directions and starting processors that result from the application of coupling communication reduction for the 3-d `CFD` and 2-d Multiblock problems respectively.

4.2.4 Limitations -vs- Advantages

This approach to ICRM parallelization is limited to those problems for which it makes sense to distribute every mesh over all processors. This limits the use of this approach for some problems. For example, in many Nuclear Reactor Simulations [BSL85] many of the meshes have few elements ($\sim 10 - 100$) and hence cannot reasonably be distributed over many processors. Further, if enough processors are used, even large meshes can not be distributed over all processors. Chapter 6 presents an approach to dealing with this situation.

This limitation does not say that every mesh must have many more elements than processors. The real requirement is that for every processor the computation outweighs the communication. Hence, this approach may work even when some of the meshes have about the same number of elements as the number of processors.

Offsetting these limitations are a number of advantages to using this approach. Unlike other currently available approaches, all analysis and communication generation may be performed at compile-time. To do this, runtime constant data (coupling information) is required. Compile-time analysis and communication generation may provide a significant savings in time when the same mesh/coupling configuration is used with different initial data sets. For example, the same multiblock representation of an aircraft may be used for simulation of the flow over the aircraft with many different initial conditions. Furthermore, once cloning has been done and the HPF program is generated and compiled, the algorithm can be applied to different configurations without recompilation.

Not only does this approach result in a program that fits the SPMD model, but the code also fits a uniform memory model. Each processor uses (approximately) the same amount of memory for each array and the declarations for each processor are identical. When different decompositions are assigned to different processors, the memory allocation and access may be complicated. This uniform memory model conformance implies that the approach is easy applicable on SIMD architectures such as the SNAP-32[‡]

In some application codes, it may be difficult or even impossible to automatically verify that the computations associated with all of the different meshes can be executed in parallel. In this case, even if the decompositions are assigned to different processors, their computations will be executed sequentially. The approach presented here does not rely on being able to prove this high level parallelism, but instead relies on the parallelism inside each mesh.

Compiling these ICRM applications with a standard HPF compiler is straight forward. In addition, optimization developed for regular applications can be applied. The only extension that might be helpful in a good HPF compiler is using the coupling specifications as an upper bound for communication associated with couplings.

[‡]Robert Means, Bret Wallach, David Busby, and Robert Lengel Jr. won the 1993 Gordon Bell Prize for price/performance using a SIMD Numerical Array Processor (SNAP) with 32 processors. The price/performance index was 7,554 flops per dollar. This indicates that SIMD machines may still have a future with some applications.

The final and possibly most important advantage is that, for the first time, the distribution can be found automatically for this class of complex topology problems.

4.2.5 Fortran D Modification/Generation

In this section, I assume that a Fortran D program that includes DECOMPOSITION and ALIGN statements for each mesh was provided by the user. This section therefore applies only when there is a fixed topology encoded in the program and not when the topology is part of the input. This limitation is due primarily to the fact that Fortran D does not support the `contains` statement and modules. Without this the processor, alignment, and distribution specification in the input are not constant on entry to the module entry level. Hence, this section applies only to programs with statically encoded topology. The modification of DECOMPOSITION and ALIGN statements and the generation of DISTRIBUTE statements are discussed next. The final output of these modification/generation steps will be a standard Fortran D program. Since the Fortran D compiler needs to perform the cloning procedure automatically, the precompiler is not required to perform cloning when working with Fortran D. Therefore, the precompiler just modifies decomposition and alignment specifications at the levels given by the programmer and adds distribution specifications at the same levels. At some point the Fortran D language will probably be extended to include `contains`. The original ALIGN and DISTRIBUTE specifications for the 3-d `CFD` problem are:

```
decomposition A(40,40,40), B(40,40,40)
align wa(i,j,k) with A(i,j,k)
align xa(i,j,k) with A(i,j,k)
align ya(i,j,k) with A(i,j,k)
align za(i,j,k) with A(i,j,k)
align wb,xb,yb,zb with B
```

The original ALIGN and DISTRIBUTE specifications for the 2-d Multiblock problem are:

```
decomposition C(640,320), D(320,640), E(640,320)
align vc(i,j) with C(i,j)
align wc(i,j) with C(j,i)
align xc(i,j) with C(i-1,j-1)
align yc(i,j) with C(i-1,j+1)
align zc(i,j) with C(i+1,j+1)
align vd,wd,xd,yd,zd with D
align ve,we,xe,ye,ze with E
```

Decomposition Modification

The original DECOMPOSITION statements must be replaced with new ones that reflect the intended mapping of data to processors. To reflect this, the precompiler reorders the dimensions according to the mapping. Hence, for each decomposition D the specification

$$\text{DECOMPOSITION } D(size_1, size_2, size_3, ..., size_{D.dimensions})$$

becomes

$$\text{DECOMPOSITION } D(size_{map^{-1}(1)}, size_{map^{-1}(2)}, ...size_{map^{-1}(D.dimensions)})$$

where map^{-1} is the inverse of the map that was found for decomposition D.

For the 3-d CFD problem the modified decomposition declarations are:

$$\text{DECOMPOSITION } A(40, 40, 40), B(40, 40, 40)$$

For the 2-d Multiblock problem the modified decomposition declarations are:

$$\text{DECOMPOSITION } C(640, 320), D(640, 320), E(640, 320)$$

Alignment Modification

Next, ALIGN statements are transformed according to the mapping to processors, alignment to processors, and the original alignment offsets. For each ALIGN statement associated with a decomposition D

$$\begin{aligned} \text{ALIGN} \quad & X(I_1, I_2, I_3, I_4, ..., I_{D.dimensions}) \\ \text{WITH} \quad & D(I_1 + \mathit{offset}_1, I_3 + \mathit{offset}_3, I_2 + \mathit{offset}_2, I_4 + \mathit{offset}_4, ..., \\ & \quad I_{D.dimensions} + \mathit{offset}_{D.dimensions}) \end{aligned}$$

becomes

$$\begin{array}{ll}
\text{ALIGN} & X(\text{WRAP}((1 - D.direction[1]) * (-\tfrac{1}{2}) * D.size[1] + D.direction[1] * (I_1)), \\
& \quad \text{WRAP}((1 - D.direction[2]) * (-\tfrac{1}{2}) * D.size[2] + D.direction[2] * (I_2)), \\
& \quad \ldots \\
& \quad \text{WRAP}((1 - D.direction[D.dimensions]) \\
& \qquad\qquad *(-\tfrac{1}{2}) * D.size[D.dimensions] \\
& \qquad\qquad +D.direction[D.dimensions] * (I_{D.dimensions}))) \\
\text{WITH} & D(I_{map^{-1}(1)} + D.elt\_per\_proc[map^{-1}(1)] * (start\_proc[map^{-1}(1)] - 1) \\
& \qquad +offset_{map^{-1}(1)}, \\
& \quad I_{map^{-1}(3)} + D.elt\_per\_proc[map^{-1}(3)] * (start\_proc[map^{-1}(3)] - 1) \\
& \qquad +offset_{map^{-1}(3)}, \\
& \quad I_{map^{-1}(2)} + D.elt\_per\_proc[map^{-1}(2)] * (start\_proc[map^{-1}(2)] - 1) \\
& \qquad +offset_{map^{-1}(2)}, \\
& \quad I_{map^{-1}(4)} + D.elt\_per\_proc[map^{-1}(4)] * (start\_proc[map^{-1}(4)] - 1) \\
& \qquad +offset_{map^{-1}(4)}, \ldots, \\
& \quad I_{map^{-1}(D.dimensions)} + D.elt\_per\_proc[map^{-1}(D.dimensions)] \\
& \qquad *(start\_proc[map^{-1}(D.dimensions)] - 1) \\
& \qquad +offset_{map^{-1}(D.dimensions)})
\end{array}$$

For the 3-d CFD problem the alignment that results is:

$$\begin{array}{l}
\text{ALIGN } wa(wrap(i), wrap(j), wrap(k)) \text{ WITH } A(j,k,i) \\
\text{ALIGN } xa(wrap(i), wrap(j), wrap(k)) \text{ WITH } A(j,k,i) \\
\text{ALIGN } ya(wrap(i), wrap(j), wrap(k)) \text{ WITH } A(j,k,i) \\
\text{ALIGN } za(wrap(i), wrap(j), wrap(k)) \text{ WITH } A(j,k,i) \\
\text{ALIGN } wb(wrap(i), wrap(j), wrap(k)) \text{ WITH } B(j,k,i) \\
\text{ALIGN } xb(wrap(i), wrap(j), wrap(k)) \text{ WITH } B(j,k,i) \\
\text{ALIGN } yb(wrap(i), wrap(j), wrap(k)) \text{ WITH } B(j,k,i) \\
\text{ALIGN } zb(wrap(i), wrap(j), wrap(k)) \text{ WITH } B(j,k,i)
\end{array}$$

For the 2-d Multiblock problem the alignment that results is:

$$\begin{array}{l}
\text{ALIGN } vc(wrap(i), wrap(j)) \text{ WITH } C(i + 80 * 7, j) \\
\text{ALIGN } wc(wrap(i), wrap(j)) \text{ WITH } C(j, i + 80 * 7) \\
\text{ALIGN } xc(wrap(i), wrap(j)) \text{ WITH } C(i + 80 * 7 - 1, j - 1) \\
\text{ALIGN } yc(wrap(i), wrap(j)) \text{ WITH } C(i + 80 * 7 - 1, j + 1) \\
\text{ALIGN } zc(wrap(i), wrap(j)) \text{ WITH } C(i + 80 * 7 + 1, j + 1) \\
\text{ALIGN } vd(wrap(i), wrap(j)) \text{ WITH } D(j, i) \\
\text{ALIGN } wd(wrap(i), wrap(j)) \text{ WITH } D(j, i) \\
\text{ALIGN } xd(wrap(i), wrap(j)) \text{ WITH } D(j, i) \\
\text{ALIGN } yd(wrap(i), wrap(j)) \text{ WITH } D(j, i) \\
\text{ALIGN } zd(wrap(i), wrap(j)) \text{ WITH } D(j, i) \\
\text{ALIGN } ve(wrap(i), wrap(j)) \text{ WITH } E(i, j + 80 * 3) \\
\text{ALIGN } we(wrap(i), wrap(j)) \text{ WITH } E(i, j + 80 * 3) \\
\text{ALIGN } xe(wrap(i), wrap(j)) \text{ WITH } E(i, j + 80 * 3) \\
\text{ALIGN } ye(wrap(i), wrap(j)) \text{ WITH } E(i, j + 80 * 3) \\
\text{ALIGN } ze(wrap(i), wrap(j)) \text{ WITH } E(i, j + 80 * 3)
\end{array}$$

In HPF, the alignments must be shifted as HPF does not support wrap alignment.

Distribution Generation

Finally, DISTRIBUTE statements are generated for each decomposition D of the form

$$\text{DISTRIBUTE } D(\text{BLOCK}(SPC.procs[1]), \text{BLOCK}(SPC.procs[2]), \ldots$$
$$\text{BLOCK}(SPC.procs[D.dimensions]))$$

where $SPC.procs[i]$ is set to one for $i > SPC.dimensions$.

For the 3-d CFD problem the distribution that results is:

$$\begin{array}{l}
\text{DISTRIBUTE } A(\text{BLOCK}(4), \text{BLOCK}(4), \text{BLOCK}(2)) \\
\text{DISTRIBUTE } B(\text{BLOCK}(4), \text{BLOCK}(4), \text{BLOCK}(2))
\end{array}$$

For the 2-d Multiblock problem the distribution that results is:

$$\begin{array}{l}
\text{DISTRIBUTE } C(\text{BLOCK}(8), \text{BLOCK}(4)) \\
\text{DISTRIBUTE } D(\text{BLOCK}(8), \text{BLOCK}(4)) \\
\text{DISTRIBUTE } E(\text{BLOCK}(8), \text{BLOCK}(4))
\end{array}$$

4.3 Algorithm Validation

The two primary goals that I am trying to achieve with this algorithm are: 1) reduce the programmer burden for parallelization of ICRM problems, and 2) support regular problem optimization in the compiler via program transformation, as will be described in Section 6.3, in order to provide acceptable performance in the resulting parallelization.

Since there is not yet an HPF compiler that can correctly generate code for these applications, measures of load balance and communication are presented in place of timing results. The measure of load balance presented is the total number of floating point additions, multiplications, and divisions. This load balance measure is compared for the processor with the most work ($Comp_{max}$) and the processor with the least work ($Comp_{min}$). The following measures of communication are presented:

- *distance* is the maximum distance between any pair of communicating processors;
- *neighbors* is the maximum number of processors communicated with for any processor;
- $Comm_{max_1}$ is the maximum amount of communication, in bytes, for any process3or;
- $Comm_{max_2}$ is the maximum communication, in bytes, between any pair of processors; and
- $Comm_{total}$ is the total communication, in bytes, for the simulation.

In the tables of results, the selected processor configuration is shown below the number of processors.

The speedup bound, $Dist_{sb}$, is the speedup, with the data distribution supplied, that would be achieved if there were no communication, i.e.,

$$Dist_{sb} = \frac{Comp_{max}\ for\ n\ processors}{Comp_{max}\ for\ one\ processor}.$$

The speedup bound is rounded to two decimal places.

4.3.1 Simulation of Material Flow in an Elbow with Vanes

The Elbow problem, shown in Figure 3.1, has five 3-dimensional meshes with a total of 275,356 3-dimensional cells. The results of experiments for this problem are shown in Table 4.7.

Note that while the processor utilization drops off as more processors are added, the speedup bound, $Dist_{sb}$, does continue to increase. The large increase in the total communicationin going from 128 processors to 256 processor for the Elbow problem is due to the increase in $Comm_{max_2}$. The increase in $Comm_{max_2}$ comes about

	Number of Processors				
	2 (2)	4 (2x2)	8 (2x2x2)	16 (4x2x2)	32 (4x4x2)
$distance$	1	1	1	1	2
$neighbors$	1	2	3	4	7
$Comm_{max_1}$	161,696	205,128	147,552	167,664	109,314
$Comm_{max_2}$	161,696	123,000	61,440	61,440	30,216
$Comm_{total}$	161,696	410,028	585,192	1,326,264	1,674,224
$Comp_{min}$	5,231,764	2,615,882	1,276,040	593,560	258,704
$Comp_{max}$	5,231,764	2,615,882	1,339,842	685,482	354,882
$Dist_{sb}$	2.00	4.00	7.81	15.26	29.48

	Number of Processors				
	64 (8x4x2)	128 (16x4x2)	256 (32x4x2)	512 (32x4x4)	1024 (16x16x4)
$distance$	3	4	8	8	3
$neighbors$	9	9	8	9	8
$Comm_{max_1}$	66,399	47,013	44,872	23,036	9,933
$Comm_{max_2}$	19,416	14,472	19,656	9,792	2,280
$Comm_{total}$	2,108,444	2,990,154	5,674,404	5,828,004	5,062,386
$Comp_{min}$	80,256	7,600	0	0	0
$Comp_{max}$	185,592	96,786	50,274	26,334	14,022
$Dist_{sb}$	56.38	108.11	208.13	397.34	746.22

Figure 4.7 Validation results from flow through an elbow with 2 vanes.

because the interface between adjacent processors in the third processor dimension is significantly larger for 256 processors than for 128 processors. This interface size has increased due to the change in mapping decomposition dimensions to processor dimensions. The reverse situation occurs in the same problem when going from 512 processors to 1024 processors.

4.3.2 Simulation of Aerodynamics over Fuselage-Inlet-Nozzle of an F15e Aircraft

The Fuselage-Inlet-Nozzle aerodynamic simulation, shown in Figure 3.2, has ten 3-dimensional meshes with a total of 713,766 3-dimensional cells. The results of experiments for this problem are shown in Table 4.8. Notice that the number of processors

	Number of Processors				
	2 (2)	4 (2x2)	8 (4x2)	16 (4x2x2)	32 (4x4x2)
$distance$	1	2	3	4	4
$neighbors$	1	3	7	11	20
$Comm_{max_1}$	241,652	316,004	252,254	259,010	212,637
$Comm_{max_2}$	241,652	159,012	85,776	80,328	55,080
$Comm_{total}$	241,652	630,546	997,068	2,043,732	3,351,132
$Comp_{min}$	13,561,554	6,757,464	3,337,692	1,621,878	741,000
$Comp_{max}$	13,561,554	6,804,090	3,414,984	1,750,242	888,288
$Dist_{sb}$	2.00	3.99	7.94	15.50	30.53

	Number of Processors				
()	64 (8x4x2)	128 (8x4x4)	256 (32x4x2)	512 (32x8x2)	1024 (32x8x4)
$distance$	5	4	19	20	19
$neighbors$	23	18	25	29	24
$Comm_{max_1}$	134,711	103,386	72,567	38,822	28,404
$Comm_{max_2}$	33,068	21,300	26,544	13,776	7,000
$Comm_{total}$	4,205,268	6,224,450	9,201,546	9,783,744	14,118,076
$Comp_{min}$	271,434	120,574	0	0	0
$Comp_{max}$	454,518	233,016	117,686	60,306	30,856
$Dist_{sb}$	59.67	116.40	230.47	449.76	879.02

Figure 4.8 Validation results from F15e Fuselage-Inlet-Nozzle aerodynamic simulation.

in the last dimension is reduced by a factor of 2 in the step from 128 processors to 256 processor. The increase in $Comm_{max_2}$ is due to this change in the processor topology.

4.3.3 Simulation of Aerodynamics over a Complete F15e Aircraft

The full F15e aerodynamic simulation, shown in Figure 3.3, has 32 3-dimensional meshes with a total of 1,269,845 3-dimensional cells. The results of experiments for this problem are shown in Table 4.9.

	Number of Processors				
	2 (2)	4 (4)	8 (4x2)	16 (8x2)	32 (16x2)
$distance$	1	2	3	5	9
$neighbors$	1	3	7	15	29
$Comm_{max_1}$	809,432	1,009,132	830,304	464,802	376,244
$Comm_{max_2}$	809,432	504,100	302,816	161,846	135,192
$Comm_{total}$	809,432	2,018,064	3,295,830	3,670,234	5,991,572
$Comp_{min}$	24,043,018	11,710,954	5,779,230	2,269,056	964,136
$Comp_{max}$	24,211,092	12,181,052	6,152,656	3,150,428	1,589,578
$Dist_{sb}$	1.99	3.96	7.84	15.32	30.36

	Number of Processors				
	64 (16x4x1)	128 (16x4x2)	256 (16x8x2)	512 (16x8x4)	1024 (32x8x4)
$distance$	10	11	12	13	20
$neighbors$	56	83	110	115	114
$Comm_{max_1}$	293,874	229,931	118,518	84,483	56,260
$Comm_{max_2}$	77,640	46,128	25,752	14,952	11,392
$Comm_{total}$	9,353,564	14,419,578	14,968,986	21,278,067	28,141,528
$Comp_{min}$	428,184	214,054	65,664	27,626	5,434
$Comp_{max}$	811,642	415,720	218,500	114,000	59,850
$Dist_{sb}$	59.45	116.07	220.84	423.28	806.25

Figure 4.9 Validation results from full F15e aerodynamic simulation.

$Comm_{max_1}$ decreases as the number of processors increases (after 4 processors). The increase from 2 to 4 processors is due to having two neighbors with which to communicate instead of one. $Comm_{max_2}$, $Comp_{min}$, and $Comp_{max}$ are monotonically decreasing. The total communication, $Comm_{total}$, and the speedup bound, $Dist_{sb}$, are monotonically increasing.

4.4 Chapter Summary

An algorithm that automatically finds a distribution of data in ICRM problems given a well-structured HPF program and topological connection specifications has been presented. When meshes are generated automatically, automatic distribution of large-mesh ICRM problems is critical. This algorithm provides an important first step.

Chapter 5

Mapping Algorithm for Small Mesh Composite Grid Problems

In this chapter, the automatic distribution of ICRM problems that use only large and small meshes is discussed. Two random distribution algorithms are introduced for use as a basis of comparison for the algorithm in the experimental results of this chapter.

5.1 Intuition

For any mesh in an ICRM problem to be considered small, the mesh must have few enough elements so it has at most enough work to fill a single processor. This classification provides the basic premise for small mesh distribution. All small meshes will be packed into the processors trying to achieve load balance while using the problem topology to put meshes with the most expensive coupling on the same processor when possible.

5.2 Automatic Distribution Algorithm

Minimally, for the algorithm being presented, the following are needed (in addition to the information in coupling specifications):

- the amount of computation per element in each mesh,
- the amount of communication in each dimension of each large mesh, and
- the amount of communication between each coupled pair of elements in each coupling between meshes.

The coupling specifications are used to determine which meshes must communicate. The amount of computation for each element of each mesh is used to attempt to balance the computational load. The amount of communication in each dimension of each large mesh is used to attempt to minimize the cost of communication for distribution of the large mesh. The amount of communication for each coupling is used to prioritize the mapping order of coupled meshes and to determine the placement of coupled meshes. In nuclear reactor simulations, these measures may vary by component type, but the nature of the computation carried out for each mesh of the same type is similar. The computation can also vary by cell according to the material in the cell and the phase of the material.

For this chapter, only composite grid applications with the following properties are considered.

- All of the meshes are either large enough to be efficiently distributed over all processors or small enough to be assigned to a single processor.
- Computations inside of each mesh are regular (this implies regular internal communication for each mesh for each distribution generated by these algorithms).

A torus based communications topology is being targeted. This seems reasonable as many available machines either are tori or can have them efficiently embedded in the machine topology.

First, descriptions of both of the random distribution algorithms and the topology-based small mesh distribution algorithm will be presented. This section ends with a discussion of the limitations and advantages of the topology-based small mesh distribution approach.

For efficient parallelization of composite grid applications, each mesh should be distributed according to its dimensionality [LK94b]. Hence, all of the large meshes are distributed over all of the processors in the same dimensionality as discussed in [LK94a]. During the mapping of the large meshes, a processor configuration is selected for distribution of the entire problem. When there are no large meshes in the problem, m-dimensional processor topology closest to being "square" is selected, where m is the dimensionality of the largest dimensionality processor mesh of the specified size on the host architecture.

5.2.1 Random Algorithm

For each mesh, a pseudorandom number is used to select the processor that will own the mesh. No restriction is placed on assignment of meshes to processors. This approach has been advocated by applications developers. This algorithm takes $O(M)$ time to generate a distribution specification, where M is the number of small meshes.

5.2.2 Load-Balanced Random Algorithm

For a load-balanced random distribution, the meshes are randomly distributed, as above, but they are distributed in highest computation cost first order. Further, there is an upper bound on the amount of computation that may be allocated to a processor. This bound is overridden only after a number of failures to allocate. The number of failures is set to be on the order of the number of processors. Therefore, to map the largest unmapped mesh, a processor is randomly selected. If the processor can accommodate the mesh's computation, then map the mesh to the processor, otherwise try again up to some set number of times. If no processor with enough room is found, just assign the mesh to a processor because, even if the load balance is poor, the problem must be solved. This approach produces better worst case load balance than the random algorithm but is more expensive. This algorithm takes $O(P\ M\ log(M))$ time due to the sorting of the meshes by computational cost and the possible number of failures, where M is the number of small meshes and P is the number of processors.

5.2.3 Topoloby-based Small Mesh Algorithm

Next, a brief description of the topology-based small mesh distribution algorithm is presented. The associated data structures are described and an outline of the steps in the algorithm is shown in Figures 5.1 and 5.2.

It is assumed that before this algorithm begins any large meshes have been mapped. As a result of this mapping, a processor configuration is chosen. Further, all communications involving large size meshes are scheduled. If there are no large meshes, then 1) the largest dimensionality nearly square processor configuration possible is chosen, 2) the highest computation cost small mesh is mapped to an arbitrary processor, and 3) all communications to this mapped small mesh are scheduled.

The basic idea used in this algorithm is that meshes that communicate should be mapped to the same processor or to processors as close together as possible. If meshes were simply mapped to the processor with the greatest coupling cost, there would be few processors with small meshes allocated to them. The number of processors used in that case would be bounded by the number of couplings between large and small meshes (one in the case of a single premapped small mesh). Therefore, a load-balance measure is used to determine when a processor has too much computation already allocated to accommodate the work associated with an unmapped mesh. When this happens, the algorithm tries to allocate the mesh to a neighbor processor. If all of the neighbors are too full, the algorithm finds the closest processor with no work allocated. If this also fails, the mesh is saved as a “fill” element to be used after all other meshes are mapped. These “fill” elements are then used in a final load balancing step by assigning them to under-utilized processors.

Small mesh distribution begins by determining the amount of computation, for the small meshes only, that should be assigned to each processor for perfect small mesh load balance. Small meshes are statically distributed by trying to obtain load balance within $\pm\epsilon$ of perfect balance. If the large meshes are not perfectly load balanced, then their computation cost could be added to this consideration to improve the overall load balance. This is not currently being done, but will be in future work. This algorithm could also be used at runtime for dynamic load balancing if computation/communication statistics are collected in the program and the new alignment/distribution specifications are used for data redistribution. This may be useful for problems where the amount of computation per cell changes dramatically over time (e.g., reactor simulation with phase changes).

Two types of heaps are used in this mapping algorithm. The *mesh heap* contains all of the unmapped meshes that have scheduled communication. Communication is scheduled, between a mapped mesh, α, and each coupled unmapped mesh, β, by adding the coupling communication cost to β's communication for the processor to which α is mapped. This introduces the second type of heap. A *processor heap* is associated with each mesh in the *mesh heap*. Each entry in a *processor heap* gives the current coupling cost of the associated mesh to that specific processor. From this two level structure the mesh with highest computation cost and the processor that it is coupled to are found. First, from the *mesh heap*, the mesh is found with the maximum communication scheduled to a single processor. Then, from the mesh's *processor heap*, the associated processor is found. In the final cleanup stage of the algorithm, two heaps are also used, one being a maximum heap of meshes sorted according to the amount of computation in the mesh and the other being a minimum heap of processors sorted according to the amount of computation assigned to the processor.

For each mapped large mesh, all communications to small meshes are scheduled initially. Alternatively, the small mesh with the most computation may be mapped to a processor and all communication to other small meshes is scheduled. The details of the topology-based algorithm are now presented. Figure 5.1 gives an outline of the

```
add M's computation to P's computation
delete M's processor heap
For each unscheduled coupling involving:
        the mapped mesh, M, and an unmapped mesh, N, with weight W
    If N is not in the mesh heap
        create a processor heap with one entry for P having weight W
        add N to the mesh heap with weight W
    else
        if there is not entry for P in N's processor heap
            create an entry for P with weight W and add it to N's processor heap
            add W to N's weight in the mesh heap and bubble N's entry up
        else
            add W to P's weight in N's processor heap and bubble P's entry up
            add W to N's weight in the mesh heap and bubble N's entry up
        endif
    endif
```

Figure 5.1 Pseudocode for mapping mesh M onto processor P

steps needed to map or assign a mesh to a processor. Figure 5.2 gives an outline of the steps in the topology-based small mesh distribution algorithm. One point that needs to be mentioned is that, from studying the types of coupling communication that occurs in the test problems, I learned that there are often many couplings with the same communication cost. This led me to explore further ordering options in the *mesh heap*. In the end, a combination of maximum coupling cost to a processor and the computation cost associated with the mesh was used to select the next mesh to map. The coupling cost is still the primary selection criteria with the higher computation cost only used to make the selection when there are multiple entries with the same communication cost.

The runtime of this algorithm is:

$$O(M\ log(M)\ C\ log(C)\ P\ log(P)),$$

where M is the number of small meshes, C is the maximum number of couplings involving a single mesh, and P is the number of processors. The $P\ log(P)$ term comes from sometimes finding the nearest empty processor.

```
max distance = one half of the diameter of selected processor configuration
search distance = 1
While there are unconsidered meshes
    proc = null
    delete the maximum entry, M, from the mesh heap
    best proc = processor in the maximum entry of M's processor heap
    While ((proc is null) && (processor heap is not empty))
        delete the maximum entry, P, from M's processor heap
        If P has room for M
            proc = P
        else if there is a neighbor of P, P', within search distance with room for M
            proc = P'
        endwhile
    if (proc is null) reset M's processor heap
    While ((proc is null) && (processor heap is not empty))
        delete the maximum entry, P, from M's processor heap
        if there is any neighbor of P, P', within max distance with room for M
            proc = P'
        endwhile
    if ((proc is null) && (there is an empty processor))
        proc = empty processor nearest best proc
    mark M considered
    if (proc is null)
        save M for filler
    else
        map M onto P (see Figure 5.1)
    endwhile
create a minimum (by computation allocated) heap of processors
create a maximum (by computation to perform) heap of filler meshes
while there are filler meshes in the max heap
    delete maximum mesh from max heap
    delete minimum proc from min heap
    map M onto P
    insert P with new computation cost into the minimum processor heap
    endwhile
```

Figure 5.2 Topology-based Distribution Algorithm for Small Meshes

5.2.4 Limitations -vs- Advantages

This approach to ICRM parallelization is limited to those problems in which it makes sense to distribute every mesh either over all processors or place it on just one. This limits the use of the approach for some problems. For example, this approach does not work when there are too many elements in some meshes to allow them to fit on a single processor, but too few to distribute over all processors. In Chapter 6, another approach for automatic distribution of such problems is explored.

The most important advantage to this approach is that the parallelization burden to the developer of such codes as TRAC is greatly reduced while the applicable communication optimization technology is increased (regular distribution optimizations can be applied). In particular, the developer does not have to explicitly parallelize for a specific machine. Further, the code is not parallelized for a specific input set (reactor configuration). This is particularly important for codes such as TRAC where many users do not work on code development and many code developers do not work on reactor model development. Finally, for the scientist trying to analyze the properties of a specific reactor, the parallelization is specific to their reactor design and, hence, should run significantly faster than a parallelization that is only code and not configuration specific.

5.3 Algorithm Validation

The two primary goals of this research are: 1) reduce the programmer burden for parallelization of composite grid problems, and 2) take advantage of the partial regularity of composite grid problems to provide acceptable performance in the resulting parallelization. The first goal is achieved via the automatic distribution algorithm which removes the burden of grid mapping from the user. The second goal is achieved through the transformation process, which will be described and illustrated in Section 6.3.

Since the applications focused on in this work are not written in a form that can be worked with directly, measures of load balance and communication are presented in place of timing results. The measure of load balance that is presented is the total number of floating point additions, multiplications, and divisions. This load balance measure is compared for the processor with the most work ($Comp_{max}$) and the processor with the least work ($Comp_{min}$). This number is also presented for the highest computation small mesh in each problem. None of these algorithms can produce a mapping with less computation on every processor than that of the highest computation small mesh. Consideration of this fact provides a practical upper bound on the number of processors for a problem when using any of these algorithms. The following measures of communication will be presented:

- *distance* is the maximum distance between any pair of communicating processors;
- *neighbors* is the maximum number of processors communicated with for any processor;
- $Comm_{max_1}$ is the maximum amount of communication, in bytes, for any process3or;
- $Comm_{max_2}$ is the maximum communication, in bytes, between any pair of processors; and
- $Comm_{total}$ is the total communication, in bytes, for the simulation.

With these measures of load balance and communication, the results of the three distribution algorithms are compared for four different problems. In all of these measures, only the computation and communication associated with the small meshes in the problem are included. In the tables of results, the selected processor configuration is shown below the number of processors.

The speedup bound, $Dist_{sb}$, is the speedup, with the data distribution supplied, that would be achieved if there were no communication, i.e.,

$$Dist_{sb} = \frac{Comp_{max}\ for\ n\ processors}{Comp_{max}\ for\ one\ processor}.$$

The speedup bound is rounded to two decimal places.

For each test problem, the two random algorithms were run three times and complete results are shown for all runs to provide a minimal feel for the range of results.

5.3.1 LOFT 3-d Reactor Model

The validation results for the 3-d LOFT reactor model are shown in Figure 5.3. There are 1,567,560 floating point computations for each cycle associated with the largest mesh in the 3-d LOFT reactor model used as input for this experiment. This implies that 4.85 is the maximum speedup bound, $Dist_{sb}$, achievable using these algorithms.

Num. Procs	Validation Measure	Random			Balanced Random			Topology-based
		Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	
2	$distance$	1	1	1	1	1	1	1
	$neighbors$	1	1	1	1	1	1	1
	$Comm_{max_1}$	642,723	474,765	811,388	733,294	530,110	412,076	258,817
	$Comm_{max_2}$	642,723	474,765	811,388	733,294	530,110	412,076	258,817
	$Comm_{total}$	642,723	474,765	811,388	733,294	530,110	412,076	258,817
	$Comp_{min}$	3,076,456	2,552,654	2,976,440	3,797,726	3,526,528	3,789,070	3,795,162
	$Comp_{max}$	4,520,278	5,044,080	4,620,294	3,799,008	4,070,206	3,807,664	3,801,572
	$Dist_{sb}$	1.68	1.51	1.64	2.0	1.87	2.0	2.0
4	$distance$	2	2	2	2	2	2	2
(4)	$neighbors$	3	3	3	3	3	3	2
	$Comm_{max_1}$	424,061	502,350	617,605	577,780	524,199	449,010	227,991
	$Comm_{max_2}$	221,165	283,693	259,068	315,624	333,877	305,599	165,800
	$Comm_{total}$	692,559	991,885	1,017,226	889,352	889,289	760,743	315,573
	$Comp_{min}$	833,788	312,550	275,044	1,670,140	362,558	1,137,682	1,880,110
	$Comp_{max}$	3,288,990	4,268,956	2,801,412	2,090,080	3,135,120	2,612,600	1,917,616
	$Dist_{sb}$	2.31	1.78	2.71	3.63	2.42	2.91	3.96
8	$distance$	3	3	3	3	3	3	2
(2x2x2)	$neighbors$	7	5	6	4	5	4	3
	$Comm_{max_1}$	358,652	543,085	311,919	205,838	437,136	383,674	205,838
	$Comm_{max_2}$	155,850	209,023	143,656	143,272	131,078	131,078	143,272
	$Comm_{total}$	1,067,086	976,498	833,685	1,156,150	1,042,058	1,107,449	579,418
	$Comp_{min}$	175,028	87,514	250,040	311,268	436,288	511,300	311,268
	$Comp_{max}$	2,826,416	4,007,696	2,216,382	1,567,560	1,567,560	1,567,560	1,567,560
	$Dist_{sb}$	2.69	1.90	3.43	4.85	4.85	4.85	4.85
16	$distance$	4	4	4	4	4	4	3
(2x2x4)	$neighbors$	7	10	7	5	6	5	3
	$Comm_{max_1}$	233,743	308,623	233,743	373,982	308,946	233,739	168,288
	$Comm_{max_2}$	90,407	130,822	90,407	102,762	155,909	90,534	90,407
	$Comm_{total}$	1,014,221	1,079,588	1,014,221	1,042,060	1,054,731	1,054,560	551,581
	$Comp_{min}$	62,510	12,502	62,510	0	37,506	0	0
	$Comp_{max}$	1,828,820	2,713,898	1,828,820	1,567,560	1,567,560	1,567,560	1,567,560
	$Dist_{sb}$	4.15	2.80	4.15	4.85	4.85	4.85	4.85

Figure 5.3 Validation results for 3-d LOFT reactor model.

In the 3-d LOFT results, the maximum computation for a processor can not be reduced by using more than 8 processors for the topology-based distribution algorithm. The random distribution never minimizes the maximum computation mapped to a processor, even when 32 processors are used. The balanced random sometimes minimizes the maximum computation mapped to a processor, but it generates nearly twice as much total communication as the topology-based distribution algorithm does. These results illustrate the earlier point that the cost of the total small mesh computation divided by the cost of the highest computation small mesh bounds the number of processors that can be used effectively. It is interesting to note that using more processors than this bound can reduce the communication as in the case of going from 8 to 16 processors for the 3-d LOFT data set.

In the rest of the validation results, only the cases up to the number of processors indicated by the bound imposed by the highest computation small mesh are presented.

5.3.2 LOFT 1-d Reactor Model

The validation results for the 1-d LOFT reactor model are shown in Figure 5.4. There are 287,546 floating point computations for each cycle associated with the largest mesh in the 1-d LOFT reactor model used as input for this experiment. This implies that 5.91 is the maximum speedup bound, $Dist_{sb}$, that can be achieved using these algorithms.

In the 1-d LOFT results, the topology-based distribution algorithm produces good load balance until the maximum number of processors is used. Both random algorithms induce nearly twice as much total communication as the topology-based algorithm does and do not minimize the maximum computation mapped to a processor.

5.3.3 H.B. Robinson Reactor Model

The validation results for the H.B. Robinson reactor model are shown in Figure 5.5. There are 979,725 floating point computations for each cycle associated with the largest mesh in the H.B. Robinson reactor model used as input for this experiment. This implies that 13.93 is the maximum speedup bound, $Dist_{sb}$, that can be achieved using these algorithms.

In the H.B. Robinson results, the topology-based distribution algorithm produces good load balance until the maximum number of processors is used. Both random algorithms induce from three to ten times as much total communication as the topology-

Num. Procs	Validation Measure	Random			Balanced Random			Topology-based
		Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	
2	$distance$	1	1	1	1	1	1	1
	$neighbors$	1	1	1	1	1	1	1
	$Comm_{max_1}$	237,711	499,033	455,996	729,804	468,518	561,100	174,604
	$Comm_{max_2}$	237,711	499,033	455,996	729,804	468,518	561,100	174,604
	$Comm_{total}$	237,711	499,033	455,996	729,804	468,518	561,100	174,604
	$Comp_{min}$	75,012	950,152	1,000,160	1,137,682	1,136,404	1,148,906	1,186,412
	$Comp_{max}$	1,623,982	1,423,950	1,373,942	1,236,420	1,237,698	1,225,196	1,187,690
	$Dist_{sb}$	1.05	1.19	1.24	1.37	1.37	1.39	1.43
4 (2x2)	$distance$	2	2	2	2	2	2	2
	$neighbors$	2	2	2	2	2	2	2
	$Comm_{max_1}$	368,464	393,118	343,223	511,022	468,503	486,557	100,119
	$Comm_{max_2}$	205,745	267,990	218,095	273,484	230,620	205,751	74,875
	$Comm_{total}$	518,629	605,768	556,095	673,854	749,326	699,075	199,643
	$Comp_{min}$	200,032	200,032	250,040	337,554	462,574	437,570	587,594
	$Comp_{max}$	887,642	937,650	836,356	898,866	875,140	873,862	600,096
	$Dist_{sb}$	1.91	1.81	2.03	1.89	1.94	1.94	2.83
8 (2x4)	$distance$	3	3	3	3	3	3	3
	$neighbors$	6	7	6	6	7	6	4
	$Comm_{max_1}$	386,487	398,720	268,442	492,068	391,968	391,968	112,236
	$Comm_{max_2}$	155,684	130,656	155,693	143,170	143,170	143,170	37,543
	$Comm_{total}$	761,809	773,979	656,226	929,672	879,960	892,474	299,786
	$Comp_{min}$	112,518	37,506	75,012	112,518	162,526	100,016	275,044
	$Comp_{max}$	623,822	562,590	486,300	573,814	450,072	562,590	312,550
	$Speedup\ Bound$	2.72	3.02	3.49	2.96	3.77	3.02	5.44
16 (4x4)	$distance$	4	4	4	4	4	4	3
	$neighbors$	9	5	7	7	6	6	4
	$Comm_{max_1}$	336,343	391,968	323,881	354,426	354,426	354,426	100,063
	$Comm_{max_2}$	130,656	130,656	130,656	118,142	118,142	118,142	37,543
	$Comm_{total}$	824,035	917,170	811,519	917,160	917,164	892,143	475,137
	$Comp_{min}$	25,004	12,502	25,004	37,506	37,506	50,008	0
	$Comp_{max}$	475,076	487,578	436,292	425,068	350,056	362,558	287,546
	$Dist_{sb}$	3.58	3.48	3.89	4.0	4.85	4.69	5.91

Figure 5.4 Validation results for 1-d LOFT reactor model.

Num. Procs	Validation Measure	Random			Balanced Random			Topology-based
		Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	
2	$distance$	1	1	1	1	1	1	1
	$neighbors$	1	1	1	1	1	1	1
	$Comm_{max_1}$	1,925,037	1,594,031	2,070,846	1,695,802	1,752,610	1,769,099	175,212
	$Comm_{max_2}$	1,925,037	1,594,031	2,070,846	1,695,802	1,752,610	1,769,099	175,212
	$Comm_{total}$	1,925,037	1,594,031	2,070,846	1,695,802	1,752,610	1,769,099	175,212
	$Comp_{min}$	6,194,264	5,472,994	5,023,722	6,816,077	6,764,787	6,815,277	6,820,646
	$Comp_{max}$	7,448,792	8,170,062	8,619,334	6,826,979	6,878,269	6,827,779	6,822,410
	$Dist_{sb}$	1.83	1.67	1.58	2.0	1.98	2.0	2.0
4 (2x2)	$distance$	2	2	2	2	2	2	2
	$neighbors$	3	3	3	3	3	3	3
	$Comm_{max_1}$	1,385,452	1,684,366	1,715,044	1,980,719	1,926,480	1,682,133	340,767
	$Comm_{max_2}$	836,066	856,983	751,173	885,810	888,699	584,231	253,157
	$Comm_{total}$	2,539,938	2,645,116	2,399,371	2,748,397	2,734,669	2,505,357	453,393
	$Comp_{min}$	3,026,207	2,012,022	2,234,253	3,381,632	3,174,467	3,374,499	3,393,334
	$Comp_{max}$	4,346,050	6,607,312	5,459,451	3,440,296	3,591,843	3,441,578	3,423,225
	$Dist_{sb}$	3.14	2.06	2.50	3.97	3.80	3.96	3.99
8 (2x4)	$distance$	3	3	3	3	3	3	3
	$neighbors$	6	6	7	7	7	7	3
	$Comm_{max_1}$	1,315,890	1,571,388	1,182,595	1,074,040	1,300,735	1,079,338	256,026
	$Comm_{max_2}$	439,537	348,954	339,441	268,420	261,532	243,456	168,424
	$Comm_{total}$	3,069,682	3,082,156	3,057,056	2,876,610	3,094,870	2,926,450	757,014
	$Comp_{min}$	671,021	773,842	887,642	1,446,145	1,471,149	1,607,389	1,699,231
	$Comp_{max}$	2,765,188	2,973,635	3,615,806	1,863,039	1,835,471	1,804,857	1,723,994
	$Dist_{sb}$	4.93	4.59	3.77	7.32	7.43	7.56	7.91
16 (2x2x4)	$distance$	4	4	4	4	4	4	4
	$neighbors$	12	11	11	9	12	13	5
	$Comm_{max_1}$	1,122,450	1,290,502	107,817	966,800	979,010	1,062,721	268,424
	$Comm_{max_2}$	168,332	208,707	208,603	143,304	168,204	143,352	130,662
	$Comm_{total}$	3,076,498	3,141,973	3,363,202	3,275,604	3,300,632	3,250,504	1,013,236
	$Comp_{min}$	175,028	125,020	162,526	487,578	687,610	737,618	594,727
	$Comp_{max}$	1,887,002	2,306,701	3,419,861	979,725	979,725	979,725	979,725
	$Dist_{sb}$	7.23	5.91	4.0	13.93	13.93	13.93	13.93

Figure 5.5 Validation results for H.B. Robinson reactor model.

based algorithm does and never a smaller maximum amount of computation mapped to a processor.

5.3.4 Westinghouse AP600 Reactor Model

The validation results for the Westinghouse AP600 reactor model are shown in Figures 5.6 and 5.7. There are 3,396,380 floating point computations for each cycle associated with the largest mesh in the AP600 reactor model used as input for this experiment. This implies that 20.82 is the maximum speedup bound, $Dist_{sb}$, that can be achieved using these algorithms.

In the AP600 results for 8 processors, the first case where the balanced random distribution algorithm produces a better load balance than the topology-based algorithm occurs. This occurs because the topology-based distribution algorithm saves meshes for mapping at the end when it can not find a good placement quickly. This could be improved by keeping a minimum heap of processors and placing the meshes on the least loaded processor when good placement is not found quickly. In this particular case, the total communication is more than a factor of two better for the topology-based distribution algorithm and there is one less neighbor for the topology-based distribution algorithm. From this it appears that the distribution generated by the topology-based distribution algorithm would probably outperform the distribution generated by the balanced random algorithm. This point will be explored in the future to improve the topology-based algorithm. Both random algorithms induce from one and a half to five times as much total communication as the topology-based distribution algorithm does.

5.4 Chapter Summary

An algorithm that automatically finds a distribution of data in ICRM problems given a well-structured HPF program and topological connection specifications has been presented. When models are being developed via iteration of modification of the model and experimentation with the model, automatic distribution of small mesh ICRM problems is critical. This algorithm provides an important first step.

Num. Procs	Validation Measure	Random Run 1	Random Run 2	Random Run 3	Balanced Random Run 1	Balanced Random Run 2	Balanced Random Run 3	Topology-based
2	*distance*	1	1	1	1	1	1	1
	neighbors	1	1	1	1	1	1	1
	$Comm_{max_1}$	6,474,062	5,880,201	5,880,201	4,399,114	4,399,114	4,764,464	898,236
	$Comm_{max_2}$	6,474,062	5,880,201	5,880,201	4,399,114	4,399,114	4,764,464	898,236
	$Comm_{total}$	6,474,062	5,880,201	5,880,201	4,399,114	4,399,114	4,764,464	898,236
	$Comp_{min}$	28,654,532	34,658,144	34,658,144	35,350,248	35,350,248	35,332,616	35,353,952
	$Comp_{max}$	42,053,568	36,049,960	36,049,960	35,357,856	35,357,856	35,375,492	35,354,184
	$Dist_{sb}$	1.68	1.96	1.96	2.0	2.0	2.0	2.0
4	*distance*	2	2	2	2	2	2	2
(4)	*neighbors*	3	3	3	3	3	3	3
	$Comm_{max_1}$	5,004,068	5,373,743	5,100,365	4,249,056	5,193,137	6,072,235	2,732,938
	$Comm_{max_2}$	2,124,474	2,187,416	1,853,293	2,156,131	2,230,068	2,467,942	1,994,743
	$Comm_{total}$	9,074,508	8,961,254	8,818,410	7,623,410	9,296,276	8,393,454	3,045,056
	$Comp_{min}$	12,689,058	9,814,157	11,843,250	16,081,910	16,588,082	15,029,737	17,673,916
	$Comp_{max}$	21,971,402	28,353,922	22,836,846	19,725,130	18,745,406	20,312,964	17,682,254
	$Dist_{sb}$	3.22	2.49	3.10	3.58	3.77	3.48	4.0
8	*distance*	3	3	3	3	3	3	3
(4x2)	*neighbors*	7	7	7	7	7	7	6
	$Comm_{max_1}$	3,161,931	3,520,754	3,697,177	42,13,511	4,229,617	4,309,920	3,444,543
	$Comm_{max_2}$	831,463	909,152	781,599	1,077,368	838,499	1,099,967	1,188,476
	$Comm_{total}$	9,882,217	9,972,450	9,857,245	10,046,362	10,130,799	10,259,210	4,369,449
	$Comp_{min}$	553,318	5,213,498	2,229,684	6,847,655	8,546,404	8,051,934	8,169,262
	$Comp_{max}$	14,017,239	13,423,794	14,254,777	10,776,975	8,964,021	10,123,825	9,592,649
	$Dist_{sb}$	5.04	5.27	4.96	6.56	7.89	6.98	7.37

Figure 5.6 Validation results for Westinghouse AP600 reactor model.

Num. Procs	Validation Measure	Random Run 1	Random Run 2	Random Run 3	Balanced Random Run 1	Balanced Random Run 2	Balanced Random Run 3	Topology-based
16	$distance$	4	4	4	4	4	4	4
(2x4x2)	$neighbors$	14	14	14	12	12	11	5
	$Comm_{max_1}$	2,737,633	1,932,648	2,463,119	2,405,384	2,122,664	2,531,268	1,984,966
	$Comm_{max_2}$	417,022	588,027	442,058	417,022	417,022	470,299	653,310
	$Comm_{total}$	10,779,059	10,281,767	10,872,695	10,518,055	10,535,847	10,440,030	5,268,712
	$Comp_{min}$	1,650,746	575,092	854,464	323,850	2,467,704	3,134,638	3,765,830
	$Comp_{max}$	7,775,049	9,327,302	8,573,895	5,682,405	6,923,390	5,682,405	5,498,962
	$Dist_{sb}$	9.09	7.58	8.25	12.44	10.21	12.44	12.86
32	$distance$	6	6	6	6	6	6	6
(8x4)	$neighbors$	19	22	13	17	21	19	20
	$Comm_{max_1}$	1,912,084	1,582,016	2,039,157	1,544,462	2,896,651	1,610,225	4,162,976
	$Comm_{max_2}$	221,029	261,324	208,703	205,770	233,547	233,547	482,353
	$Comm_{total}$	11,087,906	11,321,581	11,296,545	11,165,883	11,203,437	11,072,775	7,060,487
	$Comp_{min}$	112,518	150,024	112,518	639,125	362,558	939,173	1,250,682
	$Comp_{max}$	5,986,540	5,270,880	9,745,719	4,049,530	3,527,010	4,310,790	3,406,077
	$Dist_{sb}$	11.81	13.14	7.26	17.46	20.05	16.40	20.76
64	$distance$	7	7	7	7	7	7	6
(2x8x4)	$neighbors$	16	22	26	21	23	17	19
	$Comm_{max_1}$	1,223,433	1,447,059	1,678,217	1,444,670	1,482,224	1,466,645	2,529,909
	$Comm_{max_2}$	168,216	143,500	195,993	130,806	130,822	143,276	286,360
	$Comm_{total}$	11,346,617	11,228,393	11,489,797	11,427,159	11,215,955	11,321,517	6,375,264
	$Comp_{min}$	12,502	0	0	50,008	25,004	62,510	25,004
	$Comp_{max}$	6,566,201	6,324,335	4,969,309	3,396,380	3,396,380	3,396,380	3,396,380
	$Dist_{sb}$	10.77	11.18	14.23	20.82	20.82	20.82	20.82

Figure 5.7 Validation results for Westinghouse AP600 reactor model.

Chapter 6

Mapping Algorithm and Precompiler Support for Medium and Mixed Size Mesh Composite Grid Problems

So far, algorithms have been discussed to automatically distribute problems with meshes that are large or small relative to the number of processors. Now, an algorithm is presented for automatic distribution of problems with meshes that are too large to fit on a single processor and too small to distribute over all processors. This becomes more common as more processors are used.

In the first section of this chapter, the intuition behind the approach is presented. The next section presents the algorithm used to distribute ICRM problems that contain medium and mixed size meshes. The following section describes and illustrates the transformation that an HPF precompiler would perform. Next, algorithm validation is achieved by increasing the number of processors used on all of the problems that have been used to validate the previous algorithms. Finally, the results of this chapter are summarized.

6.1 Intuition

When meshes are too large to fit on a single processor and too small to distribute over all of the processors, they should be mapped, not necessarily uniformly, to a subset of the processors. For example, it may be best to map 3/4 of the elements to one processor, thereby completely filling that processor, and the remaining 1/4 to another processor that will need further computation to achieve load balance.

One approach would be to break the medium size meshes into pieces that have at most the amount of computation that is to be allocated to an individual processor and then distribute the pieces via a small mesh distribution algorithm. The problem with this approach is that there is no longer a guarantee that the communication within a single mesh will be nearest neighbor. For this reason, a different approach is used. Here, packing medium size meshes into large size bins is explored. The large size bins are then distributed using the large mesh distribution algorithm previously described. This approach ensures that all communication within a single mesh is nearest neighbor in the mapping.

Basically, I want to find the packing of medium size meshes into large bins that minimizes the total volume of the bins, with the restriction that each bin must be big enough to distribute over all processors. To do this, the number of bins, the size of the bins, and the packing of the meshes into the bins are optimized.

This problem is an extension to the nonlinear optimization problem 3-dimensional bin packing. The extension to 3-dimensional bin packing for this problem is that the sizes of the bins are variable and unknown.

6.2 Automatic Distribution Algorithm

Now consider how to pack medium size meshes into bins and how the packing of medium size meshes and their subsequent distribution affects the distribution of other meshes. The overall distribution algorithm is described for problems with medium size meshes in three stages. First, the basic distribution procedure when there are only medium size meshes in the problem is described. Next, the basic distribution procedure is modified when large meshes are present in addition to the medium size meshes. Finally, the full automatic distribution procedure for the case where small size meshes are added to the mix of meshes in the problem is presented. Section 6.2.1 will present the details of the mesh packing algorithm. This is followed by a discussion of the limitations of the approach.

Medium Meshes Only: First, consider the case when there are only medium size meshes in the problem. All of the meshes are packed into some number of large bins for distribution. This is done as follows.

Pack Medium: Medium size mesh packing consists of the following steps.

Find Bin Bound: An upper bound on the number of bins is found by adding up the number of elements in the meshes and dividing by the number of processors (assuming the same computation is taking place on each mesh).

Perform Packing: Now for each number of bins, up to the maximum, the optimal packing of the meshes into the bins is found, with each bin being at least big enough to distribute over all processors. To do this, a model was developed for this nonlinear optimization problem. To solve the packing problem, a local linear approximation to the nonlinear problem is solved iteratively. The best solution is the number of bins that produce the minimum total volume over all of the bins.

Build Large: From the best solution, a large mesh is created to represent each composite bin. The size of the large mesh is the size of the composite bin. The computation per element is the maximum of the computation costs for the medium meshes that were packed into the composite.

Translate Couplings: Each new large mesh has all of the couplings translated from the medium size meshes it contains into their corresponding positions in the large composite mesh. This translation involves dimension alignment and offset alignment. The direction of alignment is always increasing, but that could be modified to reduce coupling communication cost between two meshes packed in the same composite.

Distribute Large: The large composite meshes are distributed and have coupling communication reduction performed exactly as with the original large distribution algorithm. During this distribution, each large mesh is assumed to have the same computation being performed for each element, i.e., the new composite meshes are distributed according to the assumption that they are regular.

Add Large Meshes: Next, let's consider the case where there are both large and medium size meshes.

Sort: Before composite bins can be built, the meshes must be sorted to determine which meshes are large and which are medium. The current separating heuristic is that each processor should have at least four computational elements for each distributed dimension. This is somewhat arbitrary and will be studied in future work.

Pack Medium: Perform packing as described above.

Distribute Large: In this case, large mesh distribution is performed for the new large composite meshes and the original large size meshes at the same time using the large mesh distribution algorithm described in Chapter 4.

Add Small Meshes: Finally, consider the case where there are meshes of all three size classifications in the problem.

Sort: The first step is to sort the meshes by size and separate them into the three size classifications. Large meshes are separated out first. Then, to classify small meshes, the amount of computation from non-large meshes that should be assigned to each processor is determined. A small mesh is one that has less than one half of that amount. This could be further refined after having obtained an initial separation of medium and small sized meshes. This separation criteria will be studied further in future research.

Pack Medium: Perform packing as described above.

Distribute Large: Distribute all (original and composite) large meshes as described above.

Distribute Small: Small meshes are mapped by repeatedly putting the unmapped small mesh, with the most computation, onto the least computationally loaded processor.

Now the overall distribution algorithm has been described for problems with medium size meshes, some of the details of the packing algorithm will be discussed.

6.2.1 Algorithm Details

All of the meshes are packed into some number of large bins for distribution. This allows distribution of all meshes with nearest neighbor communication for all communications that involve only one mesh. Only meshes having three or fewer dimensions are packed. All of the test problems involve meshes with at most three dimensions, so this does not restrict the algorithm for the test problems. Further, all of the physical phenomena simulations that are being targeted here use at most 3-dimensional meshes. This work can be extended to higher dimension with the effect that the optimization problem becomes even more difficult and expensive.

First, an upper bound on the number of bins is found by summing up the number of elements in the meshes and dividing by the number of processors. This assumes that the same computation is taking place on each mesh. If this is not the case, the meshes should be separated into groups according to computational load and then packed. Now, for each number of bins, up to the maximum, an optimal packing of the meshes into the bins is sought, with each bin being at least big enough to distribute over all processors. To do this, a model is developed for this nonlinear optimization problem.

The model says to minimize the sum of the volumes of the bins subject to the following constraints:

- all bin volumes must be at least as large as a precomputed lower bound;
- all meshes must be completely contained in one and only one bin;
- no two meshes assigned to the same bin may overlap; and
- each dimension of a mesh assigned to a bin must be assigned to exactly one dimension of the bin.

To achieve the last constraint, all of the meshes are converted to 3-dimensional meshes by padding out with dimensions having a single element.

Since this nonlinear optimization problem is not computationally tractable, the packing problem is solved iteratively via a local linear approximation using the CPLEX callable library [CPLEX O94]. The best solution is the set of bins and associated packing that produces the minimum total volume over all of the bins.

From the best solution, a large mesh is created to represent each composite bin. This large mesh has all of the couplings translated from the medium size meshes into their positions in the composite mesh. This process is outlined in 6.1. Finally, the large meshes are distributed and have coupling communication reduction performed exactly as with the original large meshes. During this distribution, each large mesh is distributed according to the assumption that the all meshes, including the large composite meshes, are regular.

The one modification that was made to this basic optimization procedure is that meshes are prepacked when possible. A prepacking makes a single n-dimensional larger mesh out of two n-dimensional smaller meshes when the sizes in $n-1$ of the dimensions are the same. In this case there are no "holes" introduced in the new larger mesh. This was done because the optimization-based packing algorithm is expensive. This modification comes into play in the reactor simulation problems in particular. In the simulation of the AP600, with only 64 processors, this prepacking reduces the number of meshes that are to be packed using CPLEX from 32 to 11.

```
compute the maximum number of usable bins
For each number of bins, b, up to the number of usable bins
   find a feasible solution as a starting point
    while not converged
       prepare the linear programming problem for CPLEX
       call CPLEX routines to solve the local linear problem
       check to see if the solution has converged
       if not converged
          update the variables for next iteration
       else if the solution is better than any previous solution
          save it
create a large mesh for each bin in the best solution
fill in each large mesh's information from the meshes packed in it
translate the couplings from medium mesh to large mesh with offsets
```

Figure 6.1 Medium Mesh Packing Algorithm

Linear Optimization Model

Now, the linear optimization problem is described in detail. It is presented here in a form that allows for easy generation of input files for CPLEX. In the following, $1 \leq i \leq b$ and $1 \leq j \leq N$, where N is the number of meshes and b is the number of composite 'bins'. All values are integers and every variable in font $\texttt{FONT}$ is a constant that will appear as an integer in the input file for CPLEX. Further, all δ and in variables can have only value 0 or 1.

The following relationships hold:

- $in_{ij} = 1 \iff$ mesh j is in composite bin i;
- $\texttt{MAX}$ is an upper bound on x, y, and z for all meshes;
- X_i, Y_i, and Z_i are the sizes of composite bin i;
- $\texttt{XOC}_i, \texttt{YOC}_i$, and $\texttt{ZOC}_i$ are the initial sizes of composite bin i;
- $\texttt{DX}_j, \texttt{DY}_j$, and $\texttt{DZ}_j$ are the sizes for mesh j;
- $x0_j, y0_j$, and $z0_j$ are the starting locations for mesh j in the final packing;

- xm_j, ym_j, and zm_j are the ending locations for mesh j in the final packing;
- $\delta a_{kj} = 1 \Longleftrightarrow$ mesh k starts after mesh j ends in dimension a $(a \in \{x, y, z\})$;
- $\delta ab_j = 1 \Longleftrightarrow$ dimension a of mesh j is mapped to dimension b of the composite bin it is packed into $(a, b \in \{x, y, z\})$.

$\mathtt{XOC}_i, \mathtt{YOC}_i$, and $\mathtt{ZOC}_i$ are used, with $X0_i, Y0_i$, and $Z0_i$, to limit the amount of change allowed in X_i, Y_i, and Z_i for the iterative local linear approximation.

What should be minimized is

$$\sum_{i=1}^{b} X_i \cdot Y_i \cdot Z_i,$$

but CPLEX can only deal with linear problems. Therefore, an iterative local linear optimization is performed to solve, approximately, the nonlinear problem of interest. Hence, the following model results. This description is not in CPLEX input form, but does reflect the type of problem that can be solved using CPLEX.

Minimize

$$\sum_{i=1}^{b} X_i \cdot \mathtt{YOC}_i \cdot \mathtt{ZOC}_i + \mathtt{XOC}_i \cdot Y_i \cdot \mathtt{ZOC}_i + \mathtt{XOC}_i \cdot \mathtt{YOC}_i \cdot Z_i$$

Subject to

$$X_i \cdot \mathtt{YOC}_i \cdot \mathtt{ZOC}_i + \mathtt{XOC}_i \cdot Y_i \cdot \mathtt{ZOC}_i + \mathtt{XOC}_i \cdot \mathtt{YOC}_i \cdot Z_i \geq \mathtt{LB} \tag{6.1}$$

$$\begin{aligned}
X0_i &= \mathtt{XOC}_i \\
Y0_i &= \mathtt{YOC}_i \\
Z0_i &= \mathtt{ZOC}_i \\
X0_i - X_i &<= \mathtt{TRUST\%}\ of\ \mathtt{XOC}_i \\
Y0_i - Y_i &<= \mathtt{TRUST\%}\ of\ \mathtt{YOC}_i \qquad (6.2) \\
Z0_i - Z_i &<= \mathtt{TRUST\%}\ of\ \mathtt{ZOC}_i \\
X_i - X0_i &<= \mathtt{TRUST\%}\ of\ \mathtt{XOC}_i \\
Y_i - Y0_i &<= \mathtt{TRUST\%}\ of\ \mathtt{YOC}_i \\
Z_i - Z0_i &<= \mathtt{TRUST\%}\ of\ \mathtt{ZOC}_i
\end{aligned}$$

$$\begin{aligned} x0_j &\geq 1 \\ y0_j &\geq 1 \\ z0_j &\geq 1 \\ xm_j &\geq 1 \\ ym_j &\geq 1 \\ zm_j &\geq 1 \end{aligned} \tag{6.3}$$

$$\begin{aligned} xm_j + \mathtt{MAX} \cdot in_{ij} &\leq \mathtt{MAX} + X_i \\ ym_j + \mathtt{MAX} \cdot in_{ij} &\leq \mathtt{MAX} + Y_i \\ zm_j + \mathtt{MAX} \cdot in_{ij} &\leq \mathtt{MAX} + Z_i \end{aligned} \tag{6.4}$$

$$\begin{aligned} xm\gamma_{kj} - \mathtt{MAX} \cdot \delta x_{kj} &\leq 0 \\ -xm_j + xm\gamma_{kj} &\leq 0 \\ xm_j - xm\gamma_{kj} + \mathtt{MAX} \cdot \delta x_{kj} &\leq \mathtt{MAX} \\ x0_k - \delta x_{kj} &\geq xm\gamma_{kj} \\ x0_k - \mathtt{MAX} \cdot \delta x_{kj} &\leq xm_j \end{aligned} \tag{6.5}$$

$$\begin{aligned} ym\gamma_{kj} - \mathtt{MAX} \cdot \delta y_{kj} &\leq 0 \\ -ym_j + ym\gamma_{kj} &\leq 0 \\ ym_j - ym\gamma_{kj} + \mathtt{MAX} \cdot \delta y_{kj} &\leq \mathtt{MAX} \\ y0_k - \delta y_{kj} &\geq ym\gamma_{kj} \\ y0_k - \mathtt{MAX} \cdot \delta y_{kj} &\leq ym_j \end{aligned} \tag{6.6}$$

$$\begin{aligned} zm\gamma_{kj} - \mathtt{MAX} \cdot \delta z_{kj} &\leq 0 \\ -zm_j + zm\gamma_{kj} &\leq 0 \\ zm_j - zm\gamma_{kj} + \mathtt{MAX} \cdot \delta z_{kj} &\leq \mathtt{MAX} \\ z0_k - \delta z_{kj} &\geq zm\gamma_{kj} \\ z0_k - \mathtt{MAX} \cdot \delta z_{kj} &\leq zm_j \end{aligned} \tag{6.7}$$

$$\begin{aligned} -in_{ij} + in_{ijk} &\leq 0 \\ -in_{ik} + in_{ijk} &\leq 0 \\ in_{ij} + in_{ik} - in_{ijk} &\leq 1 \\ -\delta x_{kj} - \delta x_{jk} - \delta y_{kj} - \delta y_{jk} - \delta z_{kj} - \delta z_{jk} + in_{ijk} &\leq 0 \end{aligned} \tag{6.8}$$

$$\begin{aligned} xm_j &= x0_j + \delta xx_j \cdot \mathtt{DX}_j + \delta xy_j \cdot \mathtt{DY}_j + \delta xz_j \cdot \mathtt{DZ}_j - 1 \\ ym_j &= y0_j + \delta yx_j \cdot \mathtt{DX}_j + \delta yy_j \cdot \mathtt{DY}_j + \delta yz_j \cdot \mathtt{DZ}_j - 1 \\ zm_j &= z0_j + \delta zx_j \cdot \mathtt{DX}_j + \delta zy_j \cdot \mathtt{DY}_j + \delta zz_j \cdot \mathtt{DZ}_j - 1 \end{aligned} \tag{6.9}$$

$$\begin{aligned} \delta xx_j + \delta xy_j + \delta xz_j &= 1 \\ \delta yx_j + \delta yy_j + \delta yz_j &= 1 \\ \delta zx_j + \delta zy_j + \delta zz_j &= 1 \end{aligned} \tag{6.10}$$

$$\begin{aligned} \delta xx_j + \delta yx_j + \delta zx_j &= 1 \\ \delta xy_j + \delta yy_j + \delta zy_j &= 1 \\ \delta xz_j + \delta yz_j + \delta zz_j &= 1 \end{aligned} \tag{6.11}$$

$$\sum_{i=1}^{b} in_{ij} = 1 \tag{6.12}$$

Each of the following comments pertains to a group of constraints and each group of constraints has a single label.

- Constraint 6.1 states that the volume for each composite bin must be at least as great as an input constant lower bound.

- The constraints in 6.2, limit the change that can occur in the various dimensions of the bins. This is necessary due to the use of the iterative local linear optimization for the globally nonlinear problem. `TRUST` is the current trust region size factor [JS83].

- The constraints in 6.3 and 6.4 ensure that each mesh packed in any composite bin is completely inside of that composite bin.

- No two meshes in the same composite bin are allowed to overlap. This requirement is represented in four groups of constraints: one for each dimension of the problem (the constraints in 6.5–6.7) and one that specifies that there must be no overlap in at least one dimension between each pair of meshes packed in the same composite bin (the constraints in 6.8).
- The constraints in 6.9 define the upper bounds in the three dimensions for mesh j.
- The constraints in 6.10 and 6.11 enforce that each mesh dimension is mapped to exactly one dimension of the composite bin and that no two dimensions of the mesh are mapped to the same dimension of the composite bin.
- Finally, constraint 6.12 ensures sure that each mesh gets assigned to exactly one composite bin.

In generating input for CPLEX from this model, all of the equations for all of the meshes and bins must be listed explicitly.

Small Mesh Distribution

To improve load balance via small mesh distribution, the algorithm begins by determining the amount of computation that has been assigned to each processor.

Next, the algorithm sorts processors in increasing computation order using a min heap and sorts small meshes in decreasing computation order using a max heap. While there are unmapped small meshes, the algorithm repeatedly maps the mesh with the most computation onto the processor with the least computation.

In future work, I will use the ideas developed for the general small mesh distribution algorithm in mapping small meshes according to coupling requirements while trying to improve load balance.

6.2.2 Limitations

Because of the high cost of solving the linear optimization problems, this mapping technique is quite expensive. This is illustrated by the number of variables in the optimization problem. If m is the number of meshes and b is the number of bins, then there are approximately

$$m^2 * b + 7m^2 + mb + 15m + 6b$$

variables in a given instance of the iterative procedure, e.g., for 10 meshes and 1 bin there are 966 variables. For this reason, future work will explore the use of heuristics to obtain more quickly an acceptable solution.

6.3 Support for High Performance Fortran Programs

Since the HPF language definition does not require the interprocedural analysis that is needed to perform the transformations that are about to be discussed, precompiler support must be added for HPF. Interprocedural analysis is needed in order to allow the user to write programs in the form described in Section 3.3 rather than in the style that will be illustrated in the post-transformation excerpts. With the use of a precompiler, which is completely machine independent, *any* full HPF compiler that is available for the machine that the user is interested in running on can be used.

6.3.1 Original High Performance Fortran Program

Figures 6.3–6.9 illustrate an abstracted application outline of the form described in Section 3.3. This represents the code to be developed by the application programmer. Figure 6.2 shows the original HPF module and main program. Figure 6.3 shows the original input routine for 1-, 2-, and 3-dimensional meshes. Figure 6.4 shows all of the original allocation routines for the abstracted application. Figures 6.5 and 6.6 show the original HPF main subroutine. Figure 6.7 shows the original outlines for the compute routines. Figure 6.8 shows the original HPF input routines for the couplings. Figure 6.9 shows one of the original coupling update routines.

Figure 6.9 and its translation will serve to illustrate how coupling update routines are modified for use with the automatic distribution scheme. I do not illustrate how user codes perform couplings between different dimensions of different meshes. This is because it would add to the length of the examples, but it would not change the HPF transformation process in any way.

6.3.2 Transformation and Final High Performance Fortran Program

The example from the previous section is now transformed and the resulting code is shown. An outline of the steps needed for transformation is provided in Figure 6.10. To illustrate these steps, parts of an abstracted application, as it would look after transformation, are provided.

```
      module mesh_module
      type mesh_3d
        integer size(3), mesh_id
        real, pointer, dimension(:,:,:) :: p, q, u, v, zm, x, y, z
      end type mesh_3d
      type mesh_2d
        integer size(2), mesh_id
        real, pointer, dimension(:,:) :: p, q, u, v, zm, x, y, z
      end type mesh_2d
      type mesh_1d
        integer size(1), mesh_id
        real, pointer, dimension(:) :: p, q, u, v, zm, x, y, z
      end type mesh_1d
      type coupling
        integer id_A, id_B, lo_A(3), hi_A(3), lo_B(3), hi_B(3)
      end type coupling
      type (mesh_3d), allocatable, dimension(:)::Meshes_3d
      type (mesh_2d), allocatable, dimension(:)::Meshes_2d
      type (mesh_1d), allocatable, dimension(:)::Meshes_1d
      integer Num_3d_meshes, Num_2d_meshes, Num_1d_meshes, Num_couplings
      type (coupling), allocatable, dimension(:)::Couplings
      integer, allocatable, dimension(:)::dims
      contains

C     ... subroutine main_routine goes here
C     ... runtime constant input routines go here

      end module mesh_module

      program big_mesh
      use mesh_module
C     The computation/communication reflect the CFD codes
C     at Mississippi State for aerodynamic simulations.
      read(*,*) Num_steps, Num_Meshes
      allocate(dims(Num_Meshes))
      call read_meshes_3d()
      call read_meshes_2d()
      call read_meshes_1d()
      call read_couplings()
      call main_routine()
      end program big_mesh
```

Figure 6.2 Original HPF module and main program.

```
C     Note: mesh ids run from 1 to number of meshes total.

      subroutine read_meshes_3d()
      read(*,*) Num_3d_meshes
      allocate(Meshes_3d(Num_3d_meshes))
      do i = 1, Num_3d_meshes
         read(*,*) Meshes_3d(i)%mesh_id, Meshes_3d(i)%size
         dims(Meshes_3d(i)%id) = 3
      end do
      end subroutine read_meshes_3d

      subroutine read_meshes_2d()
      read(*,*) Num_2d_meshes
      allocate(Meshes_2d(Num_2d_meshes))
      do i = 1, Num_2d_meshes
         read(*,*) Meshes_2d(i)%mesh_id, Meshes_2d(i)%size
         dims(Meshes_2d(i)%id) = 2
      end do
      end subroutine read_meshes_2d

      subroutine read_meshes_1d()
      read(*,*) Num_1d_meshes
      allocate(Meshes_1d(Num_1d_meshes))
      do i = 1, Num_1d_meshes
         read(*,*) Meshes_1d(i)%mesh_id, Meshes_1d(i)%size
         dims(Meshes_1d(i)%id) = 1
      end do
      end subroutine read_meshes_1d
```

Figure 6.3 Original HPF example input routines for all meshes.

```
      subroutine allocate_3d(Mesh_info)
      type (mesh_3d) Mesh_info
      real, pointer :: all_array(:,:,:)
      allocate(all_array(Mesh_info%size(1),Mesh_info%size(2),Mesh_info%size(3)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_3d

      subroutine allocate_2d(Mesh_info)
      type (mesh_2d) Mesh_info
      real, pointer :: all_array(:,:)
      allocate(all_array(Mesh_info%size(1),Mesh_info%size(2)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_2d

      subroutine allocate_1d(Mesh_info)
      type (mesh_1d) Mesh_info
      real, pointer :: all_array(:,:)
      allocate(all_array(Mesh_info%size(1)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_1d

      subroutine allocate_all_meshes()
      do i = 1, Num_3d_meshes
         call allocate_3d(Meshes_3d(i))
      end do
       do i = 1, Num_2d_meshes
         call allocate_2d(Meshes_2d(i))
      end do
       do i = 1, Num_1d_meshes
         call allocate_1d(Meshes_1d(i))
      end do
      end subroutine allocate_all_meshes
```

Figure 6.4 Original HPF example allocation routines for all meshes.

```
      subroutine main_routine
      call allocate_all_meshes
      forall i=1, Num_3d_meshes
         call initial_3d(Meshes_3d(i),dthlf,dt)
      end forall
C     ... similar loops for 1- and 2-dimensional initialization
      do i_step = 1, Num_steps
         forall i=1, Num_3d_meshes
            call update_mesh_3d(Meshes_3d(i),dthlf,dt)
         end forall
C        ... similar loops for 1- and 2-dimensional mesh updates
C        coupling update loop from next figure goes here
      end do
C     ... print results, etc.
C     all subroutines except input go here
      end subroutine main_routine
```

Figure 6.5 Original HPF main subroutine.

Before the actual transformations are presented, recall that the distribution algorithms have three size classifications for meshes: small, medium, and large. In the HPF transformation process, only two size classifications need to be consider: small and large. This simplification is possible because the type of information the algorithm provides for medium and large size meshes is the same. With a specific size classification for meshes, which is dependent only on the dimensionality of the mesh, cloning may be performed. Since only three or fewer parallel dimension are supported, each 3-dimensional mesh can be distributed in any of the six dimension alignment permutations or it can be mapped to a specific processor. Therefore, each original routine that takes only one mesh as input is replaced with seven clones. The coupling update routines, which take multiple meshes as input, get many more clones. For example, if one of the meshes is 3-dimensional and the other is 2-dimensional, then there are forty-nine different clones of the `coupling_update_3d2d` routine.

The first modification to the user's HPF program is the addition of the distribution specifications to the user defined data types that contain arrays to be distributed (compare Figures 6.2 and 6.11). For large and medium meshes, a distribution order specification is needed along with the decomposition size and the alignment offsets.

```
forall i=1, Num_couplings
   id_A = Couplings(i)%id_A
   id_B = Couplings(i)%id_B
   select case (dims(id_A))
   case (1)
      id_A = id_A - Num_3d_meshes - Num_2d_meshes
      id_B = id_B - Num_3d_meshes - Num_2d_meshes
      call update_couplings_1d1d(Meshes_1d(id_A),
                                 Couplings(i),Meshes_1d(id_B))
   case (2)
      id_A = id_A - Num_3d_meshes
      select case (dims(id_B))
         case (1)
         id_B = id_B - Num_3d_meshes - Num_2d_meshes
         call update_couplings_2d1d(Meshes_2d(id_A),
                                    Couplings(i),Meshes_1d(id_B))
      case (2)
         id_B = id_B - Num_3d_meshes
         call update_couplings_2d2d(Meshes_2d(id_A),
                                    Couplings(i),Meshes_2d(id_B))
      end select
   case (3)
      select case (dims(id_B))
         case (1)
         id_B = id_B - Num_3d_meshes - Num_2d_meshes
         call update_couplings_3d1d(Meshes_3d(id_A),
                                    Couplings(i),Meshes_1d(id_B))
      case (2)
         id_B = id_B - Num_3d_meshes
         call update_couplings_3d2d(Meshes_3d(id_A),
                                    Couplings(i),Meshes_2d(id_B))
      case (3)
         call update_couplings_3d3d(Meshes_3d(id_A),
                                    Couplings(i),Meshes_3d(id_B))
      end select
   end select
end forall
```

Figure 6.6 Original coupling update insert for the main routine.

```
      subroutine update_mesh_3d(Mesh_info,dthlf,dt)
      use physics
      type (mesh_3d) Mesh_info
      real dthlf, dt
C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_3d

      subroutine update_mesh_2d(Mesh_info,dthlf,dt)
      use physics
      type (mesh_2d) Mesh_info
      real dthlf, dt
C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_2d

      subroutine update_mesh_1d(Mesh_info,dthlf,dt)
      use physics
      type (mesh_1d) Mesh_info
      real dthlf, dt
C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_1d
```

Figure 6.7 Original HPF example compute routines for all meshes.

```
 subroutine read_couplings()
 read(*,*) Num_couplings
 allocate(Couplings(Num_couplings))
 do i = 1, Num_couplings
    read(*,*) Couplings(i)%id_A,Couplings(i)%id_B
    select case (dims(id_A)
    case (1)
       read(*,*) Couplings(i)%lo_A[1],Couplings(i)%hi_A[1]
    case (2)
       read(*,*) Couplings(i)%lo_A[1],Couplings(i)%lo_A[2],
&                Couplings(i)%hi_A[1],Couplings(i)%hi_A[2]
    case (3)
       read(*,*) Couplings(i)%lo_A[1],Couplings(i)%lo_A[2],
&                Couplings(i)%lo_A[3],Couplings(i)%hi_A[1],
&                Couplings(i)%hi_A[2],Couplings(i)%hi_A[3]
    end select
    select case (dims(id_B)
    case (1)
       read(*,*) Couplings(i)%lo_B[1],Couplings(i)%hi_B[1]
    case (2)
       read(*,*) Couplings(i)%lo_B[1],Couplings(i)%lo_B[2],
&                Couplings(i)%hi_B[1],Couplings(i)%hi_B[2]
    case (3)
       read(*,*) Couplings(i)%lo_B[1],Couplings(i)%lo_B[2],
&                Couplings(i)%lo_B[3],Couplings(i)%hi_B[1],
&                Couplings(i)%hi_B[2],Couplings(i)%hi_B[3]
    end select
 end do
 end subroutine read_couplings
```

Figure 6.8 Original HPF input routine for couplings.

```
      subroutine update_couplings_3d2d(Mesh_A,Couple,Mesh_B)
      type (mesh_3d) Mesh_A
      type (mesh_2d) Mesh_B
      type (coupling) Couple
      real p_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real q_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real u_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real v_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real zm_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real x_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real y_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real z_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      p_tmp(1:Couple%hi_B(1)-Couple%lo_B(1)+1,1:Couple%hi_B(2)-Couple%lo_B(2)+1)
     & = Mesh_A%p(Couple%lo_A(1):Couple%hi_A(1),Couple%lo_A(2):Couple%hi_A(2),
     &             Couple%lo_A(3):Couple%hi_A(3))
C     ... save all the temporaries
      Mesh_A%p(Couple%lo_A(1):Couple%hi_A(1),Couple%lo_A(2):Couple%hi_A(2),
     &         Couple%lo_A(3):Couple%hi_A(3))
     & = Mesh_B%p(Couple%lo_B(1):Couple%hi_B(1),Couple%lo_B(2):Couple%hi_B(2))
     &   * alpha
     &  + Mesh_A%p(Couple%lo_A(1):Couple%hi_A(1),Couple%lo_A(2):Couple%hi_A(2),
     &             Couple%lo_A(3):Couple%hi_A(3))*(1-alpha)
C     ... update all of the ``A'' variables
      Mesh_B%p(Couple%lo_B(1):Couple%hi_B(1),Couple%lo_B(2):Couple%hi_B(2))
     & = p_tmp(1:Couple%hi_B(1)-Couple%lo_B(1)+1,1:Couple%hi_B(2)-Couple%lo_B(2)+1)
     &  + Mesh_B%p(Couple%lo_B(1):Couple%hi_B(1),Couple%lo_B(2):Couple%hi_B(2))
     &  * (1-alpha)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_3d2d
```

Figure 6.9 Original HPF example update routine for 3d to 2d couplings.

```
Add distribution specifications to user defined types (UDTs).
Insert code to read distribution specifications.
Modify storage allocation to implement data distribution.
Add a ``processors'' statement to the main subroutine.
For each subroutine with distributed data:
   For each set of distribution types for the various parameters:
      Clone the subroutine.
      Replace all UDT parameters with their elements.
      Insert distribution specifications for all distributed data.
For each call to a subroutine with distributed parameters:
   Replace all UDT parameters in the call with their elements.
   Add a control structure to select the clone of the callee with the
      proper distribution specifications.
   Insert parameters with distribution information.
```

Figure 6.10 Transformation steps for composite grid HPF programs.

For small meshes, only the processor identification is needed. In order to distinguish between these two types of the distribution, a mesh type is added.

The input routine (`read_mesh_3d`) is modified to read the existing data distribution information from a file (compare Figures 6.3 and 6.12). Part of this input is dependent on the type of distribution used for the mesh. This case structured dependence will be seen throughout the transformed program. The distribution information is read from a secondary file that has been created by the automatic distribution system. Alternatively, a user could specify their own distributions in a properly formatted file. Also notice that all runtime constant topology specifications are read in before any storage is allocated for any distributed data. Other input routines are modified in a similar manner.

The array allocation routines (e.g., `allocate_2d`) are cloned for each possible distribution type including each alignment dimension order. After cloning, `template`, `align`, and `distribute` statements are added (compare Figures 6.4 and 6.13). Note that if the programmer had declared all of the arrays in the data structure to be allocatable, but not pointers, then the transformation would require that they be

```
      module mesh_module
      type mesh_3d
        integer size(3), mesh_id
        integer mesh_type, dist_order, decomp_size(3), align_offset(3), proc(3)
        real, pointer, dimension(:,:,:) :: p, q, u, v, zm, x, y, z
      end type mesh_3d
      type mesh_2d
        integer size(2), mesh_id
        integer mesh_type, dist_order, decomp_size(3), align_offset(3), proc(3)
        real, pointer, dimension(:,:) :: p, q, u, v, zm, x, y, z
      end type mesh_2d
      type mesh_1d
        integer size(1), mesh_id
        integer mesh_type, dist_order, decomp_size(3), align_offset(3), proc(3)
        real, pointer, dimension(:) :: p, q, u, v, zm, x, y, z
      end type mesh_1d
      type coupling
        integer id_A, id_B, lo_A(3), hi_A(3), lo_B(3), hi_B(3)
      end type coupling
      integer Num_Proc_i,Num_Proc_j,Num_Proc_k
      integer Num_3d_meshes,Num_2d_meshes,Num_1d_meshes,Num_couplings
      type (mesh_3d), allocatable, dimension(:) :: Meshes_3d
      type (mesh_2d), allocatable, dimension(:) :: Meshes_2d
      type (mesh_1d), allocatable, dimension(:) :: Meshes_1d
      type (coupling), allocatable, dimension(:) :: Couplings
      integer, allocatable, dimension(:) :: dims
      contains
C        subroutine main_routine goes here
C        runtime constant input routines go here
      end module mesh_module

      program big_mesh
      use mesh_module
      read(*,*) Num_steps, Num_Meshes
      allocate(dims(Num_Meshes))
      open(unit=8,file='dist.large')
      read(8,*) Num_Proc_i, Num_Proc_j, Num_Proc_k
      call read_meshes_3d()
      call read_meshes_2d()
      call read_meshes_1d()
      call read_couplings()
      call main_routine()
      end program big_mesh
```

Figure 6.11 Transformed HPF module and main program.

```
subroutine read_meshes_3d()
read(*,*) Num_3d_meshes
allocate(Meshes_3d(Num_3d_meshes))
do i = 1, Num_3d_meshes
   read(*,*) Meshes_3d(i)%mesh_id, Meshes_3d(i)%size
   dims(Meshes_3d(i)%id) = 3
   read(8,*) Meshes_3d(i)%mesh_type
   select case (Meshes_3d(i)%mesh_type)
   case (0)
      read(8,*) Meshes_3d(i)%proc
   case (1)
      read(8,*) Meshes_3d(i)%dist_order
      read(8,*) Meshes_3d(i)%decomp_size, Meshes_3d(i)%align_offset
   end select
end do
end subroutine read_meshes_2d
```

Figure 6.12 Transformed HPF example input routine.

changed to pointers and that the pointers be generated as in the figure. This form is necessary, as HPF does not allow distribution of elements of user defined data structures. The routine to allocate all of the meshes in all of the user defined data structures (`allocate_all_meshes`) has a case structure added to select the correct cloned allocation routine for mesh according to the distribution order read from the input file (compare Figures 6.13 and 6.14).

In the calling routine (`main_routine`) the processor layout is added (compare Figures 6.5 and 6.6 with Figures 6.15–6.18). The parameters to the update routines are changed to match the changes in the called routines. Further, a case structure is added which, based on the distribution type, selects the correct clone of the subroutine to call. In Figures 6.17 and 6.18, the real variables between `p` and `z` in the parameter lists are eliminated to save space.

Each routine to perform mesh updates (`update_mesh_3d`) is cloned for each possible distribution type including all possible dimension alignment orders. Distribution specifications are added to each clone, i.e., `template`, `align`, and `distribute` statements. Where user defined structures with arrays were passed, the arrays must be passed explicitly and all of their uses must be modified to explicit array use rather than structure use. This is a result of the fact that HPF does not allow distribution of

```
      subroutine allocate_2d_ijk(Mesh_info)
      type (mesh_2d) Mesh_info
!HPF$ template decomp(Mesh_info%dist_size(1),Mesh_info%dist_size(2),
!HPF$&                Mesh_info%dist_size(3))
!HPF$ align all_array(i,j) with decomp(i+Mesh_info%align_offset(1),
!HPF$      j+Mesh_info%align_offset(2),Mesh_info%align_offset(3))
!HPF$ distribute(cyclic(Mesh_info%dist_size(1)/Num_Proc_i),
!HPF$&     cyclic(Mesh_info%dist_size(2)/Num_Proc_j),
!HPF$&     cyclic(Mesh_info%dist_size(3)/Num_Proc_k)) onto procs::decomp
      real, pointer :: all_array(:,:)
      allocate(all_array(Mesh_info%size(1),Mesh_info%size(2)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_2d_ijk
C     ... similar subroutines for other permutations

      subroutine allocate_2d_small(Mesh_info)
      type (mesh_2d) Mesh_info
!HPF$ template decomp(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ align all_array(i,j) with decomp(Mesh_info%proc(1),
!HPF$                                 Mesh_info%proc(2),Mesh_info%proc(3))
!HPF$ distribute(cyclic(1),cyclic(1),cyclic(1)) onto procs::decomp
      real, pointer :: all_array(:,:)
      allocate(all_array(Mesh_info%size(1),Mesh_info%size(2)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_2d_small
C     ... similar sets of routines for 1- and 3-dimensional mesh allocation
```

Figure 6.13 Transformed HPF example allocation routines for 2-d meshes.

```
      subroutine allocate_all_meshes()
      do i = 1, Num_2d_meshes
         select case (Meshes_2d(i)&mesh_type)
         case (0)
            call allocate_2d_small(Meshes_2d(i))
         case (1)
            select case (Meshes_2d(i)%dist_order)
            case (123)
               call allocate_2d_ijk(Meshes_2d(i))
            case (213)
               call allocate_2d_jik(Meshes_2d(i))
C           ... similar for other cases
            end select
         end select
      end do
C     ... similar loops for 1- and 3-dimensional mesh allocation
      end subroutine allocate_all_meshes
```

Figure 6.14 Transformed HPF example master allocation routine.

elements of user defined types. Further, parameters are added for distribution specifications. These distribution specification values are used in the HPF statements and must therefore be constants at the subprogram entry. For an example of the transformations to update routines, compare Figures 6.7 and 6.19.

Since the routine to read couplings is not modified in the translation process, it is not shown again.

The transformation of coupling update routines is the same as that for mesh update routines except that more clones are generated. This cloning explosion is due to the number of possible combinations of mesh distributions that can be passed as parameters. For an example of the transformations to coupling update routines, compare Figure 6.9 with Figures and 6.20– 6.22.

This transformation procedure allows each clone to be optimized based on the distribution information required by HPF compilers while allowing the user to ignore such details as cloning and parameter expansion.

This section has shown how to transform the user program into a machine independent form that is not dependent on the specific problem topology. The resulting code reads in alignment and distribution specifications along with the user input. Hence, when the input changes, the user only needs to run the distribution algorithm

```
      subroutine main_routine
!HPF$ Processors procs(Num_Proc_i,Num_Proc_j,Num_Proc_k)
      call allocate_all_meshes
      do i=1, Num_3d_meshes
         select case (Meshes_3d(i)%)mesh_type)
         case (0)
            call initial_3d_small(Meshes_3d(i)%proc(1),Meshes_3d(i)%proc(2),
     &Meshes_3d(i)%proc(3),Meshes_3d(i)%p,Meshes_3d(i)%q,Meshes_3d(i)%u,
     &Meshes_3d(i)%v,Meshes_3d(i)%zm,Meshes_3d(i)%x,Meshes_3d(i)%y,
     &Meshes_3d(i)%z,dthlf,dt)
         case (1)
            select case (Meshes_3d(i)%dist_order)
            case (123)
               call initial_3d_ijk(Meshes_3d(i)%decomp_size(1),
     &Meshes_3d(i)%decomp_size(2),Meshes_3d(i)%decomp_size(3),
     &Meshes_3d(i)%align_offset(1),Meshes_3d(i)%align_offset(2),
     &Meshes_3d(i)%align_offset(3),Meshes_3d(i)%p,Meshes_3d(i)%q,
     &Meshes_3d(i)%u,Meshes_3d(i)%v,Meshes_3d(i)%zm,Meshes_3d(i)%x,
     &Meshes_3d(i)%y,Meshes_3d(i)%z,dthlf,dt)
C              ... similar case for each distribution order
            end select
         end select
      end do
C     ... similar loop for 2d and 1d mesh initialization
```

Figure 6.15 Transformed HPF main_routine example.

```
C     subroutine main_routine continued
      do i_step = 1, Num_steps
         do i=1, Num_3d_meshes
            select case (Meshes_3d(i)%)mesh_type)
            case (0)
               call update_mesh_3d_ijk(Meshes_3d(i)%proc(1),
     &Meshes_3d(i)%proc(2),Meshes_3d(i)%proc(3),Meshes_3d(i)%p,
     &Meshes_3d(i)%q,Meshes_3d(i)%u,Meshes_3d(i)%v,Meshes_3d(i)%zm,
     &Meshes_3d(i)%x,Meshes_3d(i)%y,Meshes_3d(i)%z,dthlf,dt)
            case (1)
               select case (Meshes_3d(i)%distribution_order)
               case (123)
                  call update_mesh_3d_ijk(Meshes_3d(i)%decomp_size(1),
     &Meshes_3d(i)%decomp_size(2),Meshes_3d(i)%decomp_size(3),
     &Meshes_3d(i)%align_offset(1),Meshes_3d(i)%align_offset(2),
     &Meshes_3d(i)%align_offset(3),Meshes_3d(i)%p,Meshes_3d(i)%q,
     &Meshes_3d(i)%u,Meshes_3d(i)%v,Meshes_3d(i)%zm,Meshes_3d(i)%x,
     &Meshes_3d(i)%y,Meshes_3d(i)%z,dthlf,dt)
C                 ... similar case for each distribution order
               end select
            end select
        end do
C       ... similar loop for 2d and 1d mesh updates
```

Figure 6.16 Transformed HPF main_routine example continued.

```
        do i=1, Num_couplings
           id_A = Couplings_3d(i)%id_A; id_B = Couplings_3d(i)%id_B
           select case (dims(id_A))
           case (1)
              id_A=id_A-Num_3d_meshes-Num_2d_meshes
              id_B=id_B-Num_3d_meshes-Num_2d_meshes
              if((Meshes_1d(id_A)%mesh_type==1).and.
                 (Meshes_1d(id_B)%mesh_type==1))then
                 if ((Meshes_1d(id_A)%dist_order.eq.123) .and.
     &               (Meshes_1d(id_B)%dist_order.eq.123)) then
                    call update_couplings_1d_1d_ijk_ijk(
     &Meshes_1d(id_A)%decomp_size(1),Meshes_1d(id_A)%decomp_size(2),
     &Meshes_1d(id_A)%decomp_size(1),Meshes_1d(id_A)%align_offset(1),
     &Meshes_1d(id_A)%align_offset(2),Meshes_1d(id_A)%align_offset(3),
     &Meshes_1d(id_A)%p,...,Meshes_1d(id_A)%z,Couplings(i)%lo_A(1),
     &Couplings(i)%hi_A(1),Couplings(i)%lo_B(1),Couplings(i)%hi_B(1),
     &Meshes_1d(id_B)%decomp_size(1),Meshes_1d(id_B)%decomp_size(2),
     &Meshes_1d(id_B)%decomp_size(3),Meshes_1d(id_B)%align_offset(1),
     &Meshes_1d(id_B)%align_offset(2),Meshes_1d(id_B)%align_offset(3),
     &Meshes_1d(id_B)%p,..,Meshes_1d(id_B)%z,dthlf,dt)
C                   ... similar cases for other distribution order pairs
                 endif
```

Figure 6.17 Transformed HPF main_routine example continued.

```
              elseif((Meshes_1d(id_A)%mesh_type==1).and.
                     (Meshes_1d(id_B)%mesh_type==0))then
                 if (Meshes_1d(id_A)%dist_order.eq.123) then
                    call update_couplings_1d_1d_ijk_small(
     &Meshes_1d(id_A)%decomp_size(1),Meshes_1d(id_A)%decomp_size(2),
     &Meshes_1d(id_A)%decomp_size(3),Meshes_1d(id_A)%align_offset(1),
     &Meshes_1d(id_A)%align_offset(2),Meshes_1d(id_A)%align_offset(3),
     &Meshes_1d(id_A)%p,...,Meshes_1d(id_A)%z,Couplings(i)%lo_A(1),
     &Couplings(i)%hi_A(1),Couplings(i)%lo_B(1),Couplings(i)%hi_B(1),
     &Meshes_1d(id_B)%proc(1),Meshes_1d(id_B)%proc(2),Meshes_1d(id_B)%proc(3),
     &Meshes_1d(id_B)%p,Meshes_1d(id_B)%z,dthlf,dt)
C                   ... similar cases for other distribution orders for A
                 endif
              elseif((Meshes_1d(id_A)%mesh_type==0).and.
                     (Meshes_1d(id_B)%mesh_type==1))then
C                ... similar to above except that A and B are swapped
              elseif((Meshes_1d(id_A)%mesh_type==0).and.
                     (Meshes_1d(id_B)%mesh_type==0))then
                    call update_couplings_1d_1d_small_small(
     &Meshes_1d(id_A)%proc(1),Meshes_1d(id_A)%proc(2),Meshes_1d(id_A)%proc(3),
     &Meshes_1d(id_A)%p,...,Meshes_1d(id_A)%z,Couplings(i)%lo_A(1),
     &Couplings(i)%hi_A(1),Couplings(i)%lo_B(1),Couplings(i)%hi_B(1),
     &Meshes_1d(id_B)%proc(1),Meshes_1d(id_B)%proc(2),Meshes_1d(id_B)%proc(3),
     &Meshes_1d(id_B)%p,...,Meshes_1d(id_B)%z,dthlf,dt)
              endif
C          ... similar cases for 2- and 3-dimensional As except that there is
C              also a case for B's dimension
           end select
        end do
      end do
C     ... print results, etc.
      contains
C     all subroutines except input routines
      end subroutine main_routine
```

Figure 6.18 Transformed HPF main_routine example continued.

```
      subroutine update_mesh_3d_ijk(i_size,j_size,k_size,i_off,j_off,k_off,
     &   p,q,u,v,zm,x,y,z,dthlf,dt)
      use physics
!HPF$ template decomp(i_size,j_size,k_size)
!HPF$ align (i,j,k) with *decomp(i+i_off,j+j_off,k+k_off)::p,q,u,v,zm,x,y,z
!HPF$ distribute (cyclic(i_size/Num_Proc_i),cyclic(j_size/Num_Proc_j),
!HPF$&            cyclic(k_size/Num_Proc_k)) onto procs :: decomp
      real p(:,:,:),u(:,:,:),v(:,:,:),q(:,:,:),zm(:,:,:),x(:,:,:),y(:,:,:),z(:,:,:)
      real dthlf, dt

C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_3d_ijk
```

Figure 6.19 Transformed HPF example compute routine for 3-d meshes.

to generate the new input alignment and distribution specifications. The code does not have to be recompiled. Further, the explicit distribution specifications generated in the transformation process allow the use of regular communication optimizations such as communication blocking and parallelization.

This transformation procedure requires the same type of interprocedural analysis that is used for Fortran D compilers for cloning and communication optimization [HHKT92]. The only other requirements for these transformations is that the program be written as described in Section 3.3.

6.3.3 HPF Compiler Support

Given the form of the program that is output by the precompiler, as described in the previous section on HPF program transformation, the HPF compilation step is now considered. Since HPF compilers are not required to do interprocedural analysis and cloning for communication, the transformations were necessary in order to provide complete alignment and distribution information to the HPF compiler in every subroutine with distributed data structures. Further, the transformations are sufficient to generate a standard HPF program from a program written using the template and style recommendations of Section 3.3. Since the end product of the transformation procedure just described is a standard HPF program, there are no further compiler enhancements necessary for correct code generation (for either a MIMD or SIMD machine). However, there are three areas where special attention in the compiler may improve performance. Two of these concerns are related to the boundary data ex-

```
      subroutine update_couplings_3d2d_ijk_ijk(i_A_size,j_A_size,k_A_size,i_A_off,
     & j_A_off,k_A_off,A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z,i_A_lo,j_A_lo,k_A_lo,
     & i_A_hi,j_A_hi,k_A_hi,i_B_lo,j_B_lo,i_B_hi,j_B_hi,i_B_size,j_B_size,
     & k_B_size,i_B_off,j_B_off,k_B_off,B_p,B_q,B_u,B_v,B_zm,B_x,B_y,B_z,dthlf,dt)
!HPF$ template decompA(i_A_size,j_A_size,k_A_size)
!HPF$ template decompB(i_B_size,j_B_size,k_B_size)
!HPF$ align (i,j,k) with *decompA(i+i_A_off,j+j_A_off,k+k_A_off) ::
!HPF$&     A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z
!HPF$ align (i,j) with *decompB(i+i_B_off,j+j_B_off,k_B_off) ::
!HPF$&     B_p,B_q,B_u,B_v,B_zm,B_x,B_y,B_z
!HPF$ distribute (cyclic(i_A_size/Num_Proc_i),cyclic(j_A_size/Num_Proc_j),
!HPF$&  cyclic(k_A_size/Num_Proc_k))onto procs::decompA
!HPF$ distribute (cyclic(i_B_size/Num_Proc_i),cyclic(j_B_size/Num_Proc_j),
!HPF$& cyclic(k_B_size/Num_Proc_k)) onto procs::decompB
      real A_p(:,:,:),A_u(:,:,:),A_v(:,:,:),A_q(:,:,:),A_zm(:,:,:),A_x(:,:,:)
      real A_y(:,:,:),A_z(:,:,:),B_p(:,:),B_u(:,:),B_v(:,:),B_q(:,:)
      real B_zm(:,:)B_x(:,:),B_y(:,:),B_z(:,:),dthlf,dt
      real p_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real q_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real u_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real v_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real zm_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real x_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real y_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real z_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      p_tmp(1:Couple%hi_B(1)-Couple%lo_B(1)+1,1:Couple%hi_B(2)-Couple%lo_B(2)+1)
     &    =  A_p(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi)
C     ... save all the temporaries
      A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) =
     &A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) * frac
     &+ B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) * (1.0-frac)
C     ... update all of the ``A'' variables
      B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) =
     &B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) * frac + q_tmp * (1.0-frac)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_3d2d_ijk_ijk
```

Figure 6.20 Transformed HPF example update routine for 3d ijk to 2d ijk couplings.

```
subroutine update_couplings_3d2d_ijk_small(i_A_size,j_A_size,k_A_size,i_A_off,
     &  j_A_off,k_A_off,A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z,i_A_lo,j_A_lo,k_A_lo,
     &  i_A_hi,j_A_hi,k_A_hi,i_B_lo,j_B_lo,i_B_hi,j_B_hi,i_B_proc,j_B_proc,
     &  k_B_proc,B_p,B_q,B_u,B_v,B_zm,B_x,B_y,B_z,dthlf,dt)
!HPF$ template decompA(i_A_size,j_A_size,k_A_size)
!HPF$ template decompB(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ align (i,j,k) with *decompA(i+i_A_off,j+j_A_off,k+k_A_off)::A_p,A_q,A_u,
!HPF$&     A_v,A_zm,A_x,A_y,A_z
!HPF$ align (i,j) with *decompB(i_B_proc,j_B_proc,k_B_proc)::B_p,B_q,B_u,B_v,
!HPF$&     B_zm,B_x,B_y,B_z
!HPF$ distribute (cyclic(i_A_size/Num_Proc_i),cyclic(j_A_size/Num_Proc_j),
!HPF$&            cyclic(k_A_size/Num_Proc_k)) onto procs::decompA
!HPF$ distribute (cyclic(1),cyclic(1),cyclic(1)) onto procs::decompB
      real A_p(:,:,:),A_u(:,:,:),A_v(:,:,:),A_q(:,:,:),A_zm(:,:,:),A_x(:,:,:)
      real A_y(:,:,:),A_z(:,:,:),B_p(:,:),B_u(:,:),B_v(:,:),B_q(:,:)
      real B_zm(:,:)B_x(:,:),B_y(:,:),B_z(:,:),dthlf,dt
      real p_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real q_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real u_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real v_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real zm_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real x_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real y_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real z_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      p_tmp(1:Couple%hi_B(1)-Couple%lo_B(1)+1,1:Couple%hi_B(2)-Couple%lo_B(2)+1)
     &    =  A_p(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi)
C     ... save all the temporaries
      A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) =
     &A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) * frac
     &+ B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) * (1.0-frac)
C     ... update all of the ''A'' variables
      B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) =
     &B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) * frac + q_tmp * (1.0-frac)
C     ... update all of the ''B'' variables
      end subroutine update_couplings_3d2d_ijk_small
```

Figure 6.21 Transformed HPF example update routine for 3d ijk to 2d small couplings.

```
      subroutine update_couplings_3d2d_small_small(i_A_proc,j_A_proc,k_A_proc,
     &  A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z,i_A_lo,j_A_lo,k_A_lo,i_A_hi,
     &  j_A_hi,k_A_hi,i_B_lo,j_B_lo,i_B_hi,j_B_hi,i_B_proc,j_B_proc,k_B_proc,
     &  B_p,B_q,B_u,B_v,B_zm,B_x,B_y,B_z,dthlf,dt)
!HPF$ template decomp(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ align (i,j,k) with *decomp(i_A_proc,j_A_proc,k_A_proc)::A_p,A_q,A_u,A_v,
!HPF$&     A_zm,A_x,A_y,A_z
!HPF$ align (i,j) with *decomp(i_B_proc,j_B_proc,k_B_proc)::B_p,B_q,B_u,B_v,
!HPF$&     B_zm,B_x,B_y,B_z
!HPF$ distribute (cyclic(1),cyclic(1),cyclic(1)) onto procs::decomp
      real A_p(:,:,:),A_u(:,:,:),A_v(:,:,:),A_q(:,:,:),A_zm(:,:,:),A_x(:,:,:)
      real A_y(:,:,:),A_z(:,:,:),B_p(:,:),B_u(:,:),B_v(:,:),B_q(:,:)
      real B_zm(:,:)B_x(:,:),B_y(:,:),B_z(:,:),dthlf,dt
      real p_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real q_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real u_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real v_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real zm_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real x_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real y_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      real z_tmp(Couple%hi_B(1)-Couple%lo_B(1)+1,Couple%hi_B(2)-Couple%lo_B(2)+1)
      p_tmp(1:Couple%hi_B(1)-Couple%lo_B(1)+1,1:Couple%hi_B(2)-Couple%lo_B(2)+1)
     &    =  A_p(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi)
C     ... save all the temporaries
      A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) =
     &A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) * frac
     &+ B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) * (1.0-frac)
C     ... update all of the ``A'' variables
      B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) =
     &B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi) * frac + q_tmp * (1.0-frac)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_3d2d_small_small
```

Figure 6.22 Transformed HPF example update routine for 3d small to 2d small couplings.

change associated with the coupling of the regular meshes. The third area is separate compilation.

Boundary iterations are those iterations of loops that perform updates to the boundary elements of a mesh. If these iterations are peeled from a loop, then the communication associated with coupling can be moved outside of the loop. This implies that there should not be as much time spent waiting to get values from other processors. This same approach is seen in hand parallelized material dynamics calculations. Unfortunately, the recognition of this interaction crosses subroutine boundaries. The compiler must recognize that the iterations for boundaries should be peeled in the routine to update meshes and that the send portion of the coupling communication updates should be lifted out of the coupling update routines and set down in the mesh update routine. Another option would be a form of optimistic communication. In this case, the coupling specifications can be used to insert communication for the coupled elements every time they are updated and the reception of messages can be verified in the coupling update routine.

The second boundary communication optimization does not require interprocedural analysis. The coupling specification may also be used as an explicit upper bound on the necessary communication by compilers that can not determine more precise communication bounds. This use of the coupling directive is analogous to the use of the independent specification for loops in High Performance Fortran. Although it is not an executable statement, it allows optimization that would be incorrect if the specification is incorrect.

The third issue deals with efficiency in compilation. Due the the large number of clones generated for coupling update routines, it is essential that separate compilation is available. With separate compilation, when a routine is modified and recompiled, the preprocessor must transform the edited routine, including generation of all the necessary clones, and then the compiler must compile all of the clones. If the routines are not separately compiled, then every routine that is passed a distributed mesh would have to be transformed, including generation of all the necessary clones and then the compiler would have to recompile the entire program with its multitude of clones. At this point, it is clear why the transformations to the program should be independent of the specific mesh specifications and distributions and why separate compilation is necessary.

6.4 Algorithm Validation

Since there is not yet an HPF compiler that can correctly generate code for these applications, measures of load balance and communication are presented in place of timing results. The measure of load balance that is presented is the total number of floating point additions, multiplications, and divisions. The load balance measure is compared for the processor with the most work ($Comp_{max}$) and the processor with the least work ($Comp_{min}$). The following measures of communication will be presented:

- *distance* is the maximum distance between any pair of communicating processors;
- *neighbors* is the maximum number of processors communicated with for any processor;
- $Comm_{max_1}$ is the maximum amount of communication, in bytes, for any process3or;
- $Comm_{max_2}$ is the maximum communication, in bytes, between any pair of processors; and
- $Comm_{total}$ is the total communication, in bytes, for the simulation.

For each of these measures, I present the value obtained without use of the packing algorithm, the value obtained with use of the packing algorithm and finally, where appropriate, the percentage improvement. Note that to improve $Comp_{min}$ it must be increased. The entries labeled *large* and *small* refer to the results obtained when not using the packing algorithm depending on whether the primary algorithm used was the large mesh distribution algorithm or the topology-based small mesh distribution algorithm, respectively. The entries labeled *packed* refer to the results obtained when packing was applied. All entries present values for all meshes, therefore, the results for the small mesh distribution algorithm will not be the same as those presented in Chapter 5. One further note on the small mesh algorithm, when packing is not in effect and the size of the mesh gets too big relative to the number of processors being used, then the mesh is distributed over all processors as a large mesh. This can make the speedup smaller as the coupling alignment algorithm shifts big meshes to overlap. This does reduce communication, but it increases the percentage of the work on some processors. In the tables of results, the selected processor configuration is shown below the algorithm identification.

The speedup bound, $Dist_{sb}$, is the speedup, with the data distribution supplied, that would be achieved if there were no communication, i.e.,

$$Dist_{sb} = \frac{Comp_{max}\ for\ n\ processors}{Comp_{max}\ for\ one\ processor}.$$

The speedup bound is rounded to two decimal places. $Dist_{time}$ is the execution time of the algorithm being used to solve the problem.

6.4.1 Simulation of Material Flow in an Elbow with Vanes

This first data set provides a hint of one thing about the packing algorithm. Packing can improve the speedup bounds. In fluid/aerodynamics simulations, the improvements are not as impressive as will be seen for the reactor simulations because there are no small meshes to put in the holes that are left after packing. The speedup bound, $Dist_{sb}$, improvement will be further illustrated with the other test problems.

	Number of Processors					
	1024			2048		
	large (16x16x4)	*packed* (16x16x4)	% *imp.*	*large* (32x16x4)	*packed* (32x32x2)	% *imp.*
distance	3	18		15	23	
neighbors	8	9		9	14	
$Comm_{max_1}$	9,933	7,834	21.1	5,613	5,052	10.0
$Comm_{max_2}$	2,280	1,776	22.1	2,016	1,176	41.7
$Comm_{total}$	5,062,386	3,835,926	24.2	5,690,376	4,697,616	17.4
$Comp_{min}$	0	0	0	0	0	0
$Comp_{max}$	14,022	12,768	8.9	7,182	6,612	7.9
$Dist_{sb}$	746.22	759.99		1456.91	1582.51	
$Dist_{time}$	0.399	0.142		0.034	3.1906	

Figure 6.23 Validation results from flow through an elbow with 2 vanes.

6.4.2 Simulation of Aerodyanmics over Fuselage-Inlet-Nozzle of an F15e Aircraft

In the case of the Fuselage-Inlet-Nozzle simulation of the F15e, there are too few processors for the packing algorithm to come into practical use. In fact, the load balance gets worse with the use of the packing algorithm until 2,048 processors are used. This is due to the fact that five meshes have been packed. Those meshes could not be evenly distributed individually. When they are packed into two larger ones, the larger ones are less evenly distributed. For the larger mesh the number of elements per processor on busy processors is greater than the sum of the elements assigned from the smaller meshes.

The algorithm can be expensive. This applies in particular to higher dimensional meshes with large sizes. This is the only situation where the size of a mesh changes the execution time of one of the automatic distribution algorithms. This long and

	Number of Processors					
	256			512		
	large (32x4x2)	*packed* (32x4x2)	% *imp.*	*large* (32x8x2)	*packed* (32x8x2)	% *imp.*
distance	19	16		20	19	
neighbors	25	26		29	34	
$Comm_{max_1}$	72,567	72,402	0.2	38,822	40370	-4.0
$Comm_{max_2}$	26,544	26,976	-1.6	13,776	13,752	1.5
$Comm_{total}$	9,201,546	9,167,698	0.4	9,783,744	10,148,412	-3.7
$Comp_{min}$	0	0	0	0	0	0
$Comp_{max}$	117,686	117,914	-0.3	60,306	60,724	-0.7
$Dist_{sb}$	230.47	230.02		449.76	446.66	
$Dist_{time}$	0.08	1.88		0.102	4.39	

	Number of Processors					
	1024			2048		
	large (32x8x4)	*packed* (32x8x4)	% *imp.*	*large* (32x32x2)	*packed* (64x8x4)	% *imp.*
distance	19	18		30	33	
neighbors	21	19		31	24	
$Comm_{max_1}$	28,404	29,013	-2.1	11,918	19,832	-66.4
$Comm_{max_2}$	7,000	7,000	0	3,048	5,428	-78.1
$Comm_{total}$	14,118,076	14,370,702	-1.8	11,771,085	20,103,130	-82.1
$Comp_{min}$	0	0		0	0	0
$Comp_{max}$	30,856	31,160	-1.0	16,758	15,694	6.3
$Dist_{sb}$	879.02	870.45		1,618.52	1,728.25	
$Dist_{time}$	0.112	3.58		0.197	54.6	

Figure 6.24 Validation results from F15e Fuselage-Inlet-Nozzle aerodynamic simulation.

size dependent runtime of the automatic distribution algorithm is due directly to the difficulty of the problem being solved and the use of CPLEX to obtain a solution.

6.4.3 Simulation of Aerodyanmics over a Complete F15e Aircraft

From the runtimes presented for 32 to 256 processor runs, the execution time for the packing algorithm is seen to be prohibitive on this problem for more processors. Also notice that more processors need to be used before packing becomes effective on this problem. This indicates that more algorithmic development is needed. If the prepacking heuristic can be extended to significantly reduce the number of meshes that are input to the local linear optimization procedure in cases such as this one where the meshes are essentially all different odd sizes, then it would be reasonable to apply this approach to these general aerodynamic simulation data sets.

6.4.4 3-d LOFT Reactor Model

This first reactor simulation data set indicates two important trends that will be seen in the other three reactor simulation data sets. First, in the 128 processor case, a 2-dimensional distribution is selected even though the reactor core is 3-dimensional. This is because the packing of the meshes favors a 2-dimensional distribution for this problem. There will be more significant illustrations of the performance implications of this phenomenon shortly. Second, when this conflict is overcome, there is significant improvement in not only the speedup bound, $Dist_{sb}$, but also communication measures.

6.4.5 1-d LOFT Reactor Model

For 1-d LOFT, the packing algorithm packs into a 1-dimensional mesh. This is perfect for improving the performance on this problem. Not only does the speedup bound, $Dist_{sb}$, improve, but also the communication measures improve. This particular problem illustrates the potential for these algorithms when the dimensionality of the packing matches the dimensionality of the large meshes.

This particular data set also illustrates a problem that was mentioned earlier. Since a greater percentage of the work gets shifted onto the same processors (as part of the large mesh distribution algorithm), the speedup bound, $Dist_{sb}$, decreases as more processors are added when packing is not used.

	Number of Processors					
	32			64		
	large (16x2)	*packed* (16x2)	% *imp.*	*large* (16x4)	*packed* (16x4)	% *imp.*
distance	9	9		10	10	
neighbors	29	29		56	59	
$Comm_{max_1}$	376,244	372,834	0.9	293,874	293,768	< 0.1
$Comm_{max_2}$	135,192	136,800	-1.2	77,640	76,992	0.8
$Comm_{total}$	5,991,572	5,939,256	0.9	9,353,564	9,330,136	64.4
$Comp_{min}$	964,136	961,856	$< -.1$	428,184	428,754	0.1
$Comp_{max}$	1,589,578	1,594,138	-0.3	811,642	814,264	-0.3
$Dist_{sb}$	30.36	30.27		59.45	59.26	
$Dist_{time}$	0.239	50.17		0.403	14,735.8	

	Number of Processors					
	128			256		
	large (16x4x2)	*packed* (16x4x2)	% *imp.*	*large* (16x8x2)	*packed* (32x4x2)	% *imp.*
distance	11	11		12	19	
neighbors	83	83		119	91	
$Comm_{max_1}$	229,931	235,003	-2.2	118,518	166,931	-40.8
$Comm_{max_2}$	46,128	46,128	0	25,752	45,264	-75.8
$Comm_{total}$	14,419,578	14,706,807	-2.0	14,968,986	20,916,801	-39.7
$Comp_{min}$	214,054	210,596	-1.6	65,664	50,046	-31.2
$Comp_{max}$	415,720	428,260	-3.0	218,500	227,848	-4.3
$Dist_{sb}$	116.07	112.67		220.84	211.78	
$Dist_{time}$	0.39	18,276.6		0.367	34,018.3	

Figure 6.25 Validation results from full F15e aerodynamic simulation.

	Number of Processors					
	64			128		
	small (16x4)	*packed* (16x4)	% *imp.*	*small* (16x4x2)	*packed* (32x4)	% *imp.*
distance	9	10		9	18	
neighbors	9	23		14	34	
$Comm_{max_1}$	50,402	88,242	-75.1	38,130	50,582	-32.6
$Comm_{max_2}$	12,622	25,050	-100.7	12,722	12,534	1.5
$Comm_{total}$	386,281	420,857	-9.0	273,106	343,925	-25.9
$Comp_{min}$	0	87,514		0	0	
$Comp_{max}$	1,328,507	929,725	30.0	1,142,255	799,095	30.0
$Dist_{sb}$	22.79	32.57		26.51	37.89	
$Dist_{time}$	2.465	8.836		1.132	5.993	

	Number of Processors					
	256			512		
	small (16x4x4)	*packed* (64x4x1)	% *imp.*	*small* (32x4x4)	*packed* (64x4x2)	% *imp.*
distance	8	33		17	30	
neighbors	22	52		26	63	
$Comm_{max_1}$	38,261	37,942	0.8	25,900	571	97.8
$Comm_{max_2}$	12,774	12,518	2.0	12,834	56	99.6
$Comm_{total}$	235,261	173,657	26.2	273,007	102,495	62.5
$Comp_{min}$	0	0		0	0	
$Comp_{max}$	1,074,133	668,465	34.3	1,099,137	432,205	60.7
$Dist_{sb}$	28.19	45.29		27.55	70.05	
$Dist_{time}$	1.113	6.167		3.319	13.262	

Figure 6.26 Validation results for 3-D LOFT reactor model.

	Number of Processors					
	32			64		
	small (32)	*packed* (32)	% *imp.*	*small* (64)	*packed* (64)	% *imp.*
distance	6	16		22	32	
neighbors	16	15		12	30	
$Comm_{max_1}$	100,400	62,654	37.6	88,390	50,240	43.2
$Comm_{max_2}$	25,172	25,032	-0.6	25,412	12,534	49.3
$Comm_{total}$	542,745	551,955	-1.7	325,691	393,921	-20.9
$Comp_{min}$	12,502	50,008	75.0	0	12,502	
$Comp_{max}$	268,154	155,636	57.7	430,680	248,762	42.2
$Dist_{sb}$	8.85	15.25		5.51	9.54	
$Dist_{time}$	0.499	0.084		0.381	0.093	

	Number of Processors					
	128			256		
	small (128)	*packed* (128)	% *imp.*	*small* (256)	*packed* (256)	% *imp.*
distance	22	62		13	124	
neighbors	17	56		18	84	
$Comm_{max_1}$	38,630	37,870	2.0	1,416	452	68.1
$Comm_{max_2}$	13,026	12,522	3.9	656	16	97.6
$Comm_{total}$	157,261	196,351	-24.9	179,244	56,540	68.5
$Comp_{min}$	0	0	0	0	0	0
$Comp_{max}$	518,194	130,632	74.8	618,210	118,130	80.9
$Dist_{sb}$	4.58	18.17		3.84	20.10	
$Dist_{time}$	0.531	0.109		0.161	0.076	

Figure 6.27 Validation results for 1-D LOFT reactor model.

6.4.6 H.B. Robinson Reactor Model

Improving the speedup bound, $Dist_{sb}$, does not necessarily produce improved runtime. In the 16 processor case of the H.B.Robinson model simulation, the 156% increase in total communication may cost more than the 22.5% improvement in load balance yields. One way that this problem may be addressed is by improving the small mesh distribution algorithm when medium meshes are present. This will be discussed in Chapter 7.

The results for H.B.Robinson model are poor on 64 processors as packing of the 1-dimensional meshes favors a 1-dimensional distribution, but a 3-dimensional distribution is favored for reactor core consideration. This suggests an extension to the algorithm that will be discussed briefly in the future work section of Chapter 7.

6.4.7 Westinghouse AP600 Reactor Model

The results for the AP600 simulation are somewhat disappointing, but in hindsight understandable. The packing algorithm does a 2-dimensional packing if there are no meshes to pack of higher dimensionality. For this reason, when 64 or 128 processors are used, there is a direct conflict between the processor layout that is preferred for the reactor core and the one that is preferred for the large packed mesh. Although the problem is present with the unpacked algorithm (as indicated by the limited speedups), it is amplified by packing. To lessen this effect, the packing algorithm should be extended to give substantial preference to 3-dimensional packings. Actually, substantial preference should be given to n-dimensional packings where n is the dimensionality of the highest dimensionality large mesh. If there are no large meshes, then the easiest packing should be done.

Even without this extension, the use of the packing algorithm can improve the speedup bound, $Dist_{sb}$, by as much as 10 on this problem. As more processors are applied, the packing algorithm should improve over the results for the non-packing algorithm since it will pass the point where the 2-dimensional distribution becomes more effective. Further, as more processors are applied to the problem, the packing algorithm can take advantage of more wasted processors.

6.5 Chapter Summary

Distribution algorithms for all combinations of different sizes of meshes have now been presented. The performance measures for the medium mesh results make a discussion of the applicability of packing desirable. There are three factors that determine how well the packing approach will work for medium size meshes. The first factor is

	Number of Processors					
	8			16		
	small (4x2)	*packed* (4x2)	% *imp.*	*small* (8x2)	*packed* (8x2)	% *imp.*
distance	3	3		5	5	
neighbors	4	7		11	11	
$Comm_{max_1}$	276,088	1,157,791	-319.4	363,962	773,400	-112.5
$Comm_{max_2}$	100,516	374,318	-272.4	65,403	221,261	-238.3
$Comm_{total}$	741,358	2,966,240	-300.1	1,058,385	2,709,363	-156.0
$Comp_{min}$	3,825,812	3,827,094	< 0.1	665,652	1,913,553	65.2
$Comp_{max}$	3,836,791	3,838,073	-0.03	2,484,558	1,926,055	22.5
$Dist_{sb}$	7.98	7.99		12.34	15.92	
$Dist_{time}$	4.689	0.129		3.061	0.383	

	Number of Processors					
	32			64		
	small (8x4)	*packed* (8x4)	% *imp.*	*small* (16x2x2)	*packed* (32x2)	% *imp.*
distance	6	9		7	17	
neighbors	9	13		11	21	
$Comm_{max_1}$	314,350	567,666	-80.6	168,528	263,434	-56.3
$Comm_{max_2}$	88,110	208,651	-136.8	65,439	25,112	61.6
$Comm_{total}$	1,327,335	2,510,668	-89.2	1,385,777	1,631,238	-17.7
$Comp_{min}$	262,542	908,812	71.1	0	112,518	
$Comp_{max}$	1,801,023	1,035,355	42.5	1,409,127	2,024,295	-43.7
$Dist_{sb}$	17.02	29.61		21.75	15.14	
$Dist_{time}$	1.654	0.07		1.566	0.198	

Figure 6.28 Validation results for H.B. Robinson reactor model.

	Number of Processors					
	32			64		
	small (8x4)	*packed* (4x2x2)	% *imp.*	*small* (16x4)	*packed* (16x4)	% *imp.*
distance	6	6		10	10	
neighbors	22	26		20	20	
$Comm_{max_1}$	2,697,672	2,567,018	4.8	1,184,538	1,373,460	-15.9
$Comm_{max_2}$	208,511	208,671	$<$ -0.1	522,648	535,166	-2.4
$Comm_{total}$	10,638,084	9,977,937	6.8	8,067,823	7,982,898	1.1
$Comp_{min}$	3,929,684	5,915,914	34.6	100,016	1,071,338	90.7
$Comp_{max}$	7,624,060	7,493,430	1.7	6,218,763	815,4690	-31.1
$Dist_{sb}$	25.69	26.15		31.51	24.03	
$Dist_{time}$	0.162	8,654.0		0.09	8,124.15	

	Number of Processors					
	128			256		
	small (16x8)	*packed* (16x8)	% *imp.*	*small* (16x8x2)	*packed* (32x8)	% *imp.*
distance	12	12		10	20	
neighbors	18	20		30	82	
$Comm_{max_1}$	1,095,998	1,171,780	-6.9	1,083,482	523714	51.7
$Comm_{max_2}$	261,326	273,842	-4.8	196,044	65,347	66.7
$Comm_{total}$	8,066,357	8,120,377	-0.7	4,060,758	4,185,695	-3.1
$Comp_{min}$	0	535,030		0	75,012	
$Comp_{max}$	4,176,105	5,567,090	-33.3	3,247,654	2,789,175	14.11
$Dist_{sb}$	46.92	35.19		60.33	70.25	
$Dist_{time}$	0.056	18,332.6		0.521	4,463.35	

Figure 6.29 Validation results for Westinghouse AP600 reactor model.

whether the meshes being packed are the same dimensionality as any large meshes in the problem. If the medium size meshes have the same dimensionality as the large meshes, then the packed mesh(es) will also. This implies that there will not be a conflict between the dimensionalities of the preferred distributions for the meshes.

The second factor in determining the effectiveness of the packing approach is how well the meshes pack. One measure of how well meshes pack is the number and size of the holes left in the bin once it is packed. Packing performs particularly well when subsets of the meshes to be packed are the same size in $n-1$ (or n) dimensions. In this case, the prepacking algorithm packs the subsets of meshes with no wasted space. Thus, prepacking reduces the number of meshes to be packed and this in turn makes packing take less time.

The final factor in determining the effectiveness of packing is whether there are small meshes in the problem. In problems with small meshes, the small meshes are used to fill up the holes left during packing of medium meshes. This factor can even help to improve poor load balance that is due to the other two factors.

All of my autmatic distribution algorithms produce mappings that maintain the regularity of communication involving a single mesh. All of the algorithms guarantee nearest neighbor communication in the specified processor topology for communication involving a single mesh. Further, I have shown how a user program, with no notion of data layout, can be transformed into a standard, machine specific, HPF program with full distribution specifications.

Chapter 7

Conclusion

This chapter reviews the contributions of the research presented in this dissertation, describes future research that addresses the limitations of the results, and sums up the research progress.

7.1 Contributions of the Dissertation

This dissertation has shown that topology is a key to efficient parallelization support for partially regular applications. For linearized applications, this dissertation has provided:

- an example of language extensions that supply sufficient information to the compiler for deciphering linearized array and index array usage;
- compiler technology to support, via the language extensions, the natural topology (multi-dimensional) data distribution of linearized arrays with regular application optimization; and
- experimental results that illustrate the actual performance gains that can be achieved via this linearized application support.

For composite grid application, this dissertation has provided:

- descriptions of a set of composite grid applications that may be used (with the exception of the AP600 data, which is proprietary and therefore can not be released) as a test suite for ICRM support;
- a template for HPF composite grid code development that, along with the style guideline, specifies a form for applications to allow use of the transformation and automatic distribution support;
- a transformation procedure that converts HPF composite grid applications, with the proper form but no data layout specifications, into standard HPF programs with fully specified data layout;

- a final HPF form for composite grid programs that allows input of distribution specifications in the program and all regular application optimization in the HPF compiler;
- a set of distribution algorithms that determine how to map composite grid applications onto parallel architectures based on the problem and machine topology;
- experimental results that illustrate the applicability, potential, and limitations of the automatic distribution algorithms.

Next, future research is described that addresses the limitation that these experiments have revealed.

7.2 Future Research

The general observation that the data mappings automatically generated are complicated to understand leads me to consider the development of a visualization tool. One possibility for such a tool would be to create a "virtual" processor environment that a user can explore with each room in the environment being a processor. Inside of each room could be displayed graphical statistics about the computation and communication associated with the processor. The couplings between computations on the various processors could be represented as virtual doorways that lead to coupled processors with statistics attached to each virtual doorway on the cost associated with the "trip through the door".

For problems like nuclear reactor simulation, the computational load is dynamic. This implies that future research should include dynamic load balancing. To this end, parallel versions of all of the distribution algorithms should be developed. In addition, statistics on computation associated with meshes should be gathered to allow load rebalancing according to the execution of the problem as it progresses.

My other plans for future research involve specific parts of the distribution algorithms. For this reason, the remaining future research will be discussed in the context of medium and small size meshes separately.

7.2.1 Future Medium Mesh Algorithm Research

Medium size meshes can occur in another form in applications such as water-cooled nuclear reactor simulations, where meshes are often grouped into subsystems. For example, there are eleven meshes (including heat structures) grouped together in the the Passive Residual Heat Removal System in the AP600 [LB94]. For problems with subsystem specifications, this grouping may be taken advantage of as an extra level of topological information. Subsystems are normally much more tightly coupled internally than the coupling between the subsystems. In this case, each subsystem might be considered as a single medium size mesh. Using this information, locality may be improved by allocating a block of processors to each subsystem. The small meshes in the subsystem could then be allocated to those processors using one of the small mesh ICRM distribution algorithms. This might be implemented as a two level packing problem. First meshes in a subsystem are packed into a single mesh. Then the subsystem meshes are packed into large meshes, which can finally be distributed over all processors.

One of the lessons learned in this research was that using the iterative local linear optimization to solve the non-linear packing problem is expensive. Future work will include exploration of heuristics for packing. The first step of one such heuristic is to pack together any pair of n-dimensional meshes with $n - 1$ dimensions that are the same size by stacking the other dimension. This first step in a general packing heuristic was used to reduce the number of meshes packed in the optimization approach described in Chapter 6. Along with the development of heuristics for packing, efficacy of the heuristics versus the optimization approach should be studied. This might lead to insight that would allow for automatic selection of the most appropriate version of packing algorithms according to the input.

There is a direct conflict between the processor layout that is preferred for the 3-dimensional large meshes and the one that is preferred for the large packed meshes when the packed meshes are 1- or 2-dimensional. Although the problem is present with the unpacked algorithm, it is amplified by packing. To lessen this effect, the packing algorithm should be extended to give substantial preference to n-dimensional packings where n is the minimum dimensionality of the large meshes. This observation would also apply to heuristic approaches. By packing into bins of the same dimensionality as the large meshes, the only algorithmic factor in applicability of packing is eliminated. It must be noted, however, that packing into a specific dimensionality can effect how well the meshes pack. For example, two 1-dimensional meshes of size 17 and 22 pack perfectly into a 1-dimensional bin. They do not, however, pack perfectly into a 2-dimensional bin. This is the reason that a substantial preference is given to packing into an n-dimensional bin, rather than making it a requirement. With a fast heuristic packing approach, the tradeoff between better packing (less wasted space) and better dimensionality (matching the n-dimensional meshes) could be evaluated and the solution with the best predicted runtime could be chosen.

Although packing has some drawbacks, it is still the only way to parallelize these applications for large numbers of processors (relative to mesh size) that generates a regular distribution. This makes compilation in HPF possible. This also allows the program to be optimized using all of the regular application optimizations.

7.2.2 Future Small Mesh Algorithm Research

In some cases, load imbalance occurs when using the general small mesh mapping algorithm because it saves meshes for mapping at the end when it can not find a good placement quickly. This could be improved by keeping a minimum heap of processors and placing the meshes on the least loaded processor when a good placement is not found quickly. This would probably only be applied in the case when the mesh is relatively large compared to other small meshes. Truly small meshes should still be kept for load balance fillers.

As can be seen by the increase in communication for nuclear reactor problems, when the packing algorithm is used, there is a need to extend the general small mesh algorithm so it can be used to reduce coupling communication costs while balancing the load after large (and packed medium) meshes are distributed. There are straightforward and expensive ways to incorporate the coupling based distribution with the load balance measures from large mesh distribution. More efficient ways should be explored.

7.3 Final Remarks

Although there is still a great deal of research to be done in the area of efficient parallelization support for composite grid problems, I have made significant progress in moving from user-distributed irregular support to automatically-distributed regular support and optimization.

Bibliography

[All88] S.E. Allwright. Techniques in multiblock domain decomposition and surface grid generation. In S. Sengupta, J. Hauser, P.R. Eiseman, and J.F. Thompson, editors, *Numerical Grid Generation in Computational Fluid Mechanics '88*, pages 559–568. Pineridge Press, 1988.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[BSL85] B.E. Boyack, H. Stumpf, and J.F. Lime. *TRAC User's Guide.* Los Alamos National Laboratory, Los Alamos, New Mexico 87545, 1985.

[BSS90] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. ICASE Report 90-41, Institute for Computer Application in Science and Engineering, Hampton, VA, May 1990.

[CCSR92] C. Chase, K. Crowley, J. Saltz, and A. Reeves. Compiler and runtime support for irregularly coupled regular meshes. *Journal of the ACM*, July 1992.

[CN93] Geoff Chesshire and Vijay K. Naik. An environment for parallel and distributed computation with application to overlapping grids. Technical Report RC 18987 (82949), IBM Reasearch Division, T.J. Watson Research Center, Almaden, June 1993.

[Com74] Louis Comtet. *Advanced Combinatorics.* D. Reidel Publishing Company, Boston, Mass., 1974.

[CPLEX O94] CPLEX Optimization, Inc. *Using the* CPLEX *Callable Library*, 1994. Version 3.0.

[Dag93] Leonardo Dagnum. Automatic partitioning of unstructured grids into connected components. In *Proceedings of Supercomputing '93*, Seattle, November 1993.

[Eri87] Lars-Erik Eriksson. Flow solution on a dual-block grid around an airplane. *Computer Methods in Applied Mechanics and Engineering*, 64:79–83, 1987.

[FHK$^+$90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[FL93] C. Farhat and M. Lesoinne. Mesh partitioning algorithms for the parallel solution of partial differential equations. *Applied Numerical Mathematics*, 12(5):443–457, May 1993.

[FO84] G. Fox and S. Otto. Algorithms for concurrent processors. *Physics Today*, 37:50–59, May 1984.

[HB91] D.W. Heermann and A.N. Burkitt. *Parallel Algorithms in Computational Science.* Springer-Verlag, New York, NY, 1991.

[HHKT92] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, November 1992.

[HKT91a] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT91b] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, January 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[HKT91c] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors.* Elsevier, Amsterdam, The Netherlands, to appear 1991.

[JS83] J.E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.

[JWL94] Susan J. Jolly-Woodruff and James F. Lime. Los Alamos National Laboratories, March 1994. Private communication.

[Kel94] Joe Kelly. Nuclear Regulatory Commission, January 1994. Private communication.

[KLS$^+$94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, Massachusetts, 1994.

[Koe90] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[LB94] J.F. Lime and B.E. Boyack. TRAC large-break loss-of-coolant accident analysis for the AP600 design. In *International Topical Meeting on Advanced Reactors Safety*, Pittsburgh, PA, April 1994.

[LHKD91] L.M. Liebrock, D.L. Hicks, K.W. Kennedy, and J.J. Dongarra. Using problem and algorithm topology in parallelization. Technical Report 91-166, Rice University, Center for Research in Parallel Computation, Houston, TX, August 1991.

[LK94a] L.M. Liebrock and K. Kennedy. Automatic data distribution of large meshs in coupled grid applications. Technical Report 94-395, Rice University, Center for Research in Parallel Computation, Houston, TX, April 1994.

[LK94b] L.M. Liebrock and K. Kennedy. Parallelization of linearized application in Fortran D. In *International Parallel Processing Symposium '94*, pages 51–60, Washington, DC, April 1994.

[LMS93] J.C. Lin, V. Martinez, and J.W. Spore. *TRAC Developmental Assessment Manual.* Los Alamos National Laboratory, Los Alamos, New Mexico 87545, 1993.

[Lub93] Olaf Lubeck. Computer Research and Development, Los Alamos National Laboratories, February 1993. Private communication.

[MF94] Nashat Mansour and Geoffrey C. Fox. Parallel physical optimization algorithms for allocating data to multicomputer nodes. *The Journal of Supercomputing*, 8:53–80, 1994.

[Mou93] Vince Mousseau. EG&G Idaho, Idaho National Engineering Laboratories, March 1993. Private communication.

[NZ80] Ivan Niven and H.S. Zuckerman, editors. *An Introduction to the Theory of Numbers.* John Wiley and Sons, New York, NY, fourth edition, 1980.

[OR94a] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. Technical report, Northeast Parallel Architecture Center, Syracuse University, August 1994.

[OR94b] Chao-Wei Ou and Sanjay Ranka. Parallel remapping algorithms for adaptive problems. Technical report, Northeast Parallel Architecture Center, Syracuse University, 1994.

[PSL90] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *Siam Journal on Matrix Anal. Appl.*, 11(3):430–452, July 1990.

[Ram92] Marcelo Ramé. Computational and Applied Mathematics Department, Rice University, December 1992. Private communication: This code was supplied as representative of the computations in UTCOMP.

[Ram93] Marcelo Ramé. Computational and Applied Mathematics Department, Rice University, February 1993. Private communication.

[RAP87] D. Reed, L. Adams, and M. Patrick. Stencils and problem partitioning: Their influence on performance of multiprocessor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[Sal93] Joel Saltz. Computer Science Department, University of Maryland, February 1993. Private communication.

[SB87] Joseph L. Steger and John A. Benek. On the use of composite grid schemes in computational aerodynamics. *Computer Methods in Applied Mechanics and Engineering*, 64:301–320, 1987.

[SE87] Robert E. Smith and Lars-Erik Eriksson. Algebraic grid generation. *Computer Methods in Applied Mechanics and Engineering*, 64:285–300, 1987.

[SG92] Mark S. Shephard and Marcel K. Georges. Reliability of automatic 3d mesh generation. *Computer Methods in Applied Mechanics and Engineering*, 101:443–462, 1992.

[SGCS93] Alan Sussman, S. Gupta, Kay Crowley, and Joel Saltz. *A Manual for the Multiblock PARTI Runtime Primitives.* Institute for Computer Application in Science and Engineering, NASA Langley Research Center, revision 3 edition, 1993.

[SGW88] J.A. Shaw, J.M. Georgala, and N.P. Weatherill. The construction of component adaptive grids for aerodynamic geometries. In S. Sengupta, J. Hauser, P.R. Eiseman, and J.F. Thompson, editors, *Numerical Grid Generation in Computational Fluid Mechanics '88*, pages 383–394. Pineridge Press, 1988.

[SKC88] J.P. Steinbrenner, S.L. Karmen, and J.R. Chawner. Generation of multiple block grids for arbitrary 3-d geometries. In H. Yoshihara, editor, *3-Dimensional Grid Generation for Complex Configurations.* AGARDograph 309, 1988.

[SW92] J.A. Shaw and N.P. Weatherill. Automatic topology generation for multiblock grids. *Applied Mathematics and Computation*, 52:355–388, 1992.

[Tho87] Joe F. Thompson. A general three-dimensional elliptic grid generation system on a composite block structure. *Computer Methods in Applied Mechanics and Engineering*, 64:377–411, 1987.

[Thu93] Michael Thuné. A partitioning algorithm for composite grids. *Parallel Algorithms and Applications*, 1(1):69–81, 1993.

[vKK$^+$92] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[Wol84] G. Wolfgang. *Notes on Numberical Fluid Mechanics, Volumne 8: Vectorization of Computer Programs with Applications to computational Fluid Dynamics*. Friedr. Vieweg and Sohn, Wiesbaden, Germany, 1984.

[Yam90] Yukimitsu Yamamoto. Numerical simulation of hypersonic viscous flow for the design of H-II orbiting plane (HOPE). *Computer Methods in Applied Mechanics and Engineering*, 89:59–72, 1990.

[ZI93] A. Zaafrani and M.R. Ito. Partitioning the global space for distributed memory systems. In *Proceedings of Supercomputing '93*, Seattle, November 1993.

Appendix A

Modeling Parallel Computation

A.1 Introduction

Parallel computations that are composed of one or more irregularly coupled regular meshes (ICRMs) are modeled. This model is used to determine whether a distribution of an ICRM problem is a good one. Further, it was used to evaluate and improve algorithms for finding distributions of ICRM problems.

A.2 Assumptions

Assumptions for the model are as follows. First, the simulation problem in the computation must be static. The amount of computation is the same for each element of each DECOMPOSITION. The computation does not have to be uniform across all DECOMPOSITIONs, just internally for each individual DECOMPOSITION. It is further assumed that all of the various operations can be grouped into at most three groups with all components of each group having approximately the same runtime. Each pair of "neighbors" in each dimension of a DECOMPOSITION performs the same set of communications. This does not imply that the communication is the same in rows and columns, but that the amount of communication in each row is the same as that in every other row and similarly for the columns. Further, each pair of "neighbors" in a given coupling between any fixed pair of DECOMPOSITIONs perform the same set of communications. Index arrays will always be mapped the same way as the arrays they are used to index if they are used to index intra-DECOMPOSITION variables. This does not imply that communication will not be necessary for intra-DECOMPOSITION index arrays. If index arrays are used to index inter-DECOMPOSITION variables, then they may be mapped differently than the variables they index and some communication may be required.

A.3 Notation

Given a set of coupled DECOMPOSITIONs, $D = \{D_1, D_2, ..., D_N\}$ and a set of processors, $P = \{P_1, P_2, ..., P_M\}$, the following notation is used in the model.

For each DECOMPOSITION, D_i, define the following quantities. The elements of D_i are e_{ij} for $1 \leq j \leq |D_i|$. The number of computations performed for e_{ij} is $add_i + function_i + divide_i$ where add_i is the number of computations in the fastest operation group, $function_i$ is the number of computations in the medium speed operation group, and $divide_i$ is the number of computations in the slowest operation group. The number of memory cells needed for storing variables associated with element e_{ij} is $|e_{ij}|$. The set of communications between neighbors e_{ij} and e_{ik} in D_i is denoted $comm_{id}$, where d is the dimension for communication. The set of communications between neighbors e_{ij} in D_i and e_{jk} in D_j is denoted $comm_{ij}$.

Next, define the following terms to simplify the modeling discussion. Two DECOMPOSITIONs that are neighbors in the simulation are defined to be "coupled". When neighboring elements in coupled DECOMPOSITIONs are located on different processors, communication will result. The set of elements for which communication is necessary in such a mapping is defined as follows:

$$Coupling(D_i, D_j) = \{(e_{ik}, e_{jl}) \mid e_{ik} \in D_i \ \& \ e_{jl} \in D_j \ \& \ e_{ik} \ is \ coupled \ to \ e_{jl}\}$$

For each element of each DECOMPOSITION there is one processor, its owner, on which it is permanently stored. The owner of an element performs all computations for that element. The owner of an element e_{ij} is defined by:

$$Map(e_{ij}) = P_k \equiv e'_{ij}s \ owner.$$

This is just following the owner-computes rule.

A.4 The Model

A.4.1 The Complete Model

The components being modeled are: communication, computation, indirection and the inspector. In all of the components, the maximum over all processors is used to get a worst case upper bound on the total time of execution. This will allow comparison of different distributions of the same problem and selection of the better one.

Communication represents the cost associated with a series of "gets", in which communication is grouped to reduce overhead where possible. Communication occurs when sub-blocks of a DECOMPOSITION are mapped to different processors and when coupled DECOMPOSITIONs (or coupled sub-blocks thereof) are mapped to different processors.

Computation represents the cost of performing all of the computations associated with the elements of decompositions mapped to a given processor. This is simplified to use only three types of computations with only three associated execution times.

Indirection represents the cost of extra communication associated with the use of any index arrays.

Inspector represents the cost of running an inspector to determine what communication is necessary when compile-time techniques are not sufficient for such determination. When indirection is not used, the inspector should not be needed.

For the complete model, a simple sum of the component terms is sufficient:

$$\begin{aligned} Cost_{Model}(Map) &= Computation(Map) + Communication(Map) \\ &\quad + Indirection(Map) + Inspector(Map). \end{aligned}$$

A.4.2 Communication

Modeling of communication is based on the use of the "get" as opposed to the use of send/receive pairs.

In order to effectively model communication, a two level approach is used. At the bottom, fine granularity, the cost of individual communications is modeled. At the top, large granularity, the cost of the set of communications imposed by a particular distribution is modeled.

For communication modeling, the following definitions apply.

- Let $hops$ be the distance, in number of steps, between the source and sink for a communication.
- Let $cost_{neighbor}$ be the cost for communicating between nearest-neighbor processors.
- Let $hops_{general}$ be the maximum number of steps used in general communication between any pair of processors.
- Let $cost_{byte}$ be the cost for communication of one byte between nearest-neighbor processors. This is strictly greater than zero.
- Let $bytes$ be the number of bytes being communicated.
- Let $cost_{startup\ constant}$ be the startup constant that is associated with calling the "get" command regaurdless of what is being gotten or where it is located.

- Let $constant_{transfer}$ be the constant associated with buffering data on intermediate processors between the source and destination. This is used, for example, to model the pipelining effect in communication.

All of the constants, including the number of steps between any two processors, are a function of the parallel processor being used and are greater than or equal to zero.

Modeling single communications

A single communication on a given architecture may have a variety of costs associated with it depending on the size of the value being communicated, the distance between the processors that are the source and sink of the communication, and the startup cost of communication.

First, the distance between the source and sink of the communication is:

$$H(hops) = \min(hops, hops_{general}).$$

On most architectures $hops_{general}$ will be the maximum distance between any pair of processors. The exceptions to this rule are machines like the Thinking Machines CM-2, which have separate networks for reducing this maximum communication distance.

The total cost for a single communication is modeled as:

$$\begin{aligned} C_T(hops, bytes) &= cost_{startup\ constant} + (H(hops) - 1) * cost_{neighbor} \\ &\quad + bytes * cost_{byte} + bytes * H(hops) * constant_{buffering}. \end{aligned}$$

where the current mapping in effect, Map, determines the location of the source and sink for communication. This is composed of the cost of the communication instruction, the cost of setting up the route for wormhole routing, the cost of buffering the message if necessary, and the cost of buffering in intermediate processors between the source and sink.

Modeling for a Fixed Distribution of Decompositions

Once a set of DECOMPOSITIONs has been mapped to processors, the communication requirements for the simulation are fixed. Each communication fits into one of two categories: between two neighboring elements of a single DECOMPOSITION, which are assigned in the distribution to different processors, or between coupled elements of two neighboring DECOMPOSITIONs when the elements are assigned to different processors.

These two categories correspond, respectively, to the two terms of the total simulation communication model, which is a function of the mapping.

$$Communication(Map) = \max_{P_x} \sum_{P_y} \{C_T\,(\#hops(P_x, P_y), \#bytes\,(COMM_{xy}))\}$$

with

$$COMM_{xy} = \sum_{(e_{ij}, e_{ik})} comm_{id} + \sum_{(e_{ij}, e_{mn})} comm_{im}$$

where

$$Map(e_{ij}) = P_x, \qquad Map(e_{ik}) = P_y, \qquad Map(e_{mn}) = P_y,$$

and

$$(e_{ij}, e_{mn}) \in Coupling(D_j, D_m).$$

Here, $\#hops$ is the number of hops between the two processors involved in communication and $\#bytes$ is the number of bytes of communication. This assumes that all communication across a boundary that is of the same type can and will be grouped into a single vector communication. Such an assumption will give a slightly unfair advantage to random distributions as the extra work to group communications for random distributions will be ignored.

As an example, the communication timings on the Intel i860 resulted in the following approximate values for the timing variables.

Variable	*Timing*
CostNeighbor	0.04
HopsGeneral	2
CostStartupConstant	0.04
ConstantBuffering	0.2
CostByte	0.00077

A.4.3 Computation

At the instruction level, computations are modeled as having one of three fixed costs; so the model for computation is simple:

$$Computation(Map) = \max_{P_m \in P} \left\{ \sum_{e_{ij} \;|\; Map(e_{ij}) = P_m} Comp_i \right\}.$$

with

$$Comp_i = add_i * cost_{add} + function_i * cost_{function} + divide_i * cost_{divide}.$$

The number of computations that take approximately the same time as an addition is add_i. The definitions for $function_i$ and $divide_i$ are similar. This three cost component approach is used as some processors, such as the Intel i860, have groups of operations with three distinctly different costs. For example, on the i860, the following approximate timings were measured:

Operation	*Timing*
multiply	5.3e-04
add	7.4e-04
function	4.2e-03
divide	9.4e-02

Since adds and multiplies take the same order of magnitude of time, adds and multiplies are grouped, for this machine, and classified as "adds" in the model.

A.4.4 Indirection

Indirection is modeled as the cost of doing the "get" when the index array element does not reside on the processor performing the operation. Hence, for $X(IX(I))$ the cost of the indirection is:

$$Indirection(Map) = C_T(\#hops(Map(IX(I)), Map(X(IX(I)))), \#bytes(IX(I)))$$

when $Map(IX(I)) \neq Map(X(IX(I)))$.

A.4.5 Inspector

The inspector's only variation in this model is when index arrays are used. Since this component is handled above via the indirection cost, the inspector overhead is fairly simple:

$$Inspector(Map) = \max_{P_m} \left\{ constant_{inspector} * \sum_{e_{ij} \mid Map(e_{ij})=P_m} |e_{ij}| * |program| \right\},$$

which is the size of the data on processor P_m times the program size. This is actually a rather large upper bound on the cost for using the inspector. If there are no index arrays in the program, then communication can be analyzed at compile time and no inspector is needed.

A.5 Model Validation

A.5.1 Test Problem: 1-Dimensional Explicit Material Dynamics

1-dimensional explicit material dynamics provides an excellent test problem as I have run this problem on a variety of machines with many different distributions. This experience allows me to predict the relationships between the runtimes of various distributions and verify that the model produces valid results. In this test problem there are 32 "adds", 3 "functions", and 7 "divides" for each element in the mesh on each timestep. There is no indirection and there are 56 bytes of communication between neighbors on each timestep.

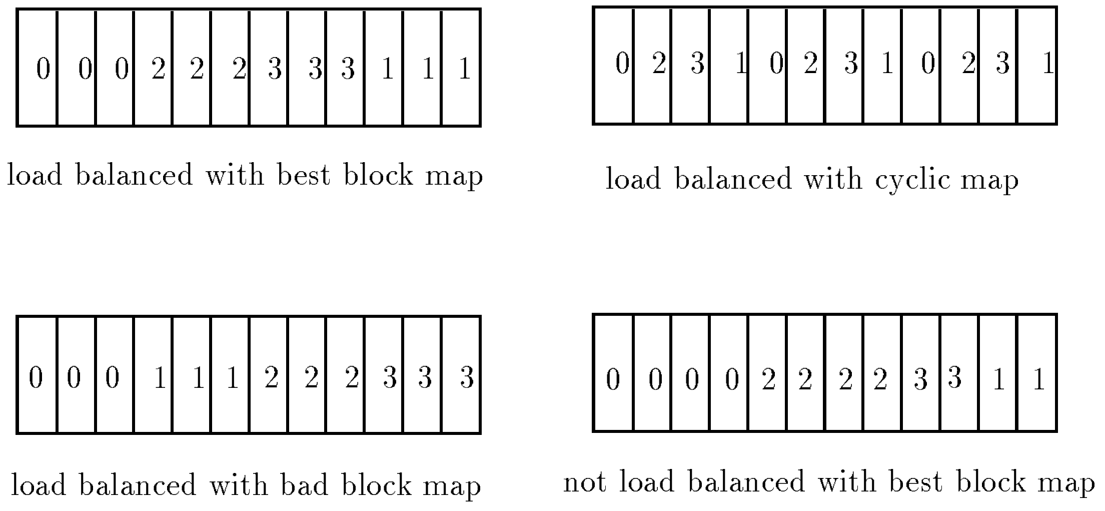

0 0 0 2 2 2 3 3 3 1 1 1
load balanced with best block map
0 2 3 1 0 2 3 1 0 2 3 1
load balanced with cyclic map
0 0 0 1 1 1 2 2 2 3 3 3
load balanced with bad block map
0 0 0 0 2 2 2 2 3 3 1 1
not load balanced with best block map

Figure A.1 Single Mesh Sample Distributions.

Single Decomposition Mapping

Here, the flow of fluid in a single section of pipe is simulated. This is arguably the simplest type of problem for distribution as each element of the decomposition has the same computation and communication pattern (with the exception of minor variations at the boundaries or ends of the pipe).

In this first test case, the model is run on a single mesh with different distributions. The variations on distribution that are modeled include: load balanced distribution with best block map, load balanced with bad block map, load balanced with cyclic map, and not load balanced with best block map. These types of distributions are illustrated in Figure A.1 for a twelve element mesh mapped onto a four processor (hypercube) machine. In the figure each element is numbered with the id of the processor who owns it. A one hundred element problem was used with four processors in the model evaluation. The predicted runtimes for the four different distributions were:

Distribution	*Predicted Time*
load balanced with best block map	17.52
load balanced with cyclic map	19.59
load balanced with bad block map	18.68
not load balanced with best block map	18.21

The ordering of the predicted runtimes is correct according to previous experiments on the Intel (and other machines, such as the Denelcor HEP, the Floating Point Systems Tesseract, the Cray XMP, the Alliant FX/8, and the Sequent Symmetry).

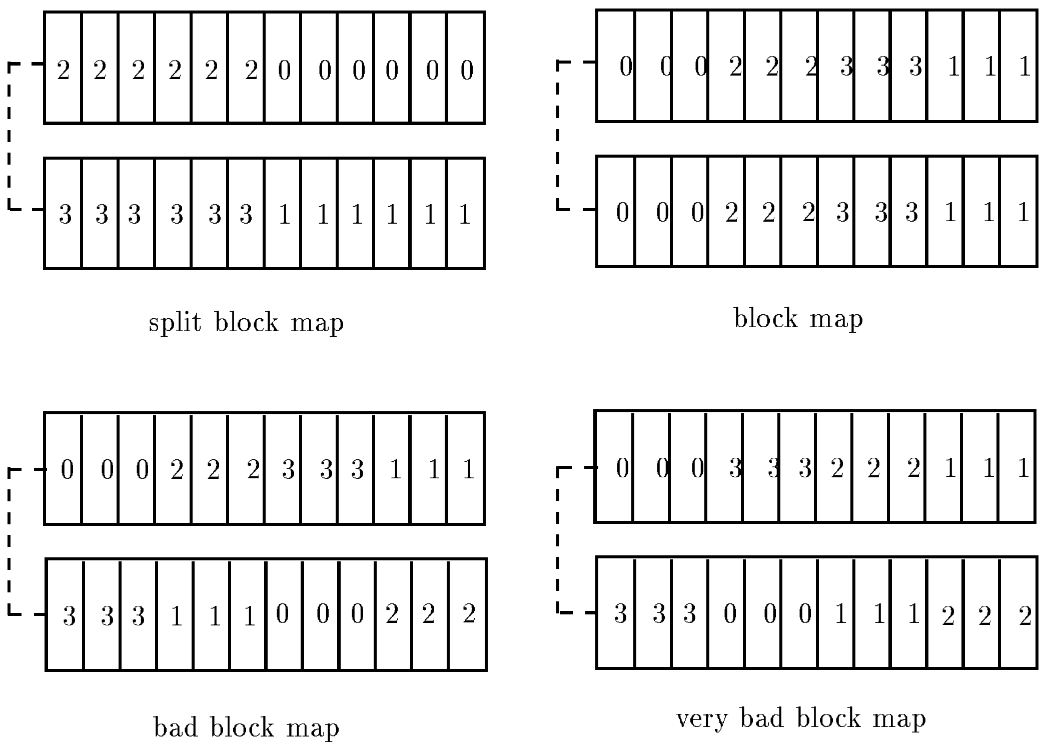


Figure A.2 Two Mesh Sample Distributions.

Multiple Decomposition Mapping

Here, the flow of fluid in two pipes, joined as a "Tee", is simulated. This is similar to the type of computation that occurs on a larger scale in the simulation of water-cooled nuclear reactors.

In this second test case, the model is run on two coupled meshes with four distributions. The variations on distribution that are modeled include: split block map, block map, bad block map, and very bad block map. These types of distributions are illustrated in Figure A.2 for two twelve element meshes mapped onto a four processor (hypercube) machine. In addition to numbering elements with the processor number, coupling between meshes is shown via a dashed line. A one hundred element (per mesh) problem was used with four processors in the model evaluation. The predicted runtimes for the four different distributions were:

Distribution	*Predicted Time*
split block map	34.77
block map	34.89
bad block map	35.30
very bad block map	37.46

In this test problem, there were 56 bytes of communication between nearest-neighbors, but only 16 bytes across the coupling. When there are more communication bytes across the coupling than between neighbors, the block map will outperform the split block map.

A.6 Theorems

Next, theorems about distribution in this model of parallel computation and extensions of the model are presented. These theorems provide direction for the design and comparison of distribution algorithms. Unless otherwise stated, elements are to be equally distributed across $p > 0$ processors with 1- to 3-dimensional decompositions considered.

A.6.1 Standard Model Theorems

The basic setting for all of these theorems is physical simulation applications for static problems unless otherwise stated.

Theorem A.1 Given two distributions with the same overheads due to load balance, context switching, indirection, and the inspector, if one distribution has a random placement of decomposition elements and the other has a neighborhood placement, e.g., based on a domain decomposition, the topology-based distribution will provide better performance. More precisely, for a large enough number of DECOMPOSITION elements per processor, the worst case communication performance of a topology-based distribution will be better than the expected communication performance for a random distribution.

Intuitive Argument: In the target physical simulation applications, the communication necessary for each element is in it's neighborhood. In the random distribution, no advantage is taken of this knowledge, every element is equally likely to be on any processor. If instead, the neighborhood property is preserved as much as possible on each processor, then there will be minimal communication across processors.

Proof: For 1-dimensional DECOMPOSITIONs, each interior element has 2 neighbors. Let there be m DECOMPOSITION elements per processor.

With a random distribution, for each element the probability of needing to communicate to reference a neighbor value is $2 * \frac{p-1}{p}$. Hence, for any processor, the expected number of communications is $2 * m * \frac{p-1}{p}$. For all processors, the expected number of communications is $2 * m * (p-1)$.

With a topology-based distribution, the worst case is for a processor storing an interior section of the distribution. On such a processor, there are 2 elements that must access 1 neighbor each on a different processor. Hence, the communication for a 1-dimensional DECOMPOSITION distributed in this manner is $2 * (p-1)$.

Therefore, for $m > 1$, the topology-based distribution has fewer communications than expected with a random distribution.

For 2-dimensional DECOMPOSITIONs, each interior element has 8 neighbors. Let there be $m * n$ DECOMPOSITION elements per processor.

With a random distribution, for each element the probability of needing to communicate to reference a neighbor value is $8 * \frac{p-1}{p}$. Hence, for any processor, the expected number of communications is $8 * m * n * \frac{p-1}{p}$. For all processors, the expected number of communications is $8 * m * n * (p-1)$.

With a topology-based distribution, the worst case is for a processor storing an interior section of the distribution. On such a processor, there are $2 * (m + n) - 4$ elements that must access 3 neighbors each on other processors and 4 elements that access 5 elements on other processors for an upper bound of $6 * (m + n) + 8$. Hence, the number of communications for a 2-dimensional DECOMPOSITION distributed in this manner is bounded by $p * [6 * (m + n) + 8]$.

Therefore, for $m, n > 3$, the topology-based distribution has fewer communications than expected with a random distribution.

For 3-dimensional DECOMPOSITIONs, each interior element has 26 neighbors. Let there be $l * m * n$ DECOMPOSITION elements per processor.

With a random distribution, for each element the probability of needing to communicate to reference a neighbor value is $26 * \frac{p-1}{p}$. Hence, for any processor, the expected number of communications is $26 * l * m * n * \frac{p-1}{p}$. For all processors, the expected number of communications is $26 * l * m * n * (p - 1)$.

With a topology-based distribution, the worst case is for a processor storing an interior section of the distribution. On such a processor, there are $2 * [(l - 2) * (m - 2) + (l - 2) * (n - 2) + (m - 2) * (n - 2)]$ elements that must access 9 neighbors each on other processors, $4 * [(l - 2) + (m - 2) + (n - 2)]$ elements that must access 15 neighbors each on other processors, and 8 elements that access 19 elements on other processors for an upper bound of $18 * (lm + ln + mn) - 12 * (l + m + n) + 8$. Hence, the number of communications for a 3-dimensional DECOMPOSITION distributed in this manner is bounded by $p * 18 * (lm + ln + mn) - 12 * (l + m + n) + 8$.

Therefore, for $l, m, n > 3$, the topology-based distribution has fewer communications than expected with a random distribution.

Throughout this proof, it was only required that p be greater than one, but as p increases the gap between the number of communications for a topology-based distribution and the expected number for a random distribution increases.

Theorem A.2 Given two distributions with the same overheads due to load balance, communication, indirection, and the inspector, if one distribution has a single block of elements from one decomposition and the other has a number of blocks of elements from various decompositions then the distribution with one block per processor will provide better performance.

Proof: This is easy to see as no context switching is performed in the single block case whereas it is necessary in the multiblock case.

Theorem A.3 Given two distributions with the same overheads due to context switching, communication, indirection, and the inspector, the distribution with the better load balance will outperform the other distribution.

Proof: This is straigtforward as the runtime is based, in part, on the maximum over all of the processors' computational times, which increases with decreasing load balance, by definition.

Theorem A.4 Consider two distributions with the same overhead due to context switching, load balance, indirection, and the inspector and the same number of communications, with the same blocking capabilities. A distribution with communication between close together processors will outperform a distribution where communication takes place between distant processors (up to, but not past, the point where general communication takes over). This implies that maintaining the neighborhoods when mapping blocks of decomposition elements to processors is important.

Proof: Let h_1 be strictly less than $hops_{general} - 1$. Consider the communication cost, C_T, for b bytes. For h_1,

$$C_T(h_1, b) = constant + h_1 * cost_{neighbor} + b * h_1 * constant_{buffering},$$

while for $h_1 + 1$,

$$C_T(h_1 + 1, b) = constant + (h_1 + 1) * cost_{neighbor} + b * (h_1 + 1) * constant_{buffering},$$

or

$$\begin{aligned} C_T(h_1 + 1, b) &= constant + h_1 * cost_{neighbor} + b * h_1 * constant_{buffering} \\ &\quad + cost_{neighbor} + b * constant_{buffering}. \end{aligned}$$

Hence,

$$C_T(h_1, b) - C_T(h_1 + 1, b) = cost_{neighbor} + b * constant_{buffering}.$$

Since $cost_{neighbor} \geq 0$ and $b,\ constant_{buffering} > 0$, this shows that, in this model, more distant communication implies longer runtime.

A.6.2 Extended Model Theorems

Now, some problems are considered that require extension of the original modeling assumptions.

Consider the case of dynamic physical simulation problems where the computational cost per element in a decomposition is not constant and may change over time. If a static distribution is used, then perfect load balance will not be maintained over time. Since the computation per element can change at each iteration of the simulation, communication overhead may be required to achieve a reasonable load balance. One approach is to break blocks into sub-blocks and map the sub-blocks onto the processor mesh in a wrapped fashion, e.g., see Figure A.3.

Using this approach, the communication overhead is increased somewhat but the load balance may be improved. This approach applies particularly well to those

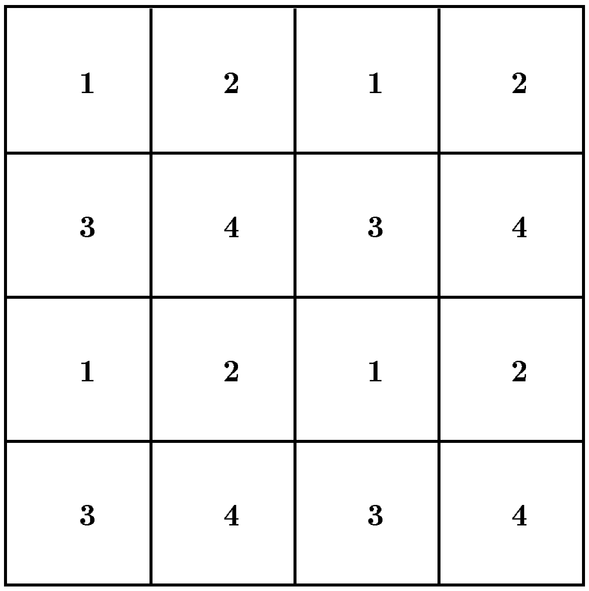


Figure A.3 Sub-block distribution mapping

applications where the difference in computation between elements can be large and the expensive type of computation occurs in large clusters of elements. In this case, the sub-blocks should be some fraction of the normal cluster size (maybe size $1/p$, where p is the number of processors).

When the expensive type of computation is spread randomly (not clustered) over the elements, then you may as well optimize the communication and just balance the number of elements per processor. This is because at any step each element is just as likely as any other to doing an expensive computation. Therefore, it does not matter how the elements are grouped in a static partition, each partitioning can be as bad as any other. In addition, if the state changes (between inexpensive and expensive) are unpredictable, this same argument implies dynamic partitioning is not beneficial.

A.7 Appendix Summary

A model of parallel computation has been presented that can be used in comparing different distributions of meshes in ICRM problems.

Theorems have been proven about the model. These theorems were used in algorithm development for ICRM distribution. In particular, distributing large meshes over all processors, packing small meshes according to coupling, and using packing for load balance in medium mesh problems are direct consequences of the theorems.